← Qihong Ruan

Stanford CS336 — Language Modeling from Scratch

Comprehensive lecture notes for the Spring 2025 edition of Stanford's CS336. Each set of notes is generated from the entire lecture video — the slides, whiteboard math, and live code — combined with the official transcript. Covers tokenization, architectures, MoE, GPUs & kernels, parallelism, scaling laws, inference, evaluation, data, and alignment (SFT/RLHF/RL).

17/17 lectures · Course playlist ↗ · Course site ↗

⭐ Star on GitHub

Algorithm Deep Dives

🧠 Beyond the slides — the trickiest algorithmic questions across the course (BPE, attention, MoE routing, FlashAttention, parallelism, scaling laws, KV-cache, PPO/DPO/GRPO), worked through with full derivations and intuition. →

Assignments — Do the Work

🛠️ The lectures are the easy part; the real learning is in the 5 implementation-heavy assignments. Study guides for each — what you build, what makes it hard, and how to approach it — **guidance, not solutions**. →

LECTURE 01 · 1:18:59

Overview and Tokenization

CS336: Language Models From Scratch aims to teach students how to build language models from

LECTURE 02 · 1:19:22

PyTorch, Resource Accounting

This lecture focuses on building language models from scratch using PyTorch, emphasizing efficiency

the ground up, emphasizing a deep understanding of the un...

and resource accounting (memory and compute).

LECTURE 03 · 1:27:03

Architectures, Hyperparameters

Modern transformer architectures have converged on pre-norm, RMSNorm, and gated linear units (GLUs) for better stability and performance.

LECTURE 04 · 1:22:04

Mixture of Experts

LECTURE 05 · 1:14:21

GPUs

GPUs are massively parallel processors optimized for throughput, not latency, by having many simple compute units (SMs) orchestrated by minimal contro...

LECTURE 06 · 1:20:22

Kernels, Triton

LECTURE 07 · 1:24:42

Parallelism 1

LECTURE 08 · 1:15:10

Parallelism 2

Data Transfer Bottleneck: The primary challenge in distributed training is minimizing data transfer bottlenecks, as moving data between different memo...

LECTURE 09 · 1:05:18

Scaling Laws 1

LECTURE 10 · 1:22:52

Inference

Inference is a critical part of language model deployment, distinct from training due to its memory-limited and dynamic nature.

LECTURE 11 · 1:18:13

Scaling Laws 2

Scaling Laws in Practice: Modern large language model (LLM) builders use scaling laws as a core

LECTURE 12 · 1:20:48

Evaluation

Evaluation is a complex topic that goes beyond simple metrics, influencing how language models

part of their design process, but the details of these...

are built and used.

LECTURE 13 · 1:19:06

Data 1

LECTURE 14 · 1:19:12

Data 2

[0:00] Lecture 14: Data 2 - Deep Dive into Data Processing

LECTURE 15 · 1:14:51

Alignment — SFT/RLHF

LECTURE 16 · 1:20:32

Alignment — RL 1

RLHF Limitations: Overoptimization and mode collapse are significant problems in RLHF, often stemming from the noisiness and complexity of human prefe...

LECTURE 17 · 1:16:09

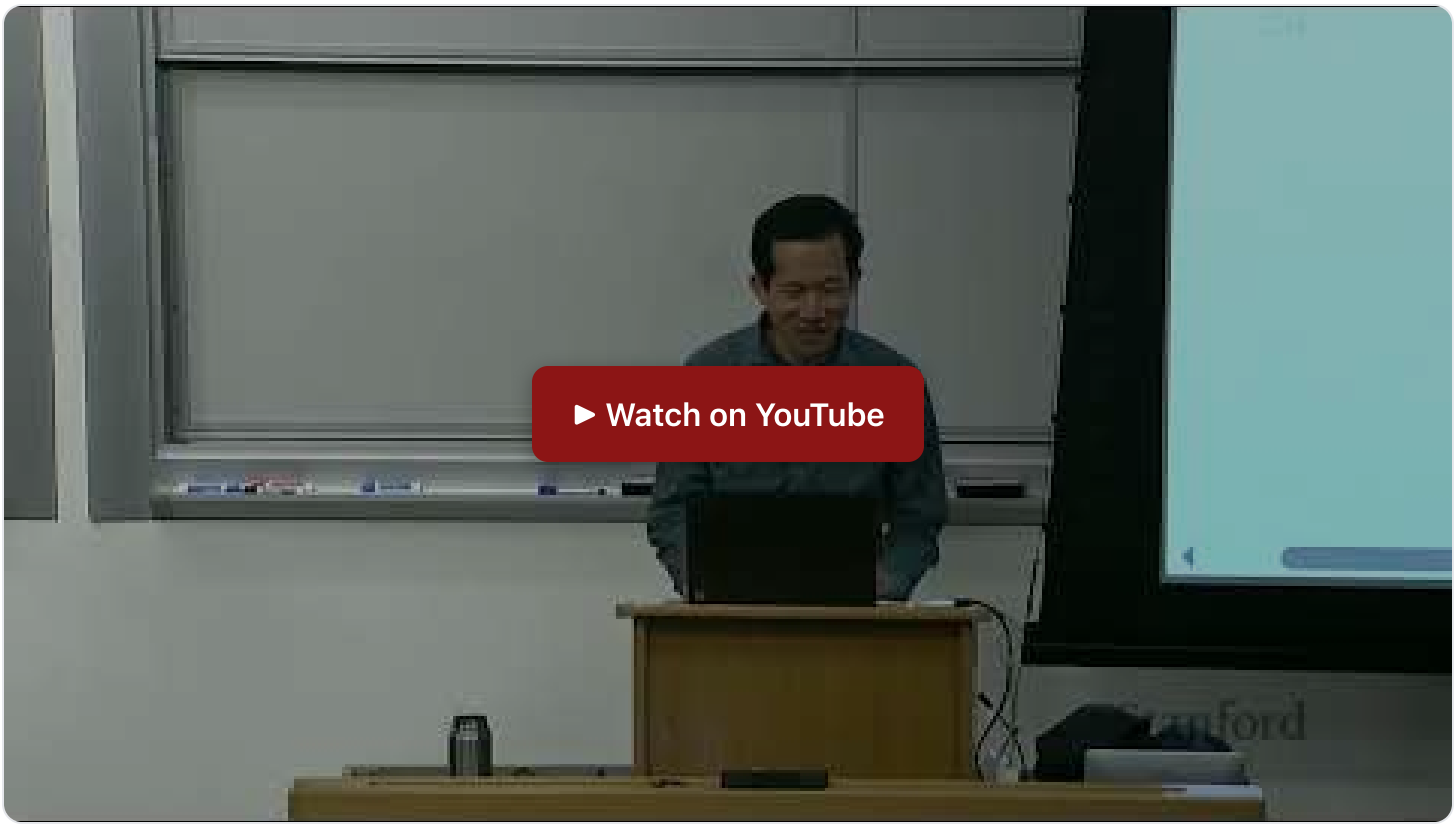
Alignment — RL 2

Reinforcement Learning (RL) is key to surpassing human abilities in Language Models (LMs).

Lecture 01: Overview and Tokenization

Stanford CS336 · Spring 2025 · 1:18:59 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



TL;DR

- CS336: Language Models From Scratch aims to teach students how to build language models from the ground up, emphasizing a deep understanding of the underlying technology.
- The course addresses the growing disconnect between researchers and the core technology, promoting a "build it to understand it" philosophy.
- Key concepts include understanding mechanics, developing a scaling mindset, and recognizing the limitations of intuition in large-scale AI.

- The course covers tokenization, model architecture, training, data processing, and alignment, with a strong focus on efficiency and resource accounting.
- Assignments involve implementing core components from scratch, benchmarking performance, and optimizing models within given compute budgets.

Key Concepts

- **Language Model Building Pipeline**: End-to-end process from data to modeling.
- **Mechanics, Mindset, Intuitions**: Three types of knowledge for understanding technology.
- **Scaling Laws**: How model performance changes with scale, and optimizing resource allocation.
- **Algorithmic Efficiency**: Importance of efficient algorithms, especially at large scales.
- **Compute vs. Data Constraints**: Different regimes of resource limitations influencing design decisions.
- **Tokenization**: Converting raw text into numerical representations for models.
- **Byte Pair Encoding (BPE)**: An adaptive tokenization algorithm based on corpus statistics.
- **Transformer Architecture**: The foundational model for modern language models.
- **Model Parallelism**: Techniques for distributing models across multiple GPUs.
- **Inference Optimization**: Strategies for efficient generation with trained models.
- **Data Curation and Processing**: Steps to prepare raw data for language model training.
- **Alignment**: Making language models follow instructions, tune style, and incorporate safety.
- **Supervised Fine-tuning (SFT)**: Training models on human-annotated prompt-response pairs.
- **Reinforcement Learning from Human Feedback (RLHF)**: Algorithms like PPO, DPO, GRPO for aligning models with human preferences.

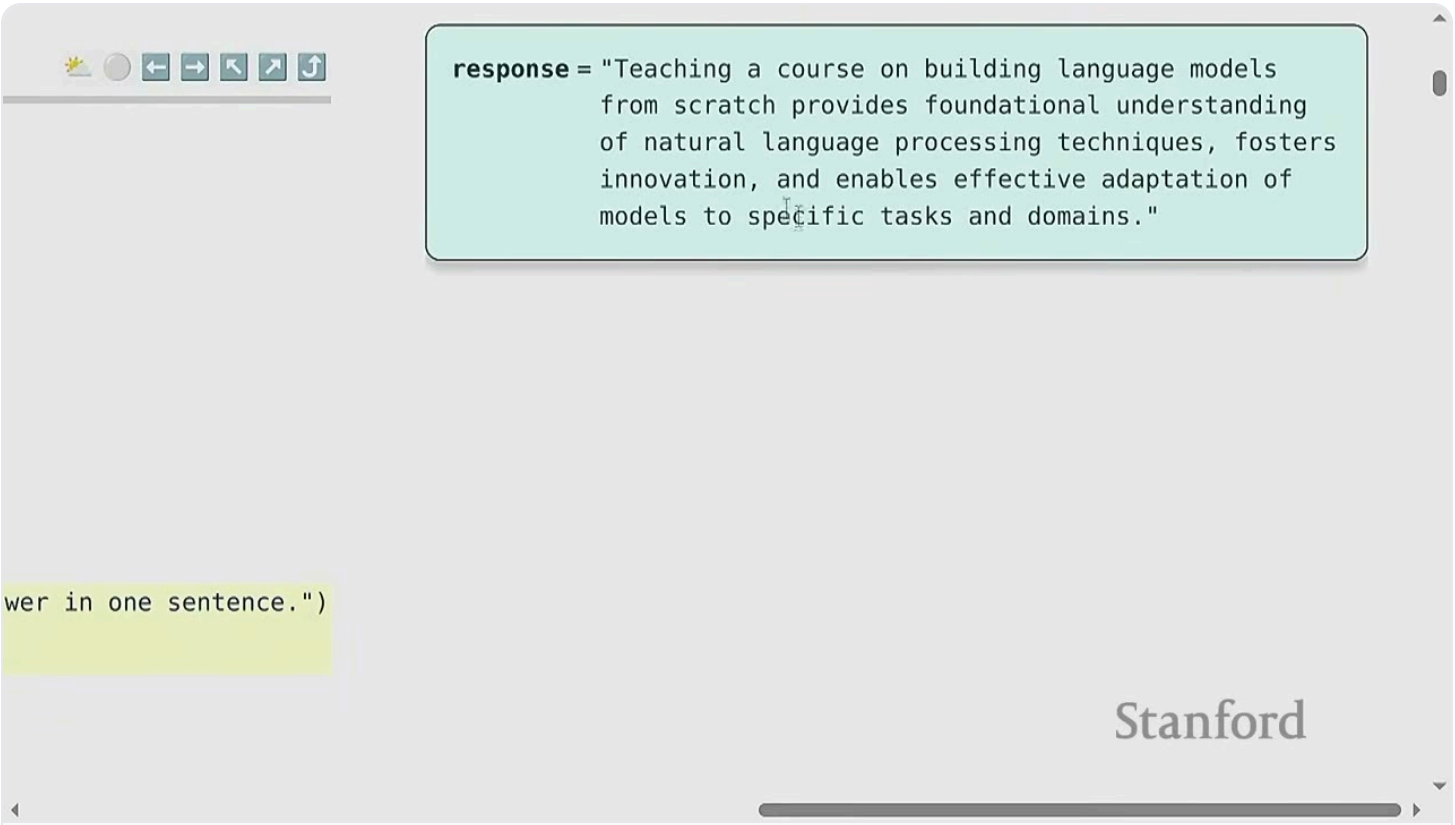
[0:00] Introduction to CS336: Language Models From Scratch

The course CS336: Language Models From Scratch is being taught for the second time. The lectures are made available on YouTube to allow global access to the content.

[0:05] Course Staff Introductions

- **Percy Liang**: Instructor. Emphasizes the excitement of seeing the entire language model building pipeline end-to-end, including data systems and modeling.
- **Tatsunori Hashimoto**: Co-instructor. Highlights the course's focus on deep technical understanding by building from scratch.
- **Rohith Kudlipudi**: CA. Acknowledges having failed the class previously, humorously noting that "anything is possible."
- **Neil Band**: CA. A PhD student interested in synthetic data, language models, and reasoning.
- **Marcel Rød**: CA. A PhD student working on health, and a top performer in previous iterations of the course.

[1:42] Why We Made This Course



2:23 A code snippet defining a response variable with a long string value.

The motivation for creating this course stems from a perceived crisis in the field of AI research.

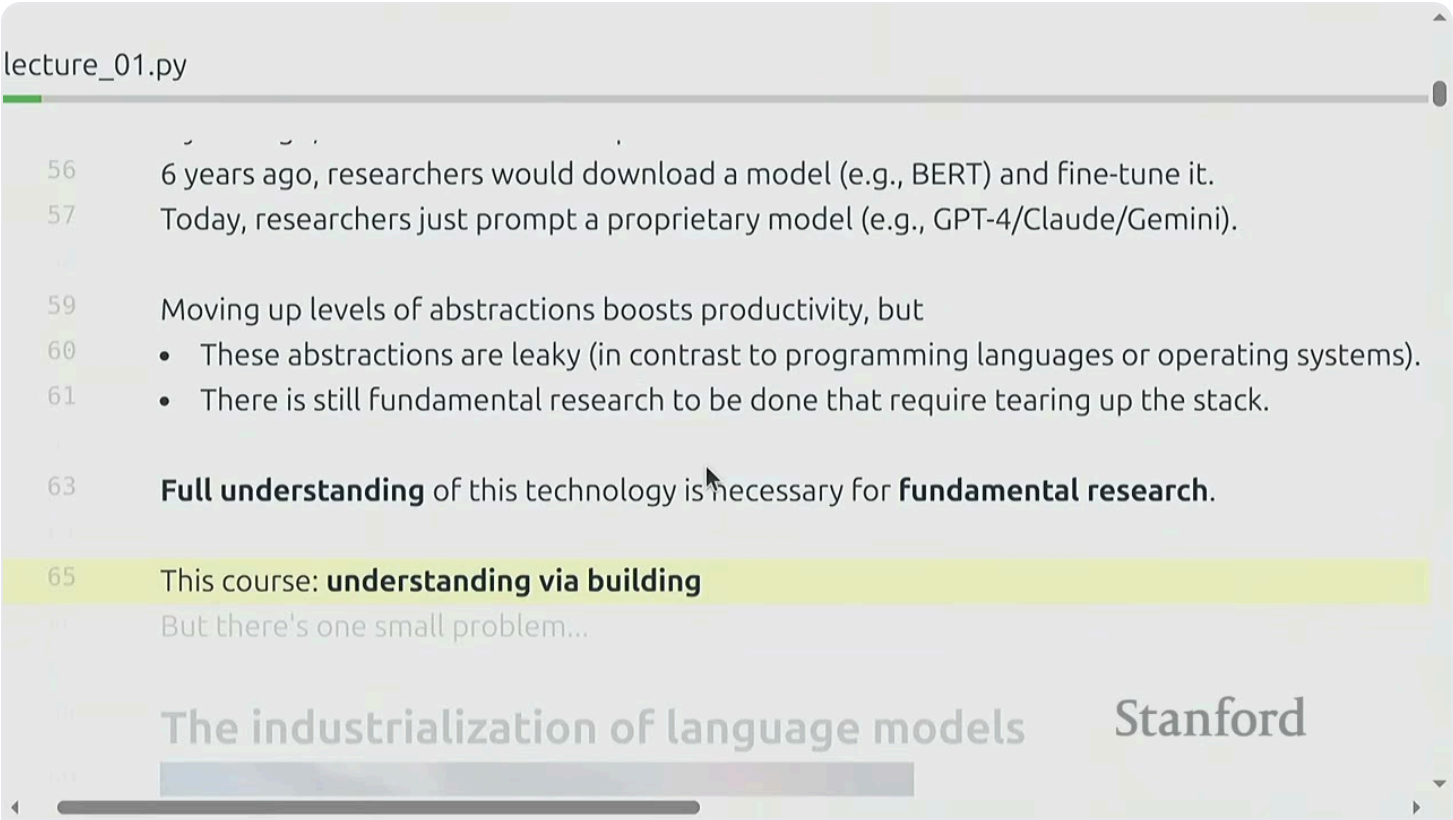
- **Problem**: Researchers are becoming increasingly disconnected from the underlying technology.
  - **8 years ago**: Researchers would implement and train their own models.
  - **6 years ago**: Researchers would download and fine-tune models like BERT.

- **Today**: Many researchers primarily interact with proprietary models (e.g., GPT-4, Claude, Gemini) through prompting.

This shift isn't inherently bad, as layers of abstraction boost productivity. However, these abstractions are "leaky," meaning that a deep understanding of the underlying mechanisms is crucial for fundamental research and pushing the boundaries of the field.

- **Course Philosophy**: To truly understand the technology, you have to build it.

[3:59] The Industrialization of Language Models

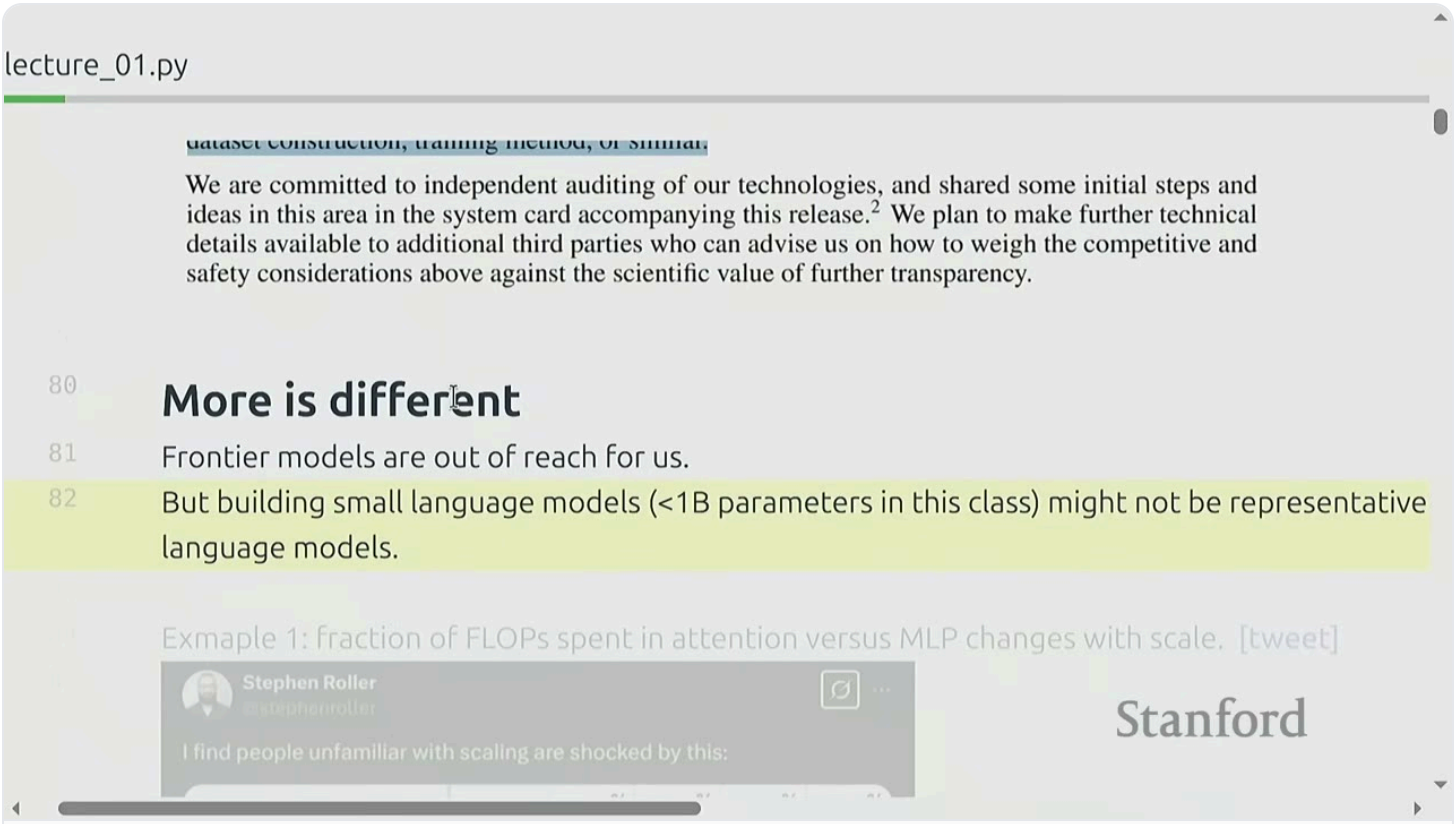


3:58 Text explaining the course's focus on understanding language models via building them.

A significant challenge in building language models is their industrial scale: - **GPT-4**: Rumored to have 1.8 trillion parameters, costing $100 million to train. - **xAI**: Building clusters with 200,000 H100 GPUs to train Grok. - **Stargate (OpenAI, Nvidia, Oracle)**: Supposedly a $500 billion investment over 4 years.

Furthermore, there's a lack of public details on how these frontier models are built. The GPT-4 technical report explicitly states that due to competitive landscape and safety limitations, they disclose no details about their architecture, training data, or methods.

[4:54] "More is Different"

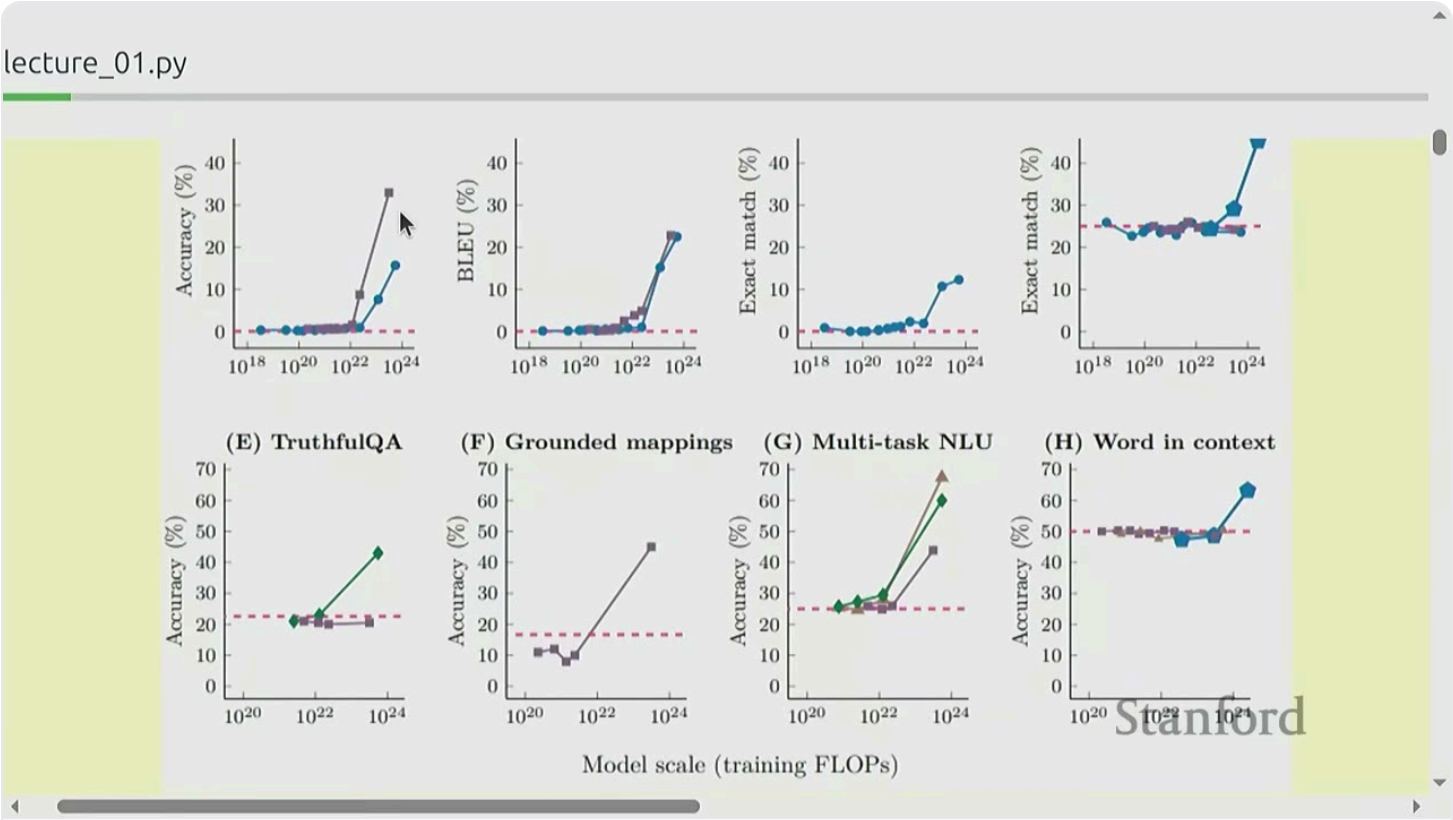


5:16 Text discussing the limitations of building small language models for research.

Frontier models are often out of reach for individual researchers. The small language models built in this class (e.g., <1B parameters) might not be representative of large models.

- **Example 1: Fraction of FLOPs spent in attention vs. MLP changes with scale.**
    - For small models, FLOPs are roughly comparable between attention and MLP layers.
    - For large models (e.g., 175B parameters), MLP layers dominate FLOPs.
    - **Implication**: Optimizing attention at small scales might be optimizing the wrong thing for large scales.
- **Example 2: Emergence of behavior with scale (Wei+ 2022).**
    - Many tasks show a "phase transition" where accuracy remains low for a long time as training FLOPs increase, then suddenly jumps when a certain scale is reached.
    - **Implication**: Research at small scales might lead to incorrect conclusions about model capabilities if emergent behaviors only appear at larger scales.

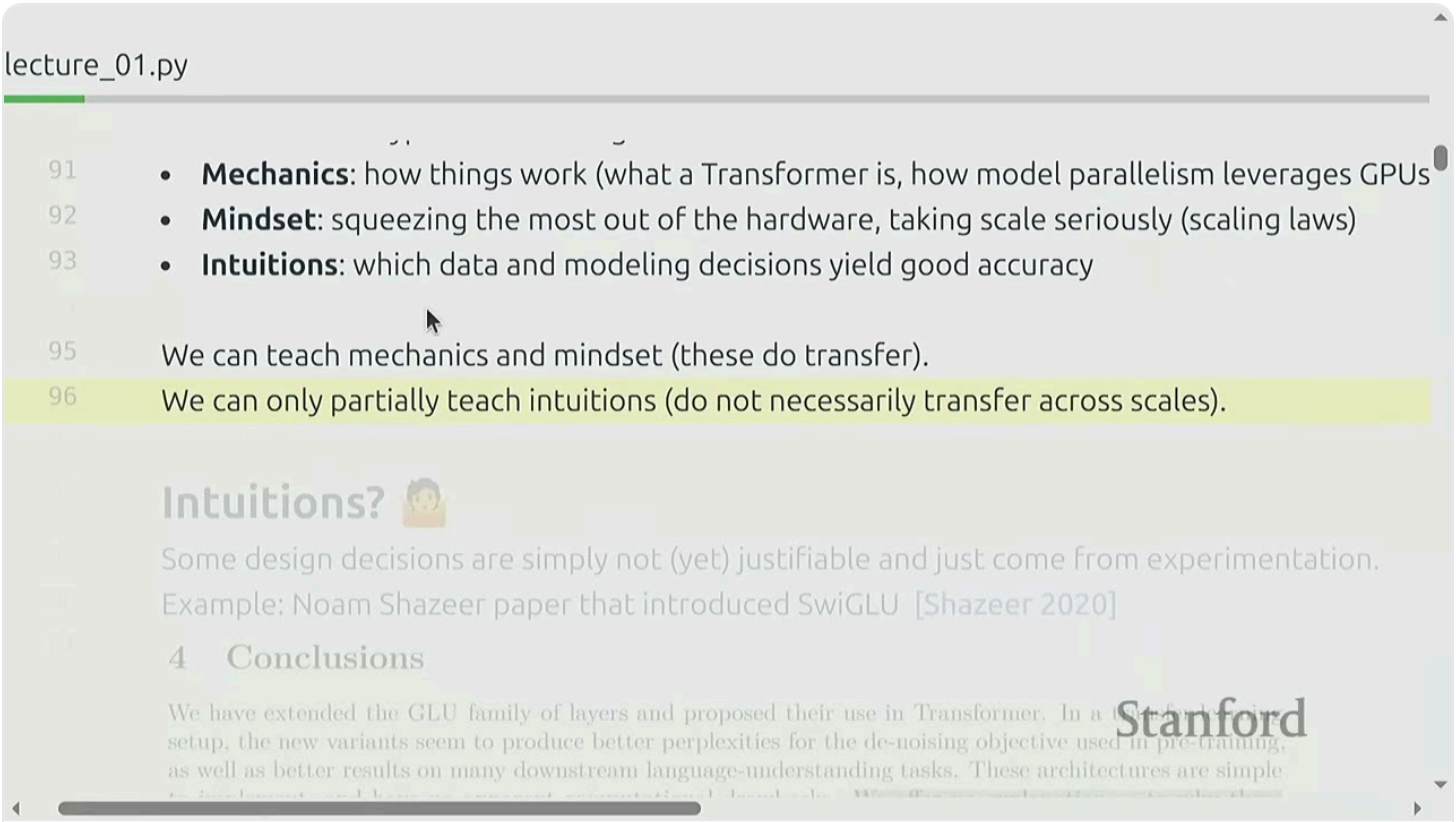
[7:00] What Can We Learn That Transfers to Frontier Models?



6:47 A graph showing model accuracy across various tasks as a function of model scale.

There are three types of knowledge: 1. **Mechanics**: How things work (e.g., what a Transformer is, how model parallelism leverages GPUs). This can be taught effectively. 2. **Mindset**: Squeezing the most out of hardware, taking scaling seriously (scaling laws). This is crucial, as the scaling mindset pioneered by OpenAI led to the current generation of AI models. 3. **Intuitions**: Which data and modeling decisions yield good accuracy. This can only be partially taught, as intuitions developed at small scales may not transfer to large scales.

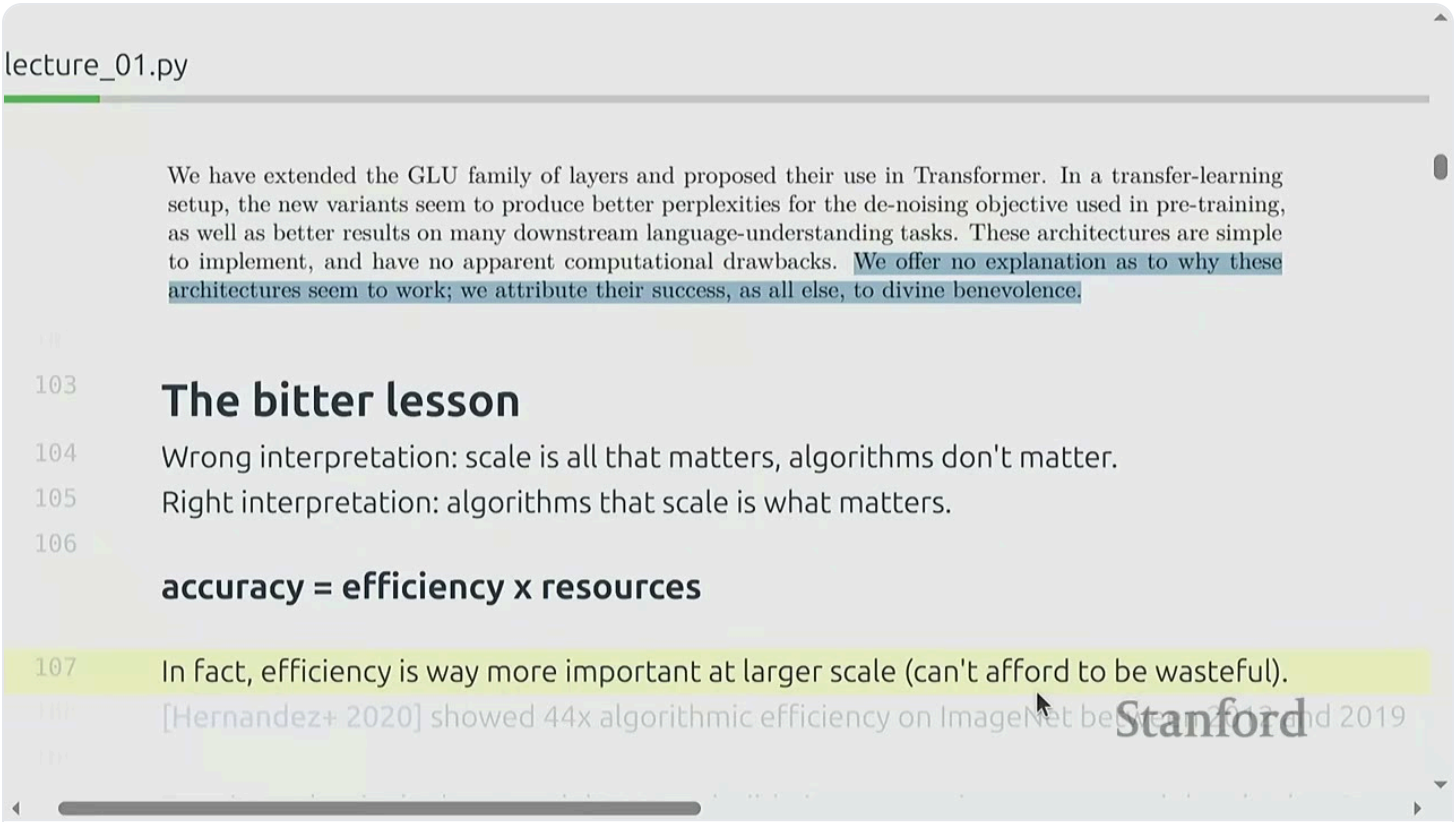
[8:44] Intuitions?



8:39 Text highlighting the teachable aspects (mechanics, mindset) versus intuitions in ML.

Some design decisions are not purely justifiable and come from experimentation. For example, the SwiGLU activation function (Shazeer 2020) was adopted due to its empirical success, with the paper honestly stating, "We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence." This highlights the limits of current understanding.

[9:33] The Bitter Lesson



10:27 Text explaining the "bitter lesson" of accuracy, efficiency, and resources in ML.

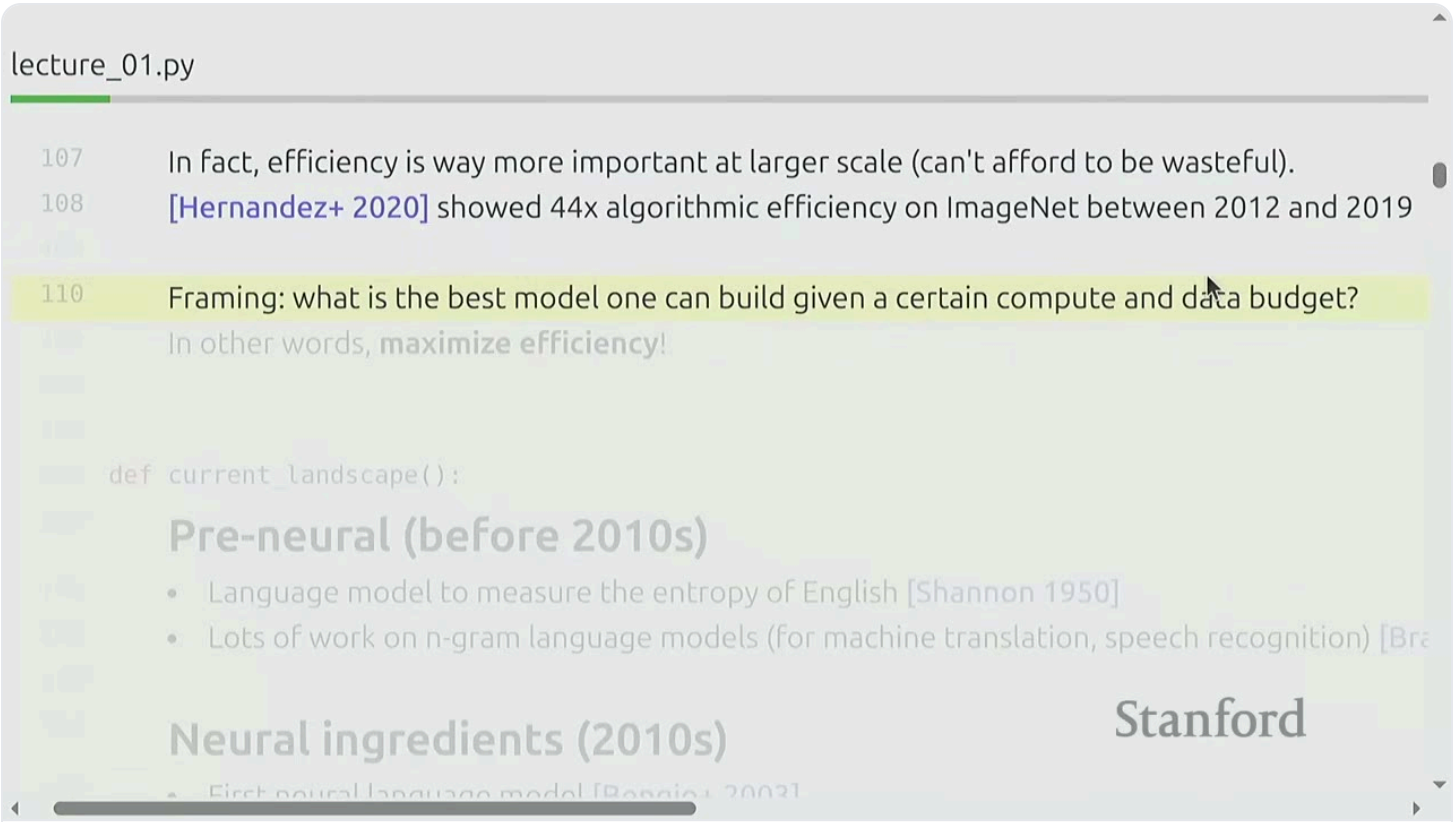
- **Wrong Interpretation**: Scale is all that matters, algorithms don't matter.
- **Right Interpretation**: Algorithms that scale is what matters.

Accuracy is a product of efficiency and resources ($accuracy = efficiency \times resources$). Efficiency is even more important at larger scales because one cannot afford to be wasteful.

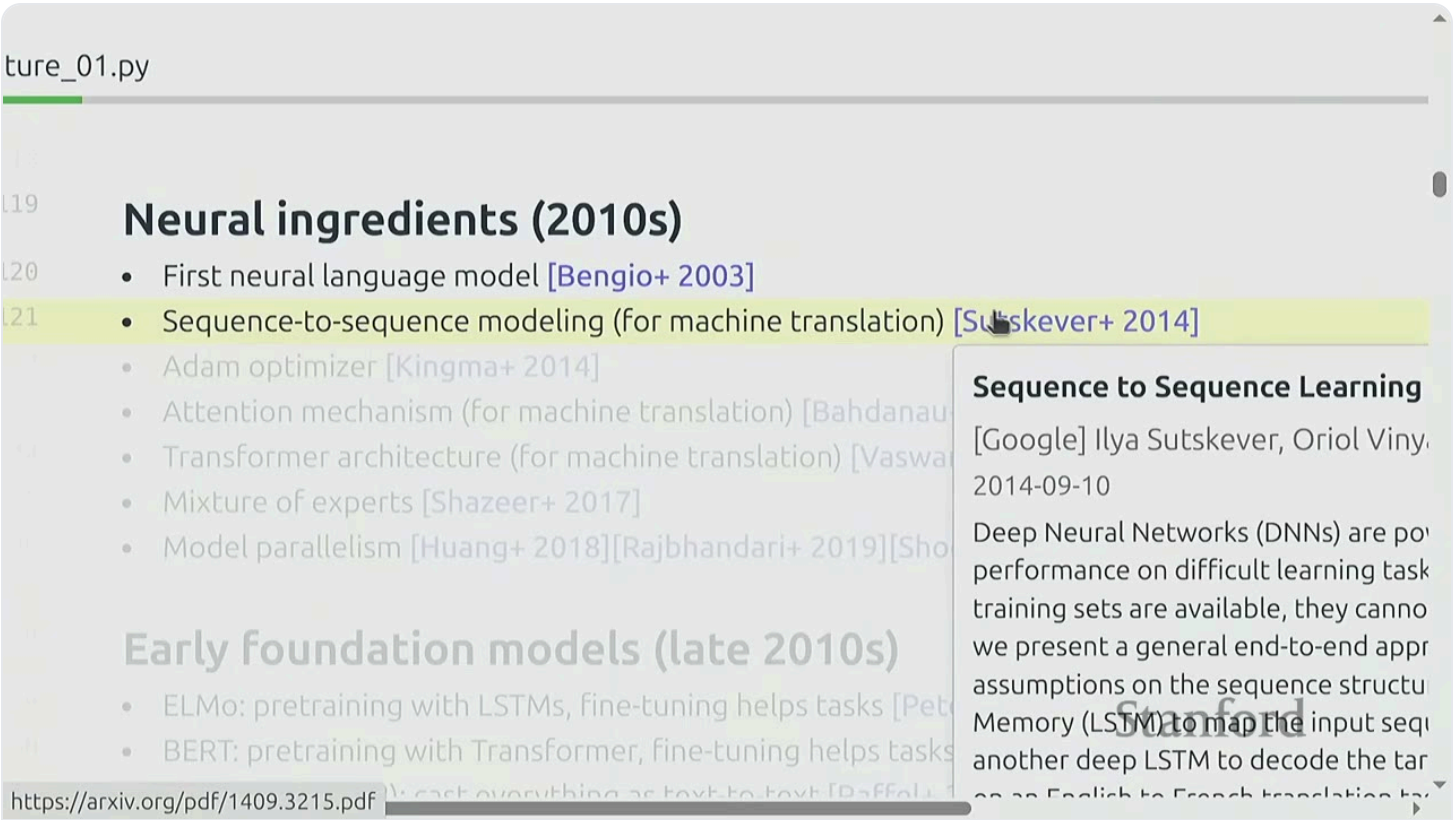
- **Hernandez+ 2020**: Showed 44x algorithmic efficiency improvement on ImageNet between 2012 and 2019. This improvement is faster than Moore's Law, demonstrating that algorithmic innovation is critical.

- **Framing**: What is the best model one can build given a certain compute and data budget? In other words, maximize efficiency! This question is relevant at any scale.

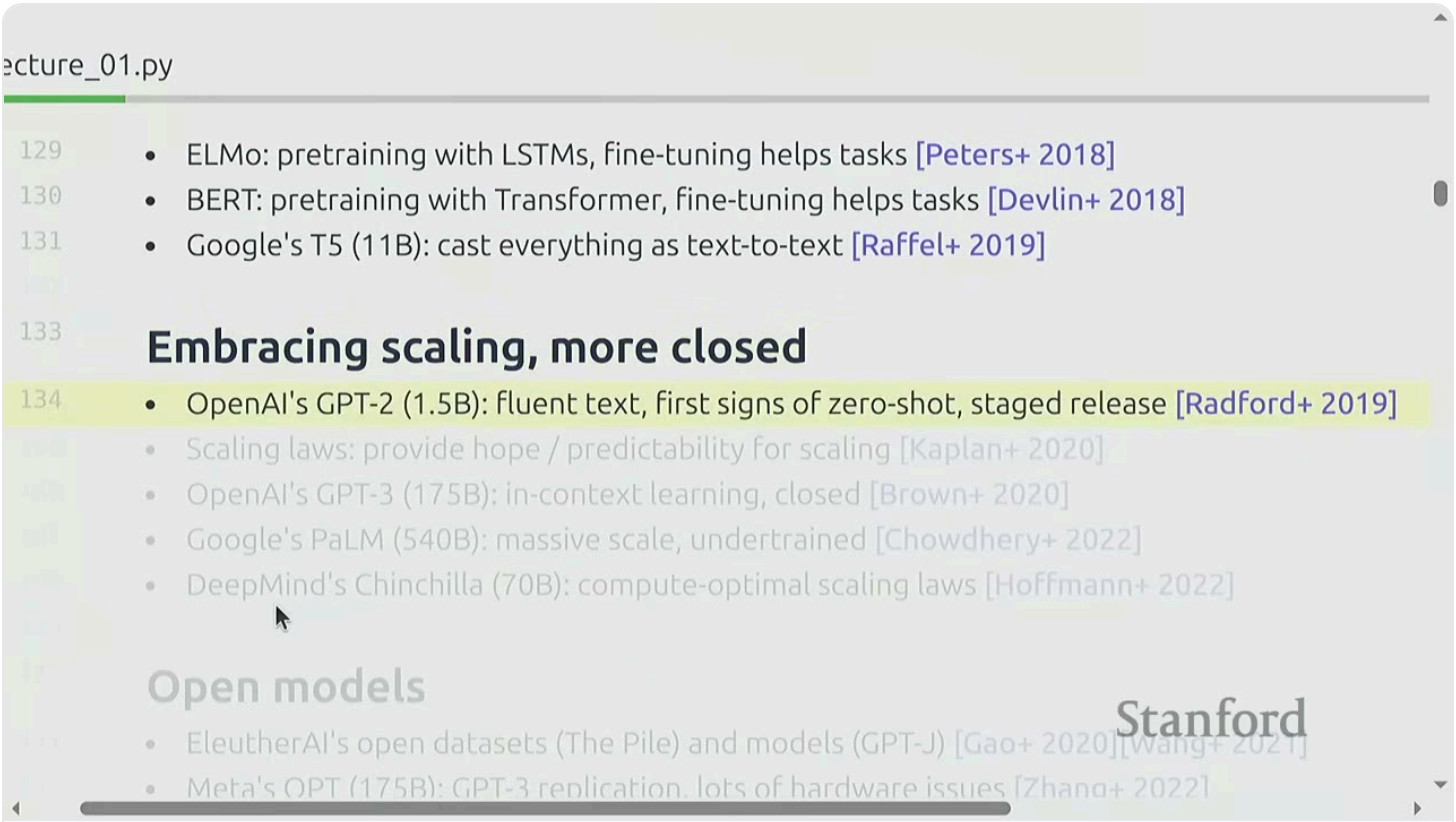
[11:17] Current Landscape (History)



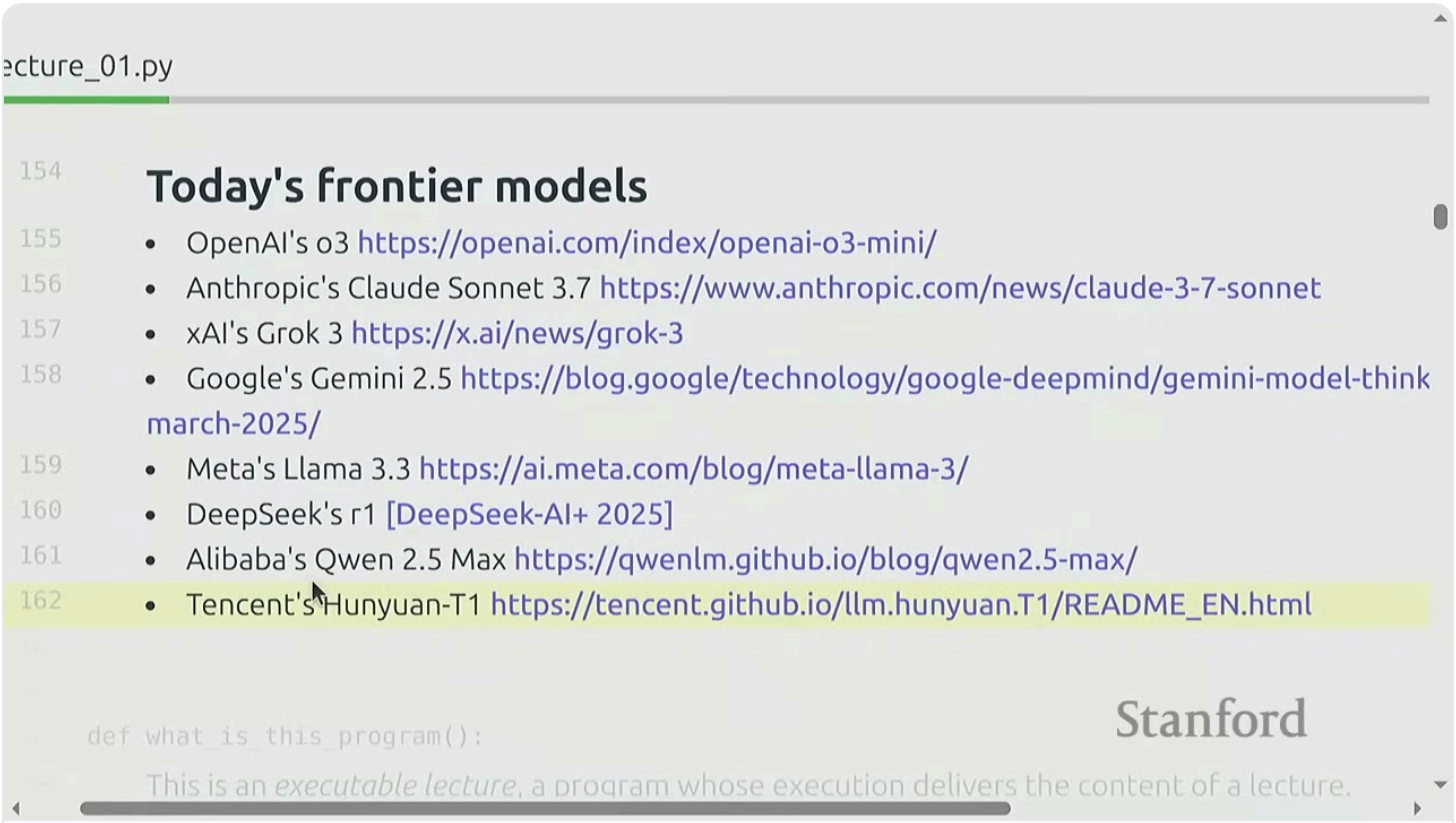
11:50 Text defining the framing of maximizing efficiency given compute and data budgets.



13:43 A list of neural ingredients from the 2010s, highlighting sequence-to-sequence modeling.



15:13 A list of early foundation models, including OpenAI's GPT-2 and Google's PaLM.



17:22 A list of today's frontier models, including OpenAI's o3 and Google's Gemini 2.5.

- **Pre-neural (before 2010s)**:

    - Language models were used to measure the entropy of English (Shannon 1950).
    - Lots of work on n-gram language models for machine translation and speech recognition (Brants+ 2007). Google trained 5-gram models on 2 trillion tokens, a scale only recently matched by modern LMs.

- **Neural Ingredients (2010s)**:

    - First neural language model (Bengio+ 2003).
    - Sequence-to-sequence modeling (Sutskever+ 2014).
    - Adam optimizer (Kingma+ 2014) - still widely used.
    - Attention mechanism (Bahdanau+ 2014) - developed for machine translation.
    - Transformer architecture (Vaswani+ 2017) - "Attention Is All You Need".
    - Mixture of experts (Shazeer+ 2017).
    - Model parallelism (Huang+ 2018, Rajbhandari+ 2019, Shoeybi+ 2019) - techniques to train models with 100B+ parameters.
    - All these ingredients were in place by 2020.

- **Early Foundation Models (late 2010s)**:

    - ELMo: pretraining with LSTMs, fine-tuning helps tasks (Peters+ 2018).
    - BERT: pretraining with Transformers, fine-tuning helps tasks (Devlin+ 2018).
    - Google's T5 (11B): cast everything as text-to-text (Raffel+ 2019).
    - These models were very exciting at the time, paving the way for larger models.

- **Embracing Scaling, More Closed**:

    - OpenAI's GPT-2 (1.5B): fluent text, first signs of zero-shot, staged release (Radford+ 2019).
    - Scaling laws: provide hope/predictability for scaling (Kaplan+ 2020).
    - OpenAI's GPT-3 (175B): in-context learning, closed (Brown+ 2020).
    - Google's PaLM (540B): massive scale, undertrained (Chowdhery+ 2022).
    - DeepMind's Chinchilla (70B): compute-optimal scaling laws (Hoffmann+ 2022).
    - This era saw a shift towards closed, proprietary models.

- **Open Models**:

    - EleutherAI's open datasets (The Pile) and models (GPT-J) (Wang+ 2021).
    - Meta's OPT (175B): GPT-3 replication, lots of hardware issues (Zhang+ 2022).
    - Hugging Face / BigScience's BLOOM: focused on data sourcing (Workshop+ 2022).

    - Meta's Llama models (Touvron+ 2023).
    - Alibaba's Qwen models (Qwen+ 2024).
    - DeepSeek's models (DeepSeek-AI+ 2024).
    - AI2's OLMo models (Groeneveld+ 2024).
    - This trend towards open models allows researchers to study and build upon frontier models.

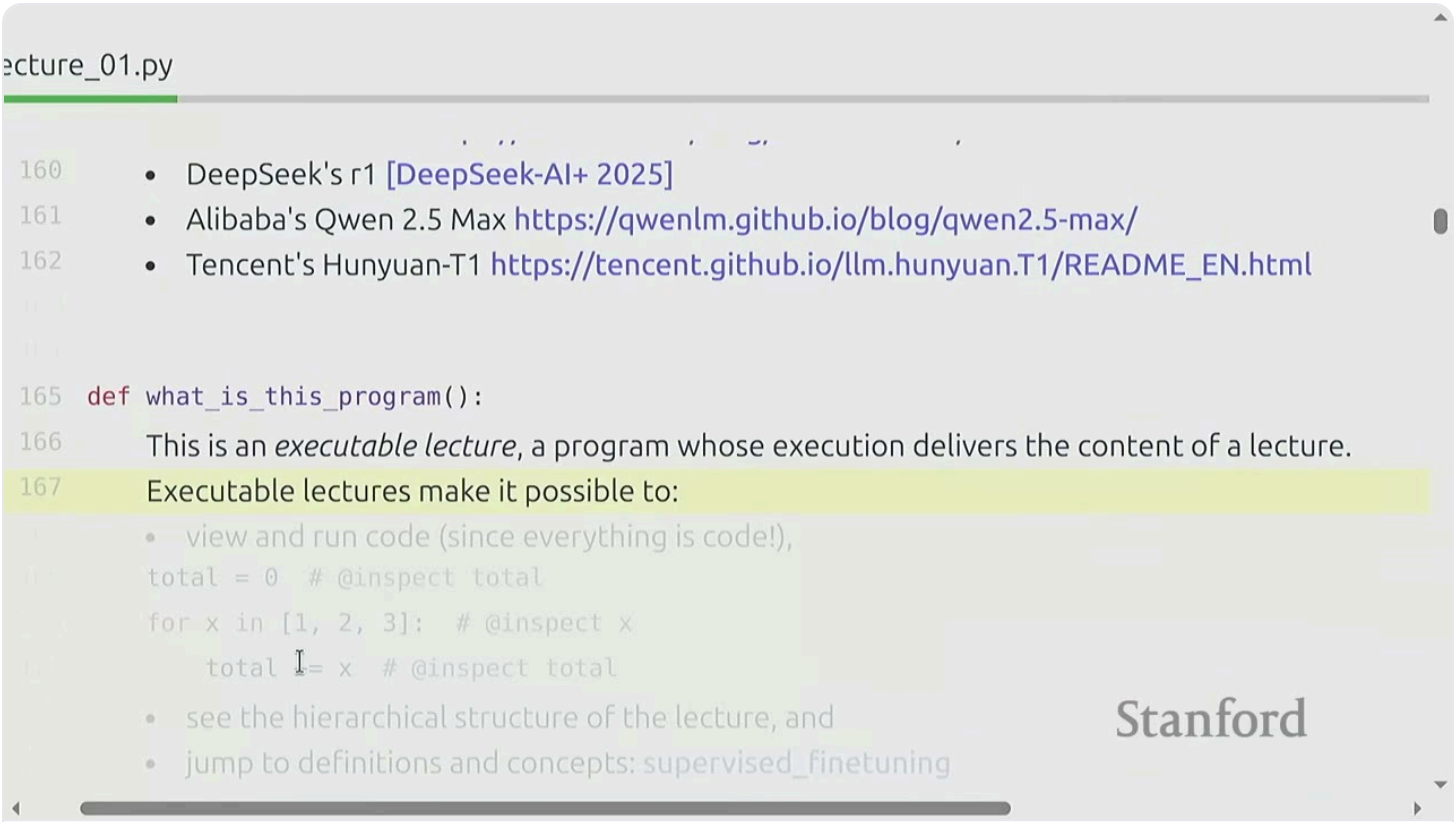
- **Levels of Openness**:

    - **Closed models**: (e.g., GPT-4o), API access only [OpenAI+ 2023].
    - **Open-weight models**: (e.g., DeepSeek), weights available, paper with architecture details, some training details, no data details [DeepSeek-AI+ 2024].
    - **Open-source models**: (e.g., OLMo), weights and data available, paper with most details (but not the rationale, failed experiments) [Groeneveld+ 2024].

- **Today's Frontier Models**:

    - OpenAI's GPT-4o, Anthropic's Claude 3.5 Sonnet, xAI's Grok 3, Google's Gemini 2.5, Meta's Llama 3.3, DeepSeek's r1, Alibaba's Qwen 2.5 Max, Tencent's Hunyuan-T1.
    - The current landscape is dominated by these large, often closed, models. The course will revisit early ingredients and trace how techniques work, then move towards best practices using information from the open community and by reading between the lines of closed models.

[18:04] What is this program?



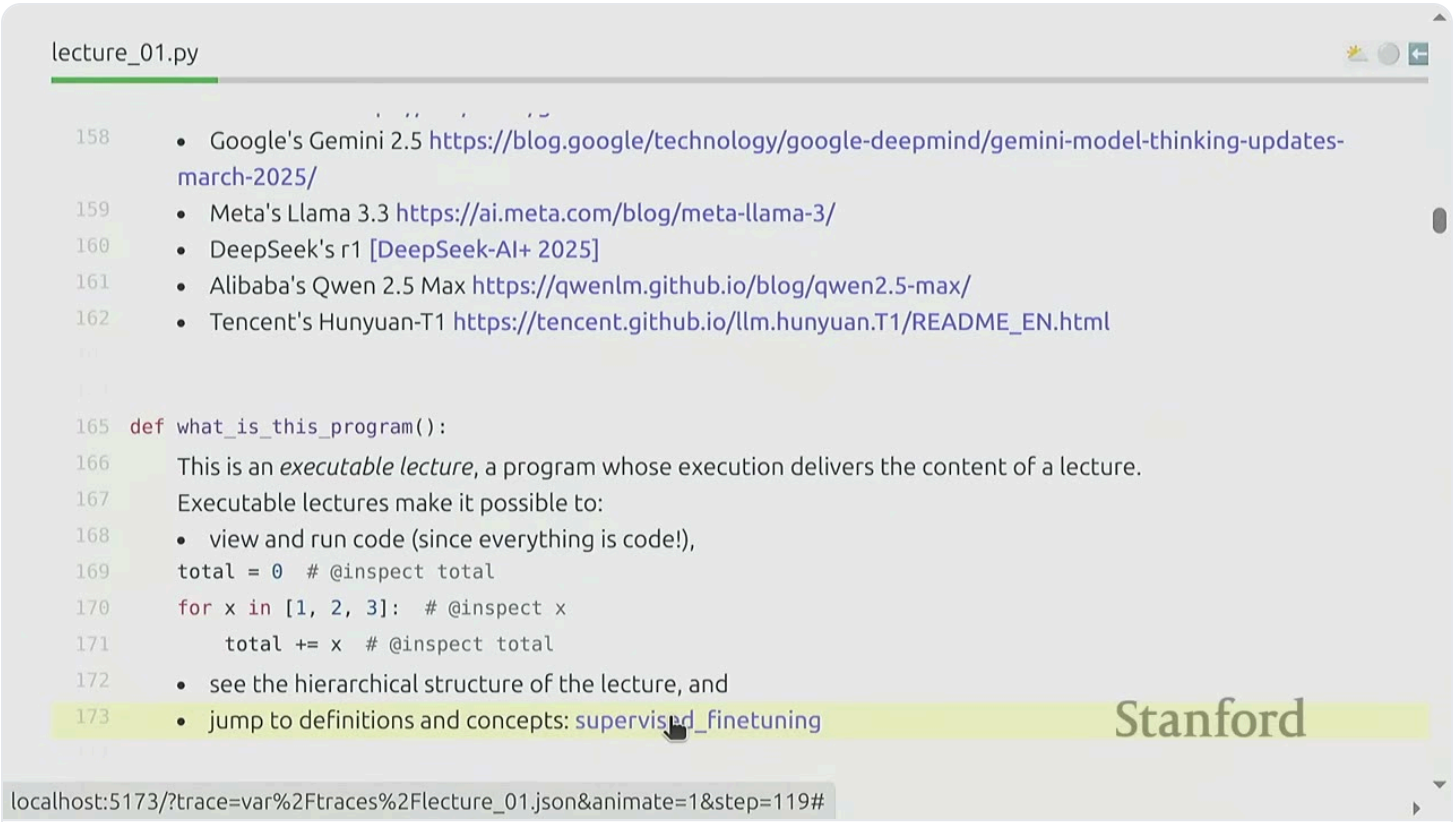
18:22 Text defining an "executable lecture" and its benefits, including viewing and running code.

The lecture itself is an executable program, allowing for: - Viewing and running code (since everything is code). - Stepping through code and inspecting variables. - Seeing the hierarchical structure of the lecture. - Jumping to definitions and concepts.

```
def what_is_this_program():
    """
    This is an executable lecture, a program whose execution delivers the content of a lecture.
    Executable lectures make it possible to:
    - View and run code (since everything is code!),
        total = 0
        for x in [1, 2, 3]: # @inspect x
            total += x # @inspect total
    - see the hierarchical structure of the lecture, and
    - jump to definitions and concepts: supervised_finetuning
    """
    pass

# All information online: https://stanford-cs336.github.io/spring2025/
```

[19:12] Course Logistics



19:01 A code snippet demonstrating an executable lecture with inspect annotations.

- **Website**: https://stanford-cs336.github.io/spring2025/
- **Units**: 5-unit class.
    - **Warning**: A previous course evaluation noted: "The entire assignment was approximately the same amount of work as all 5 assignments from CS 224n plus the final project. And that's just the first homework assignment."

[20:59] Why You Should Take This Course

- You have an obsessive need to understand how things work.
- You want to build up your research engineering muscles.

[21:32] Why You Should Not Take This Course

- You actually want to get research done this quarter (talk to your advisor).
- You are interested in learning about the hottest new techniques in AI (e.g., multimodality, RAG, etc.). (You should take a seminar class for that.)
- You just want to get good results on your own application domain. (You should just prompt or fine-tune an existing model.)

[22:45] How You Can Follow Along at Home

- All lecture materials and assignments will be posted online, so feel free to follow on your own.
- Lectures are recorded via GCOE and will be made available on YouTube (with some lag).
- We plan to offer this class again next year.

[23:21] Assignments

- **5 assignments**: Covering basics, systems, scaling laws, data, alignment.
- **No scaffolding code**: Students are given a blank file to implement components.
- **Unit tests and adapter interfaces**: Provided to help check correctness.
- **Local development**: Implement locally to test for correctness, then run on cluster for benchmarking (accuracy and speed).
- **Leaderboard**: For some assignments (e.g., Assignment 3), minimize perplexity given training budget.
- **AI tools (e.g., Copilot, Cursor)**: Can take away from learning, so use at your own risk.

[25:52] Cluster

- Thanks to Together AI for providing a compute cluster (H100s).
- Please read the guide on how to use the cluster.
- Start your assignments early, as the cluster will fill up close to the deadline.

[26:47] It's All About Efficiency

- **Resources**: Data + hardware (compute, memory, communication bandwidth).
- **Goal**: How do you train the best model given a fixed set of resources?
- **Example**: Given a Common Crawl dump and 32 H100s for 2 weeks, what should you do?

[27:14] Design Decisions

The course is organized into five units, each focusing on a set of design decisions: 1. **Basics**: Tokenization, architecture, loss function, optimizer, learning rate. 2. **Systems**: Kernels, parallelism, quantization, activation checkpointing, CPU offloading, inference. 3. **Scaling Laws**: Scaling sequence, model complexity, loss metric, parametric form. 4. **Data**: Evaluation, curation,

transformation, filtering, deduplication, mixing. 5. **Alignment**: Supervised fine-tuning, reinforcement learning, preference data, verifiers, synthetic data.

[27:44] Overview of the Course Units

[27:44] Basics

- **Goal**: Get a basic version of the full pipeline working.
- **Components**: Tokenization, model architecture, training.

[27:55] Tokenization

- **Definition**: Tokenizers convert between strings and sequences of integers (tokens).
- **Intuition**: Break up strings into popular segments.
- **Process**: `encode` (string to indices), `decode` (indices to string).
- **Course Focus**: Byte-Pair Encoding (BPE) tokenizer (Sennrich+ 2015).
- **Tokenizer-free approaches**: Promising, but not yet scaled to the frontier.

[29:06] Architecture

- **Starting point**: Original Transformer (Vaswani+ 2017).
  - Diagram shows input embeddings, transformer blocks (multi-head attention + position-wise feed-forward), normalization, add & dropout layers, and output probabilities.
- **Variants**:
  - **Activation functions**: ReLU, SwiGLU (Shazeer 2020).
  - **Positional encodings**: Sinusoidal, RoPE (Su+ 2021).
  - **Normalization**: LayerNorm, RMSNorm (Ba+ 2016), ZNnorm (Zhang+ 2019).
  - **Placement of normalization**: Pre-norm versus post-norm (Xiong+ 2020).
  - **MLP**: Dense, mixture of experts (Shazeer+ 2017).
  - **Attention**: Full, sliding window, linear (Jiang+ 2023), GQA (group-query attention), MHA (multi-head attention), LLA (latent-latent attention) (Ainslie+ 2023, DeepSeek-AI+ 2024).
  - **State-space models**: Hyena (Poli+ 2023) - alternatives to Transformers that don't use attention.

[31:35] Training

- **Optimizer**: AdamW, Muon, SOAP (Kingma+ 2014, Loshchilov+ 2017, Keller 2024, Vyas+ 2024).
- **Learning rate schedule**: (e.g., cosine, WSD) (Loshchilov+ 2016, Hu+ 2024).
- **Batch size**: (e.g., critical batch size) (McCandlish 2018).
- **Regularization**: (e.g., dropout, weight decay).
- **Hyperparameters**: (number of heads, hidden dimension): grid search.
- **Key Insight**: Details matter. Proper tuning can yield orders of magnitude difference in performance.

[32:25] Assignment 1

- Implement BPE tokenizer.
- Implement Transformer, cross-entropy loss, AdamW optimizer, training loop.
- Train on TinyStories and OpenWebText.
- **Leaderboard**: Minimize OpenWebText perplexity given 90 minutes on an H100 (last year's leaderboard: 3.12).

[33:52] Systems

- **Goal**: Squeeze the most out of the hardware.
- **Components**: Kernels, parallelism, inference.

[34:12] Kernels

- **What a GPU (A100) looks like**: A GPU is a huge array of Streaming Multiprocessors (SMs) that perform floating-point operations. Memory (DRAM) is off-chip, while L1/L2 caches are on-chip.
- **Analogy**: DRAM (memory) is a warehouse, SRAM (compute) is a factory.
- **Bottleneck**: Bandwidth cost (data movement).
- **Trick**: Organize computation to maximize utilization of GPUs by minimizing data movement.
- **Techniques**: Fusion, tiling.
- **Course Focus**: Write kernels in Triton (OpenAI's framework for GPU programming).

[36:09] Parallelism

- **Challenge**: Data movement between GPUs is even slower than within a single GPU.
- **Principle**: Minimize data movement.
- **Techniques**:

    - Use collective operations (e.g., gather, reduce, all-reduce).
    - Shard (parameters, activations, gradients, optimizer states) across GPUs.
    - Split computation (data, tensor, pipeline) parallelism.

[37:11] Inference

- **Goal**: Generate tokens given a prompt (needed to actually use models!).
- **Importance**: Inference is also needed for reinforcement learning, test-time compute, and evaluation.
- **Cost**: Globally, inference compute (every use) exceeds training compute (one-time cost).
- **Two phases**: Prefill and decode.
    - **Prefill**: Tokens are given, can process all at once (compute-bound). Similar to training.
    - **Decode**: Need to generate one token at a time (memory-bound). Autoregressive generation is memory-bound due to frequent data movement.
- **Methods to speed up decoding**:
    - Use cheaper models (via model pruning, quantization, distillation).
    - Speculative decoding: Use a cheaper "draft" model to generate multiple tokens, then use the full model to score in parallel (exact decoding!).
    - Systems optimizations: KV caching, batching.

[39:50] Assignment 2

- Implement a fused RMSNorm kernel in Triton.
- Implement distributed data parallel training.
- Implement optimizer state sharding.
- Benchmark and profile the implementations.
- **Key Habit**: Always benchmark and profile your implementations to understand performance and bottlenecks.

[40:51] Scaling Laws

- **Goal**: Do experiments at small scale, predict hyperparameters/loss at large scale.
- **Question**: Given a FLOPs budget (C), use a bigger model (N) or train on more tokens (D)?
- **Compute-optimal scaling laws**: (Kaplan+ 2020, Hoffmann+ 2022).
    - Basic idea: For a given compute budget, there's an optimal balance between model size and training data.

- Rule of thumb: $D \approx 20 \times N$. (e.g., a 1.4B parameter model should be trained on 28B tokens).
- **Limitation**: This rule doesn't take into account inference costs.

[43:00] Assignment 3

- Define a training API (hyperparameters -> loss) based on previous runs.
- Submit "training jobs" (under a FLOPs budget) and gather data points.
- Fit a scaling law to the data points.
- Submit predictions for scaled up hyperparameters.
- **Leaderboard**: Minimize loss given FLOPs budget.
- **Key Skill**: Prioritizing experiments and resource allocation, similar to frontier labs.

[44:50] Data

- **Question**: What capabilities do we want the model to have?
- **Answer**: The model's capabilities are largely determined by the data it's trained on (e.g., multilingual data -> multilingual capabilities, code data -> code capabilities).

[45:32] Composition of the Pile by Category

- Data sources are diverse: web crawls (Common Crawl), books, arXiv papers, GitHub code, Wikipedia, StackExchange, etc.

[45:56] Evaluation

- **Perplexity**: Textbook evaluation for language models.
- **Standardized testing**: (e.g., MMLU, HellaSwag, GSM8K).
- **Instruction following**: (e.g., AlpacaEval, IFEval, WildBench).
- **Scaling test-time compute**: Chain-of-thought, ensembling.
- **LM-as-a-judge**: Evaluate generative tasks.
- **Full system**: RAG, agents.

[46:45] Data Curation

- **Key Insight**: Data does not just fall from the sky. It needs to be actively acquired and processed.
- **Sources**: Webpages crawled from the Internet, books, arXiv papers, GitHub code, etc.
- **Legal Questions**: Appeal to fair use to train on copyright data (Henderson+ 2023).

- **Commercial Data**: Frontier models often buy data (e.g., Google with Reddit data) as publicly available data is limited for state-of-the-art performance.
- **Formats**: HTML, PDF, directories (not text!).

[47:26] Look at Web Data

- Example of raw Common Crawl data: shows a mix of HTML, foreign languages, and spammy content.
- **Observation**: A lot of web data is "trash" and requires significant processing.

[48:29] Data Processing

- **Transformation**: Convert HTML/PDF to text (preserve content, some structure, rewriting).
- **Filtering**: Keep high quality data, remove harmful content (via classifiers).
- **Deduplication**: Save compute, avoid memorization (use Bloom filters or MinHash).

[50:00] Assignment 4

- Convert Common Crawl HTML to text.
- Train classifiers to filter for quality and harmful content.
- Deduplication using MinHash.
- **Leaderboard**: Minimize perplexity given token budget.

[50:21] Alignment

- **Problem**: A base model is raw potential, very good at completing the next token, but needs to be made useful.
- **Alignment**: The process of making the model actually useful.
- **Goals of alignment**:
    1. Get the language model to follow instructions.
    2. Tune the style (format, length, tone, etc.).
    3. Incorporate safety (e.g., refusals to answer harmful questions).

[51:44] Two Phases

1. **Supervised Fine-tuning (SFT)**:
    - **Instruction data**: Prompt, response pairs (e.g., ChatExample).

- **Intuition**: Base model already has the skills, just need few examples to surface them (Zhou+ 2023).
- **Learning**: Fine-tune model to maximize $P(response|prompt)$.
- SFT is relatively simple and effective for initial alignment.

2. **Learning from Feedback**:

   - **Goal**: Make it better without expensive annotation.
   - **Preference data**: Generate multiple responses using model (e.g., A, B) to a given prompt. User provides preferences (e.g., A < B or A > B).
   - **Verifiers**:
     - **Formal verifiers**: (e.g., for code, math).
     - **Learned verifiers**: Train an LM as-a-judge (RLHF).
   - **Algorithms**:
     - **Proximal Policy Optimization (PPO)**: From reinforcement learning (Schulman+ 2017, Ouyang+ 2022).
     - **Direct Policy Optimization (DPO)**: For preference data, simpler (Rafailov+ 2023).
     - **Group Relative Preference Optimization (GRPO)**: Remove value function (Shao+ 2024).

[55:04] Assignment 5

- Implement supervised fine-tuning.
- Implement Direct Preference Optimization (DPO).
- Implement Group Relative Preference Optimization (GRPO).
- Evaluate.

[55:59] Efficiency Drives Design Decisions (Recap)

- Today, we are compute-constrained, so design decisions reflect squeezing the most out of given hardware.
- **Data processing**: Avoid wasting precious compute updating on bad/irrelevant data.
- **Tokenization**: Working with raw bytes is elegant, but compute-inefficient with today's model architectures.
- **Model architecture**: Many changes motivated by reducing memory or FLOPs (e.g., sharing KV caches, sliding window attention).
- **Training**: We can get away with a single epoch!

- **Scaling laws**: Use less compute on smaller models to do hyperparameter tuning.
- **Alignment**: If tune model more to desired use cases, require smaller base models.

[58:09] Tomorrow: Data-Constrained Regime

- Increasingly, frontier labs are becoming data-constrained.
- This will likely shift design decisions.
- **Example**: Training for one epoch doesn't make sense if you have more compute; you'd train for more epochs or use smarter techniques.
- **Example**: Different architectures might emerge, as the Transformer was motivated by compute efficiency.
- The design decisions will reflect the regime we're in.

[59:01] Tokenization (Deep Dive)

This unit is inspired by Andrej Karpathy's video on tokenization.

[59:50] Intro to Tokenization

- **Raw text**: Generally represented as Unicode strings.
- **Indices**: Language model places a probability distribution over sequences of tokens (usually represented by integer indices).
- **Tokenizer**: A class that implements `encode` (strings to tokens) and `decode` (tokens back into strings) methods.
- **Vocabulary size**: Number of possible tokens (integers).

[1:01:25] Tokenization Examples

- **Tiktoken (GPT-4o)**: An interactive site to visualize tokenization.
  - **Observations**:
    - A word and its preceding space are part of the same token (e.g., " world").
    - A word at the beginning and in the middle are represented differently (e.g., "hello hello").
    - Numbers are tokenized into every few digits.
  - The spacing before a token is intentional, as part of the BPE process.
  - "hello" and " hello" are different tokens, which can cause problems.

[1:05:17] Character-Based Tokenization

- **Concept**: A Unicode string is a sequence of Unicode characters. Each character is converted into a code point (integer) via `ord()`.
- **Example**: `ord('a')` is 97, `ord('🌍')` is 127757.
- **Problem 1**: Very large vocabulary (up to 150K Unicode characters).
- **Problem 2**: Many characters are quite rare, leading to inefficient use of the vocabulary.
- **Compression ratio**: Number of bytes / number of tokens. For character-based tokenization, the compression ratio is ~1.5 (for "Hello, 🌍! 안녕하세요!").

[1:07:15] Byte-Based Tokenization

- **Concept**: Unicode strings can be represented as a sequence of bytes (integers between 0 and 255). The most common Unicode encoding is UTF-8.
- **Example**: `bytes('a', encoding="utf-8")` is `b'a'`. Some characters take multiple bytes (e.g., `bytes('🌍', encoding="utf-8")` is `b'\xf0\x9f\x8c\x8e'`).
- **Problem**: Long sequences. Compression ratio is 1 (1 byte per token). This is terrible for Transformers due to quadratic attention complexity.

[1:09:16] Word-Based Tokenization

- **Concept**: Another approach (closer to what was done classically in NLP) is to split strings into words.
- **Example**: Using regex to find all alphanumeric characters together (words).
- **Problem 1**: The number of words is huge (for Unicode characters).
- **Problem 2**: Many words are rare and the model won't learn much about them.
- **Problem 3**: This doesn't obviously provide a fixed vocabulary size.
- **Problem 4**: New words not seen during training get a special UNK token, which is ugly and can mess up perplexity calculations.

[1:11:04] Byte Pair Encoding (BPE)

- **History**: Introduced by Philip Gage in 1994 for data compression. Adapted to NLP for neural machine translation by Sennrich+ 2015. Previously, papers had been using word-based tokenization. BPE was then used by GPT-2 (Radford+ 2019).
- **Basic Idea**: Train the tokenizer on raw text to automatically determine the vocabulary.

- **Intuition**: Common sequences of characters are represented by a single token. Rare sequences are represented by many tokens.
- **GPT-2 detail**: Uses word-based tokenization to break up the text into initial segments and then runs the original BPE algorithm on each segment. (This is what you'll do in this class).

[1:12:44] Training the Tokenizer

- **Algorithm**:
  1. Start with the list of bytes of a string.
  2. Count the number of occurrences of each pair of adjacent tokens.
  3. Find the most common pair.
  4. Merge that pair: Create a new token (e.g., 256) to represent the merged pair.
  5. Replace all occurrences of the pair with the new token in the sequence.
  6. Repeat steps 2-5 for a specified number of merges (`num_merges`).
- **Example**: "the cat in the hat"
  - Initial bytes: `[116, 104, 101, 32, 99, 97, 116, 32, 105, 110, 32, 104, 97, 116]`
  - After 1 merge (116, 104 -> 256): `[256, 101, 32, 99, 97, 256, 32, 105, 110, 32, 256]`
  - After 2 merges (256, 32 -> 257): `[257, 101, 99, 97, 257, 105, 110, 257]`
  - After 3 merges (257, 101 -> 258): `[258, 99, 97, 257, 105, 110, 257]`
- **Observation**: The sequence shrinks with each merge, increasing compression ratio.

[1:16:11] Using the New Tokenizer

- **Encoding**: Convert string to initial bytes, then replay the merges in the order they occurred.
- **Decoding**: Convert tokens back to bytes, then bytes to string.
- **Implementation**: In Assignment 1, you'll implement this from scratch. Your goal is to make the implementation fast.

[1:17:47] Summary of Tokenization

- **Tokenizer**: Maps strings to tokens (indices).
- **Character-based, byte-based, word-based tokenization**: Highly suboptimal.
- **BPE**: An effective heuristic that looks at corpus statistics.
- **Future**: BPE is a necessary evil, maybe one day we'll just do it from bytes.

Practical Takeaways

- **Build to Understand**: Deep understanding of ML systems comes from implementing them from scratch.
- **Efficiency is Key**: Optimize algorithms and resource usage, especially at scale, to maximize accuracy per resource.
- **Scaling Laws Inform Design**: Use small-scale experiments and scaling laws to predict large-scale behavior and make informed design decisions.
- **Data Quality and Curation**: Data is not "free" or "clean"; active acquisition, processing, filtering, and deduplication are critical.
- **Alignment is Crucial**: Base models need to be aligned to follow instructions, tune style, and ensure safety.
- **Tokenization Trade-offs**: Different tokenization methods (character, byte, word, BPE) involve trade-offs between vocabulary size, sequence length, and efficiency.

Open Questions / Things to Remember

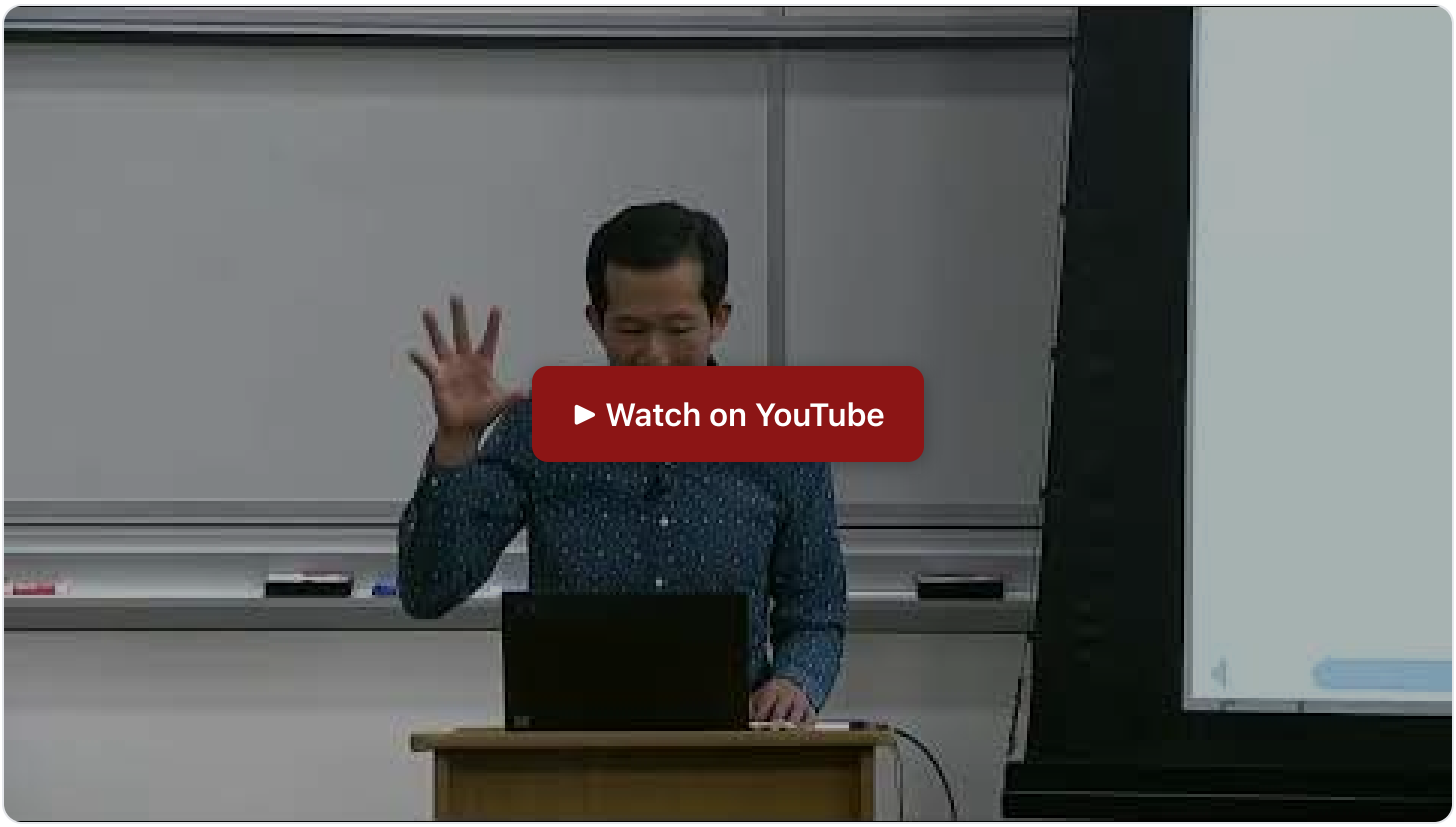
- How do emergent behaviors at scale impact research conducted at smaller scales?
- What are the ethical and legal implications of using copyrighted data for training LMs?
- How will the shift to data-constrained regimes impact future model architectures and training methodologies?
- What are the most promising alternatives to the Transformer architecture (e.g., state-space models) and how do they address current limitations?
- How can we develop more robust and transferable intuitions about model behavior across different scales?
- What are the practical limits of "divine benevolence" in explaining model successes, and how can we move towards more principled understanding?

Lec 02 →

Lecture 02: PyTorch, Resource Accounting

Stanford CS336 · Spring 2025 · 1:19:22 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.

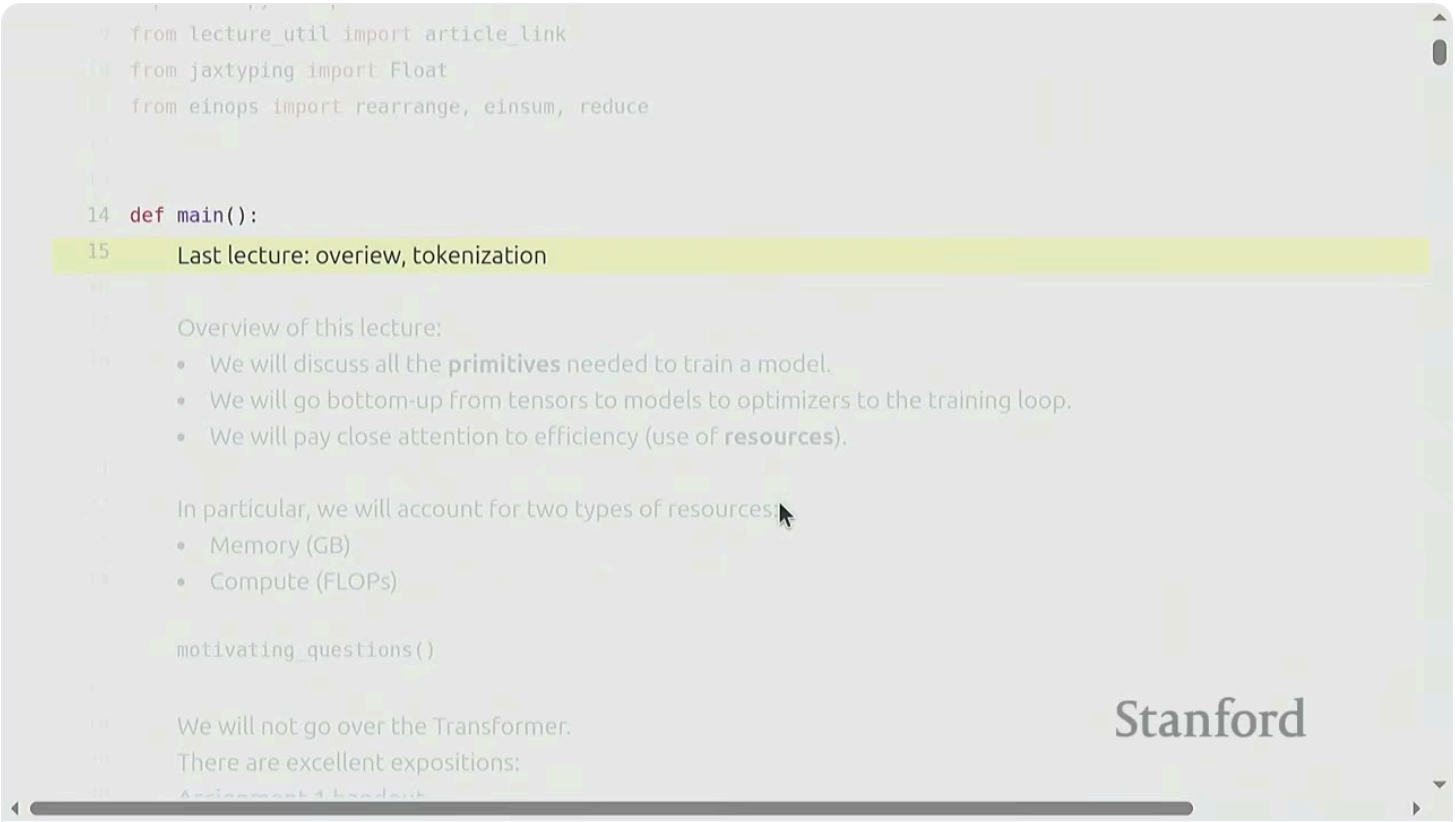


TL;DR - This lecture focuses on building language models from scratch using PyTorch, emphasizing efficiency and resource accounting (memory and compute). - Key topics include PyTorch tensors, floating-point precision types (float32, float16, bfloat16, fp8), memory accounting, and compute accounting (FLOPs). - The lecture introduces `einops` for robust tensor manipulation and discusses the computational cost of forward and backward passes in neural networks. - It also covers parameter initialization strategies, data loading, optimizers (SGD, Adam, etc.), and best practices like checkpointing and mixed-precision training.

Key concepts - PyTorch tensors as fundamental building blocks. - Floating-point precision and its impact on memory and numerical stability. - Memory accounting: how to calculate memory usage for tensors. - Compute accounting: understanding FLOPs (floating-point operations) and

FLOP/s (FLOPs per second). - `einops` for dimension-aware tensor operations. - Computational cost of forward and backward passes (gradients). - Parameter initialization techniques (e.g., Xavier initialization). - Optimizers (SGD, Adam, AdaGrad, RMSprop). - Checkpointing for fault tolerance. - Mixed-precision training for efficiency.

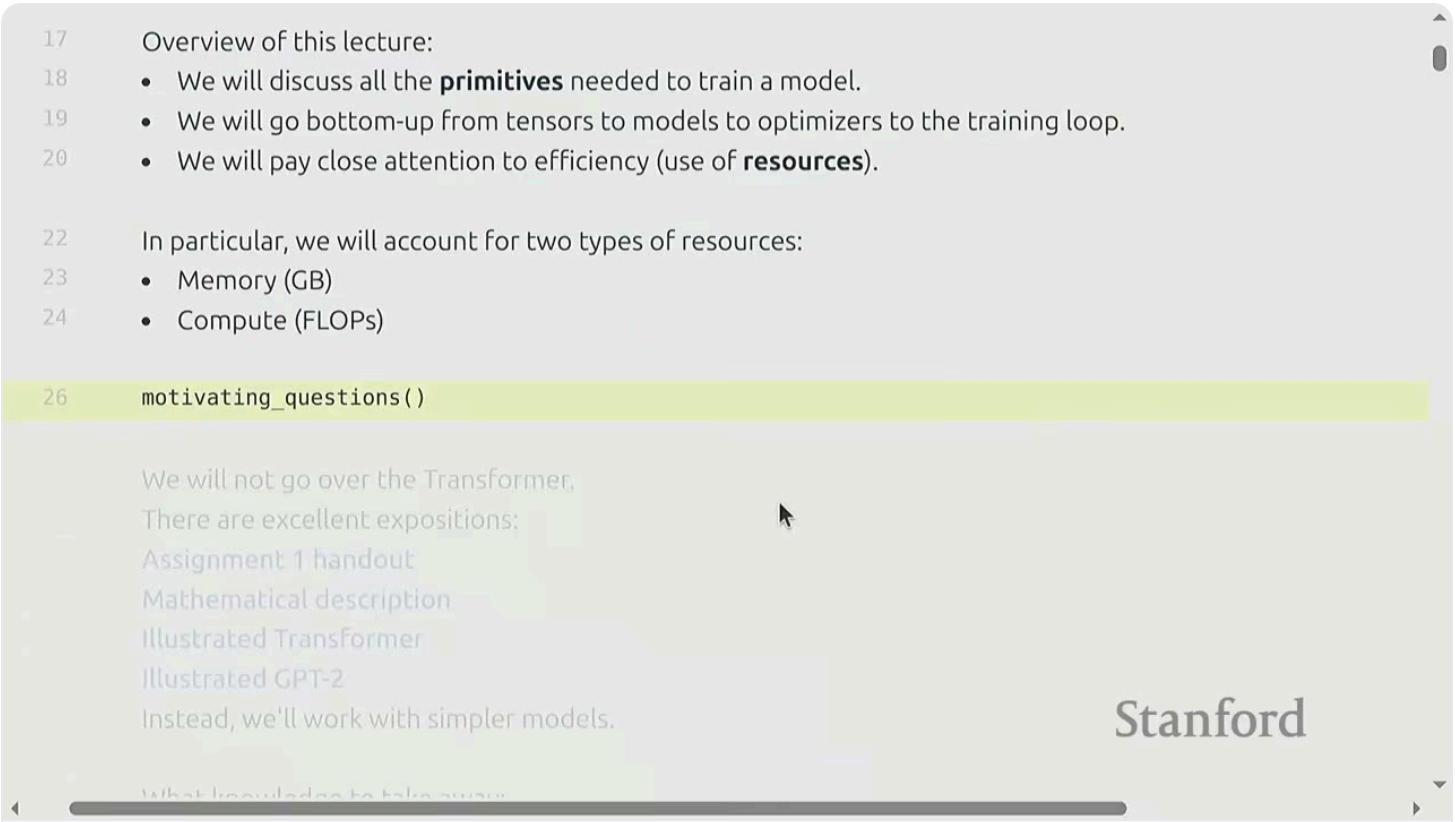
[00:00] Introduction and Lecture Overview



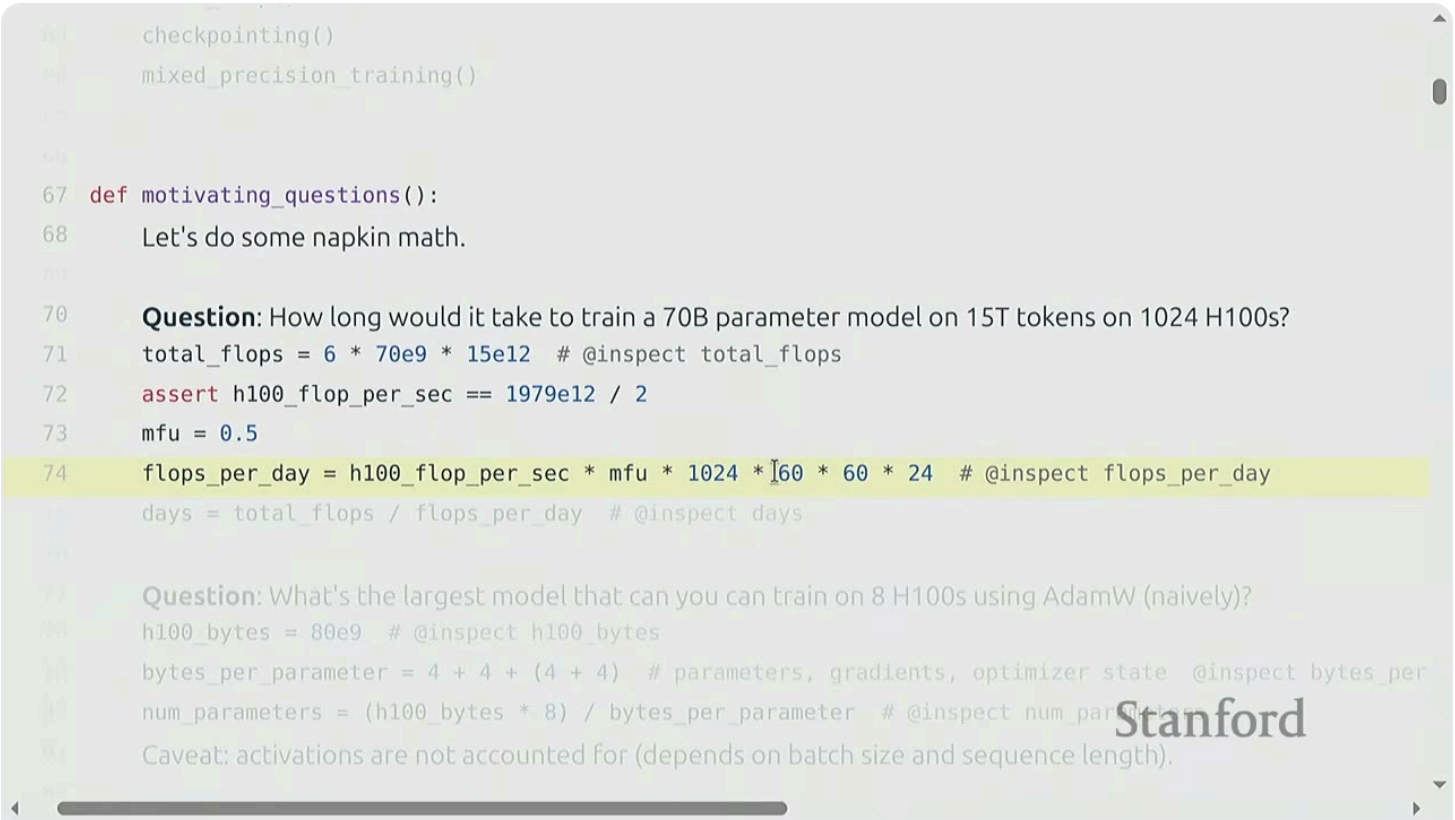
0:16 The slide shows the title "Last lecture: overview, tokenization" highlighted in yellow.

The previous lecture covered an overview of language models, building them from scratch, and tokenization (first half of Assignment 1). This lecture will focus on building a model, discussing PyTorch primitives, and emphasizing efficiency and resource usage (memory and compute).

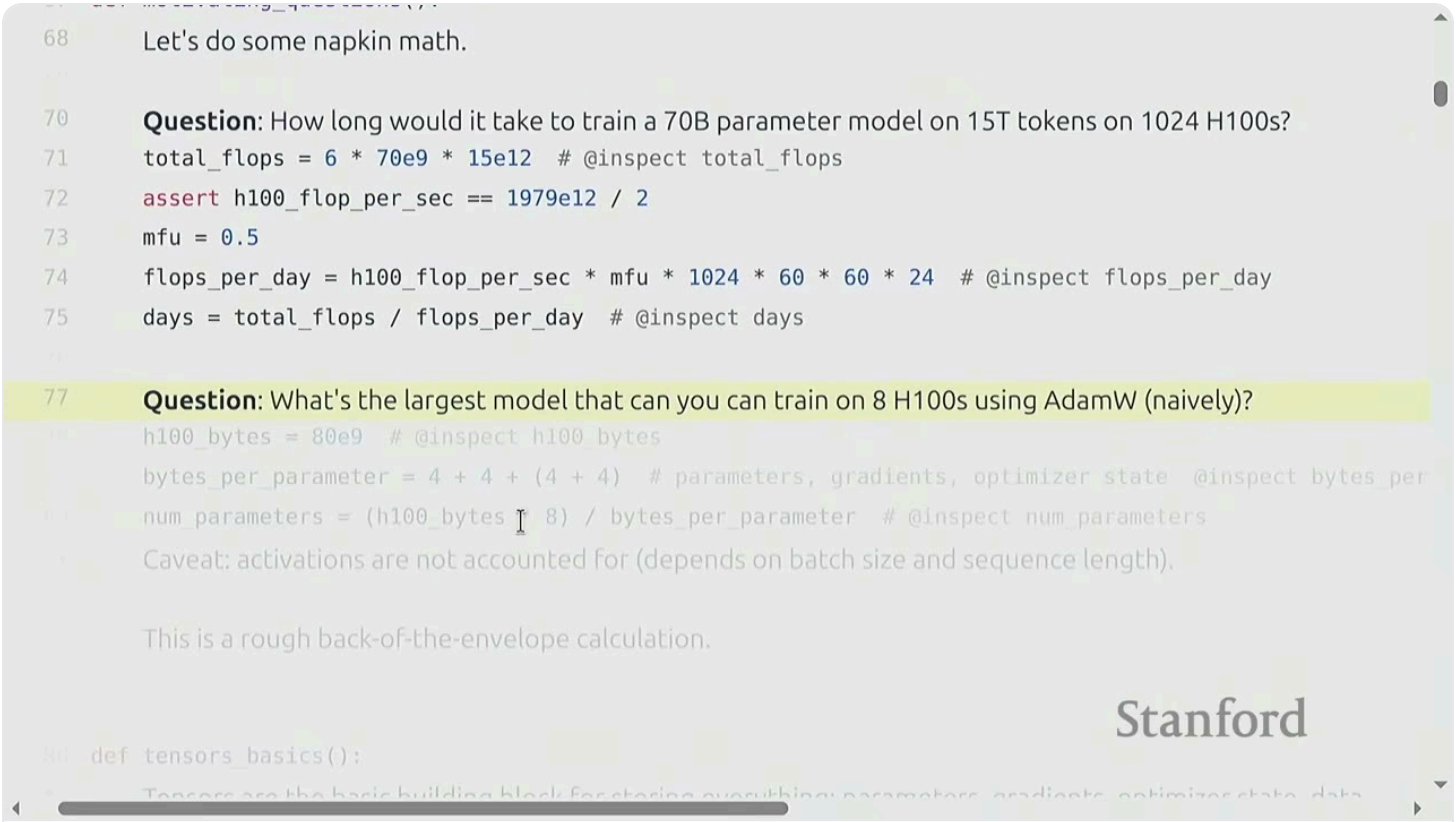
[00:43] Motivating Questions: Napkin Math for LLMs



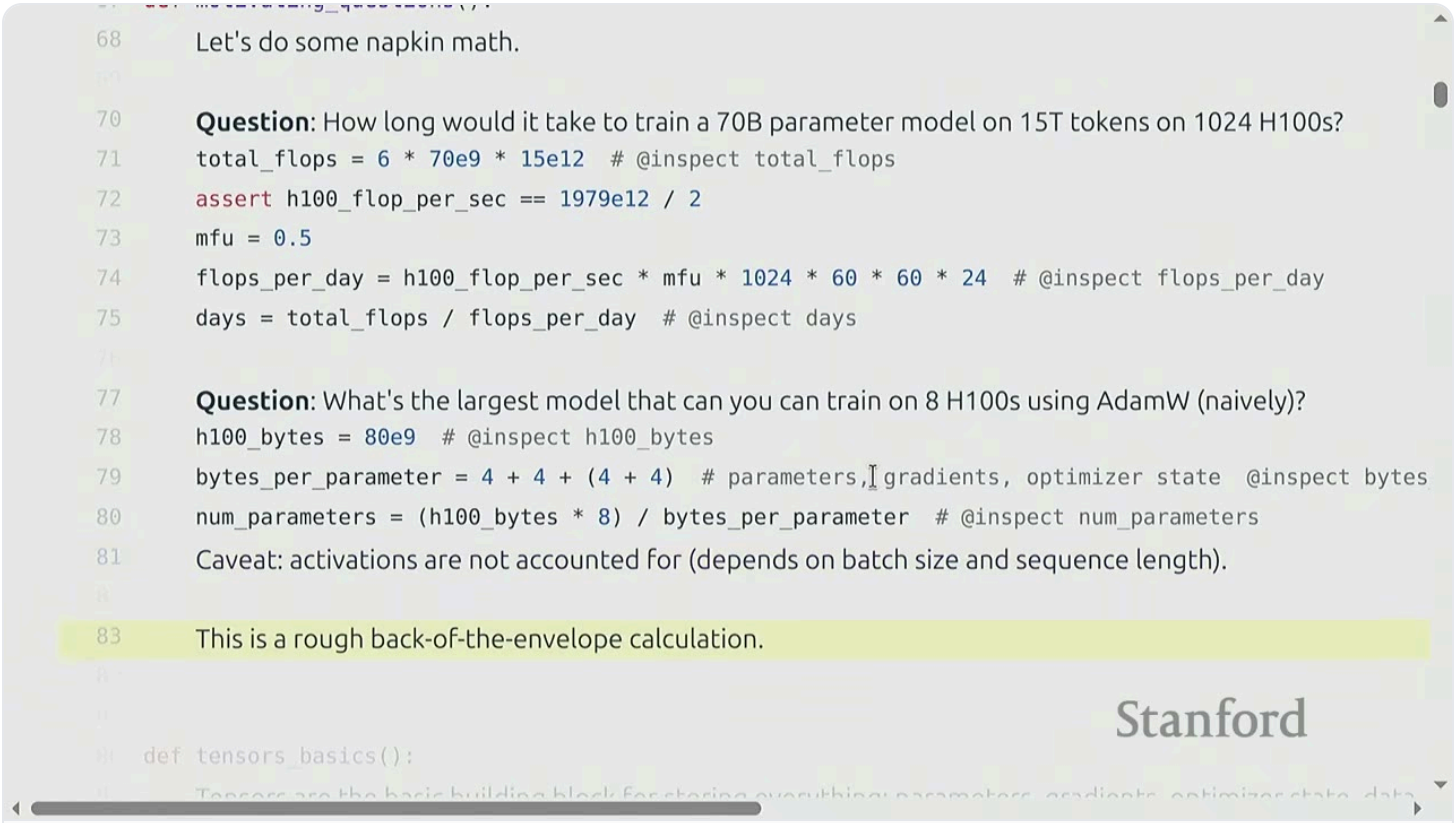
0:48 The slide shows an overview of the lecture, focusing on primitives, tensors, and efficiency.



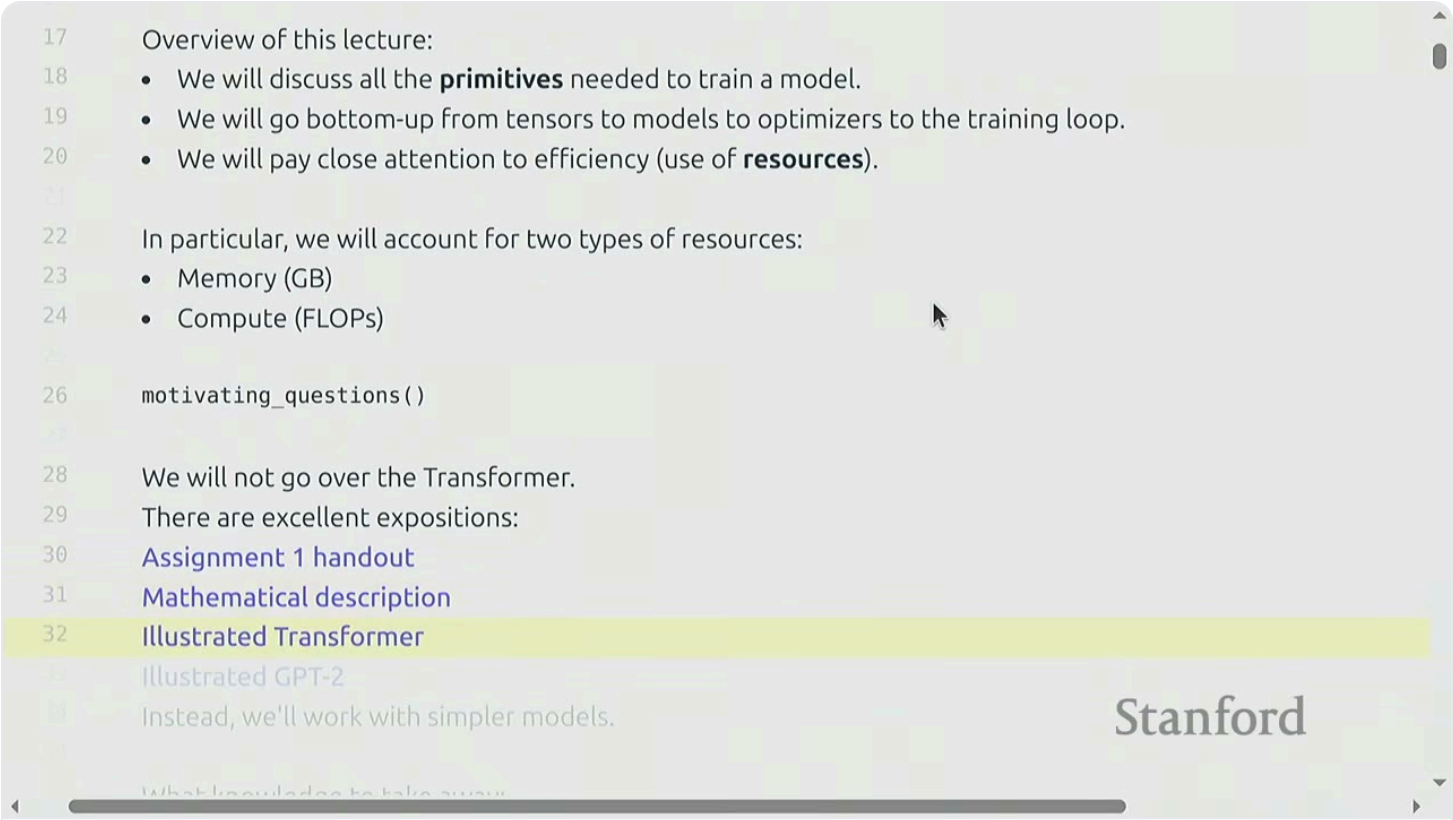
1:54 The slide shows Python code for calculating FLOPs per day for a 70B parameter model.



2:30 The slide shows a question about the largest model trainable on 8 H100s using AdamW.



3:24 The slide shows a highlighted note: "This is a rough back-of-the-envelope calculation."



4:19 The slide shows "Illustrated Transformer" highlighted, indicating a topic not covered.

The lecture begins with "napkin math" questions to motivate the importance of resource accounting.

Question 1: How long would it it take to train a 70 billion parameter dense transformer model on 15 trillion tokens on 1024 H100s?

Reasoning: 1. **Total FLOPs needed:** * A common rule of thumb for training transformer models is that the total FLOPs required is $6 \times (\text{number of parameters}) \times (\text{number of tokens})$. * For a 70 billion parameter model and 15 trillion tokens: $6 \times (70 \times 10^9) \times (15 \times 10^{12}) = 6.3 \times 10^{24}$ FLOPs. * The "6" factor will be explained later in the lecture.

2. **FLOPs per day from hardware:**

    - An H100 GPU provides a certain number of FLOPs per second (FLOP/s). Let's assume a specific value (e.g., 1979×10^{12} FLOP/s for H100 with sparsity, 50% without, as seen later).
    - We also consider Model FLOPs Utilization (MFU), which is the ratio of actual FLOP/s to promised FLOP/s. Let's assume MFU = 0.5 (a reasonable value for matrix multiplications).

- FLOPs per day = (H100 FLOP/s) $\times$ MFU $\times$ (number of H100s) $\times$ (seconds per minute) $\times$ (minutes per hour) $\times$ (hours per day)
- FLOPs per day = $(1979 \times 10^{12}) \times 0.5 \times 1024 \times 60 \times 60 \times 24 \approx 4.377 \times 10^{22}$ FLOPs/day.

3. **Total training time:**

- Time (days) = Total FLOPs / FLOPs per day
- Time (days) = $(6.3 \times 10^{24})/(4.377 \times 10^{22}) \approx 143.9$ days.
- So, approximately 144 days (about 4.8 months).

Question 2: What is the largest model you can train on 8 H100s using AdamW (naively)?

Reasoning: 1. **Memory per H100:** An H100 has 80 GB of HBM (High Bandwidth Memory). 2. **Bytes per parameter:** For parameters, gradients, and optimizer state (AdamW), you typically need 16 bytes per parameter (4 bytes for the parameter, 4 for its gradient, and 8 for AdamW's two momentum states). 3. **Total parameters:** * Total HBM = (Number of H100s) $\times$ (Memory per H100) = $8 \times 80 \times 10^9$ bytes. * Number of parameters = (Total HBM) / (Bytes per parameter) * Number of parameters = $(8 \times 80 \times 10^9)/16 = 40 \times 10^9 = 40$ billion parameters. * This calculation is rough because it doesn't account for activations, which depend on batch size and sequence length.

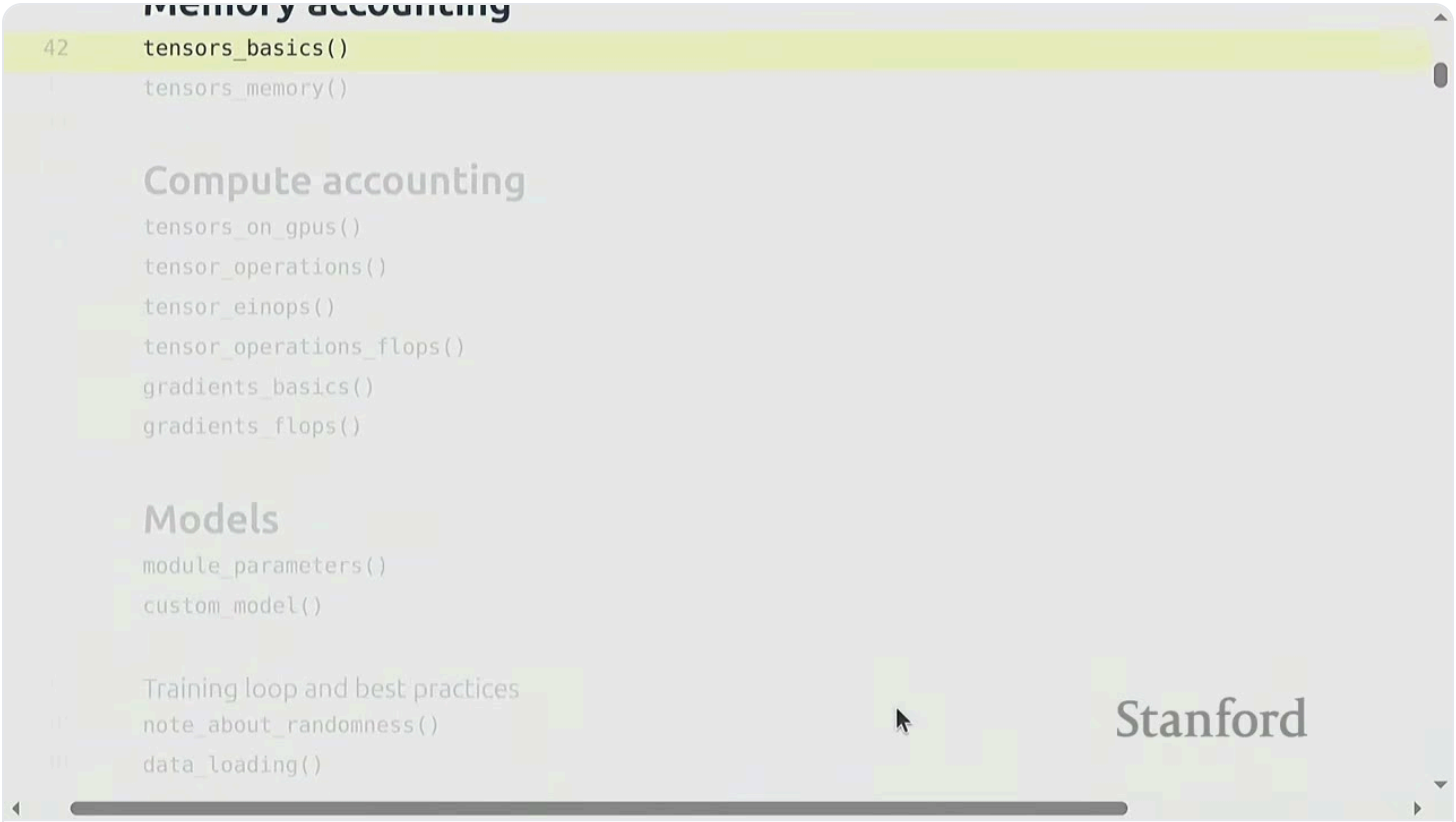
Why efficiency matters: These "back-of-the-envelope" calculations are crucial because large-scale model training directly translates to significant costs (dollars, energy). Understanding resource usage helps optimize these costs.

Note on Transformers: The lecture will not delve deeply into the Transformer architecture itself. There are excellent external resources for learning about Transformers, including Assignment 1 handout, mathematical descriptions, and illustrated guides (e.g., Illustrated Transformer, Illustrated GPT-2). The focus will be on simpler models to understand primitives and resource accounting.

What knowledge to take away: - **Mechanics:** Straightforward PyTorch usage. - **Mindset:** Resource accounting (remember to do it). - **Intuitions:** Broad strokes (not specific to large models).

[05:10] Memory Accounting

[05:20] Tensors Basics



5:17 The slide shows the title "Memory accounting" and a list of related functions.

Tensors are the fundamental building blocks in deep learning for storing everything: parameters, gradients, optimizer state, data, and activations.

```python
def tensors_basics():
    # PyTorch docs on tensors
    # You can create tensors in multiple ways:
    x = torch.tensor([[1., 2, 3], [4, 5, 6]]) # @inspect x
    x = torch.zeros(4, 8) # 4x8 matrix of all zeros @inspect x
    x = torch.ones(4, 8) # 4x8 matrix of all ones @inspect x
    x = torch.randn(4, 8) # 4x8 matrix of iid Normal(0, 1) samples @inspect x

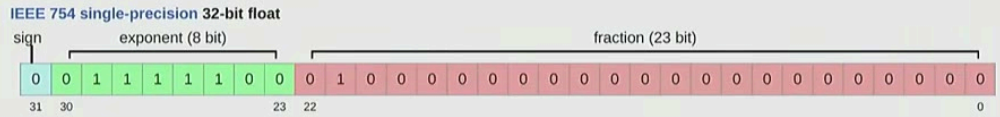
    # Allocate but don't initialize the values:
    x = torch.empty(4, 8) # 4x8 matrix of uninitialized values @inspect x
    # ...because you want to use some custom logic to set the values later
    nn.init.trunc_normal_(x, mean=0, std=1, a=-2, b=2) # @inspect x
```

[06:00] Tensors Memory

```
    nn.init.trunc_normal_(x, mean=0, std=1, a=-2, b=2)  # @inspect x


def tensors_memory():
    Almost everything (parameters, gradients, activations, optimizer states) are stored as floating point numbers.

    float32
    [Wikipedia]
```




```
    The float32 data type (also known as fp32 or single precision) is the default.
    Traditionally, in scientific computing, float32 is the baseline; you could use double precision (float64) in some
    cases.
    In deep learning, you can be a lot sloppier.

    Let's examine memory usage of these tensors.
    Memory is determined by the (i) number of values and (ii) data type of each value.
    x = torch.zeros(4, 8)  # @inspect x
```

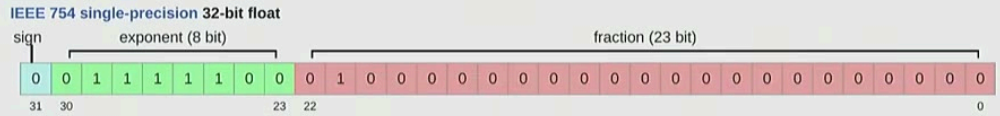
Stanford

6:38 The slide shows a diagram of the float32 data type with its sign, exponent, and fraction.

```
    nn.init.trunc_normal_(x, mean=0, std=1, a=-2, b=2)  # @inspect x


def tensors_memory():
    Almost everything (parameters, gradients, activations, optimizer states) are stored as floating point numbers.

    float32
    [Wikipedia]
```

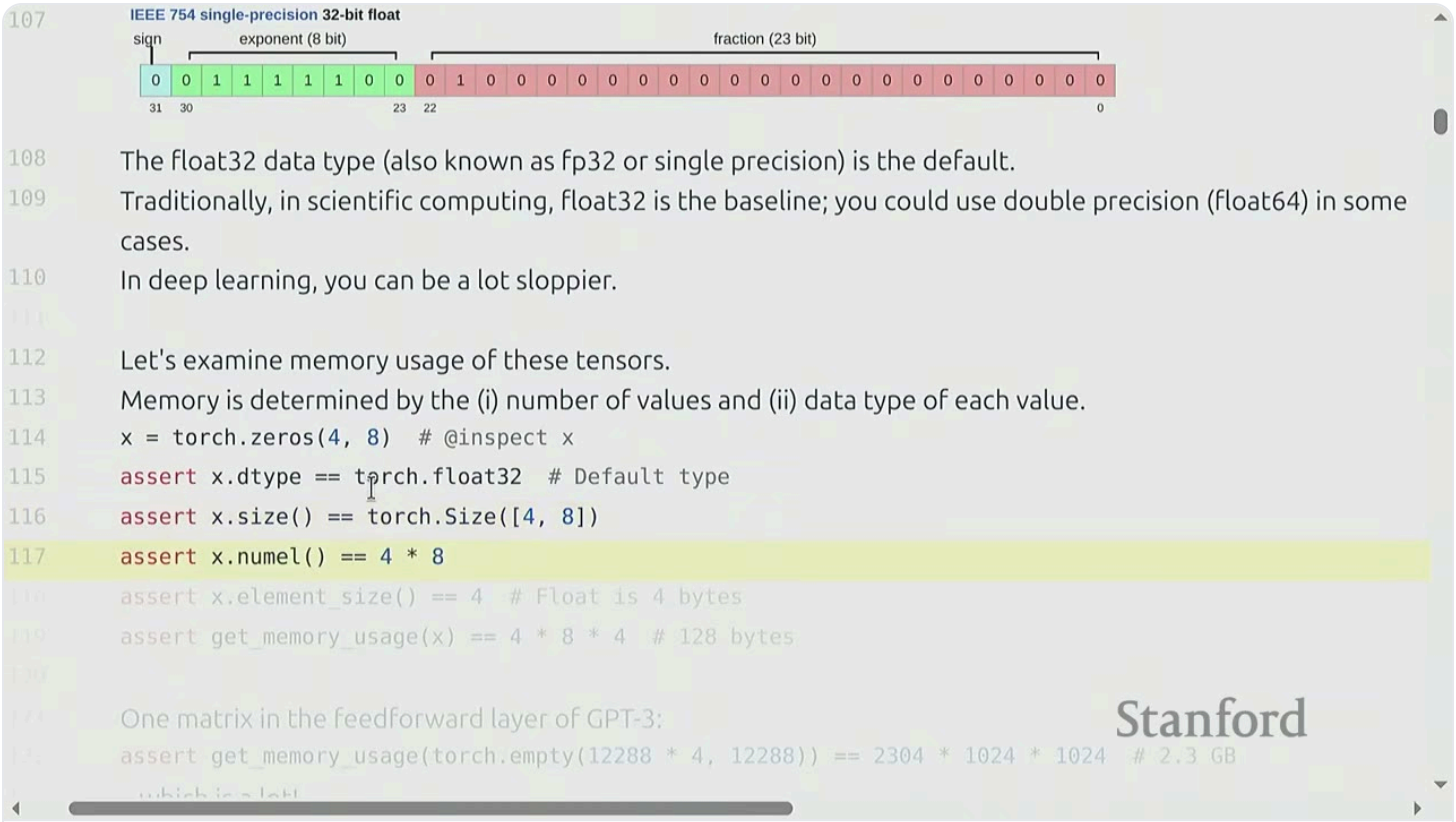



```
    The float32 data type (also known as fp32 or single precision) is the default.
    Traditionally, in scientific computing, float32 is the baseline; you could use double precision (float64) in some
    cases.
    In deep learning, you can be a lot sloppier.

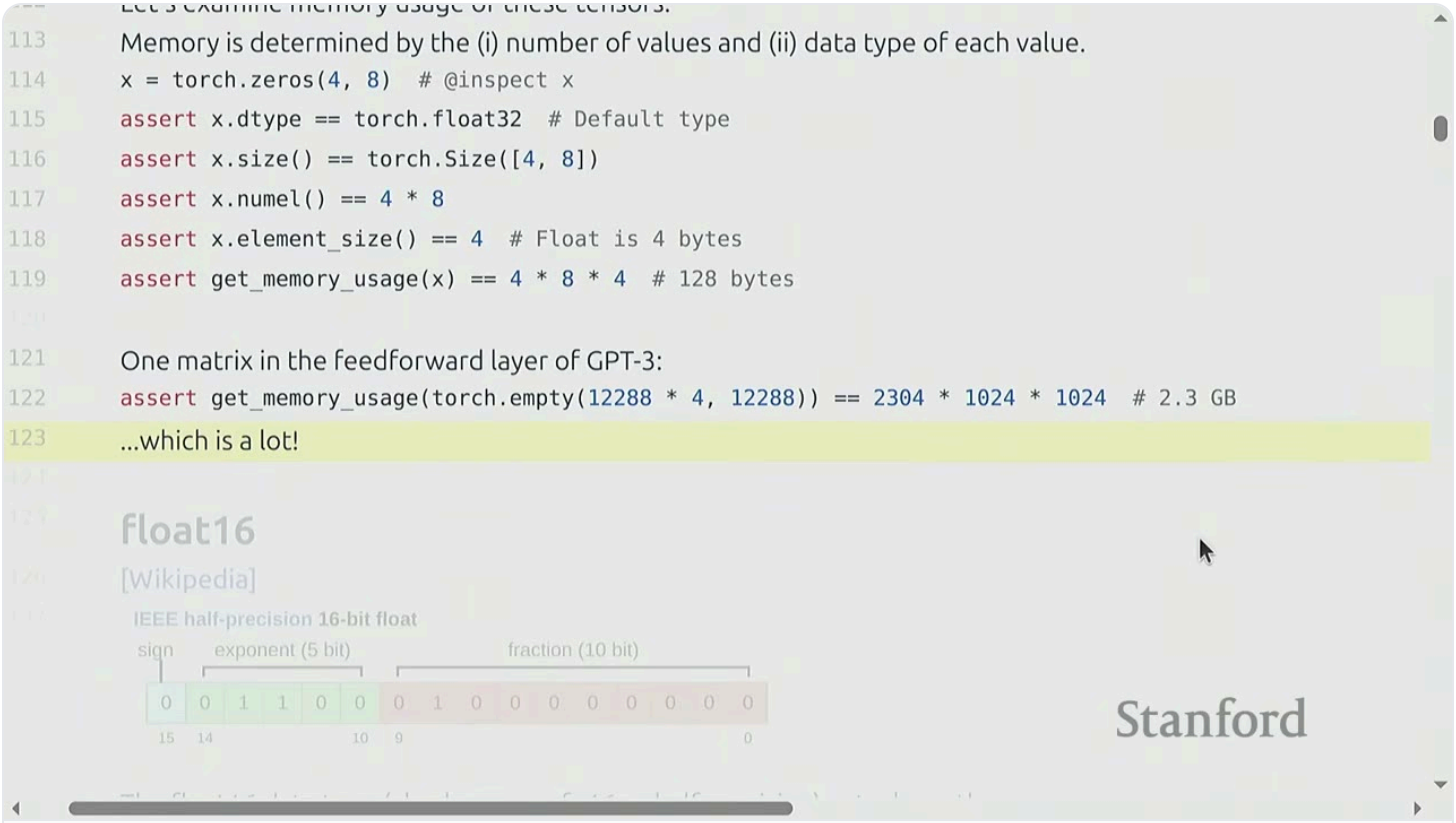
    Let's examine memory usage of these tensors.
    Memory is determined by the (i) number of values and (ii) data type of each value.
    x = torch.zeros(4, 8)  # @inspect x
```

Stanford

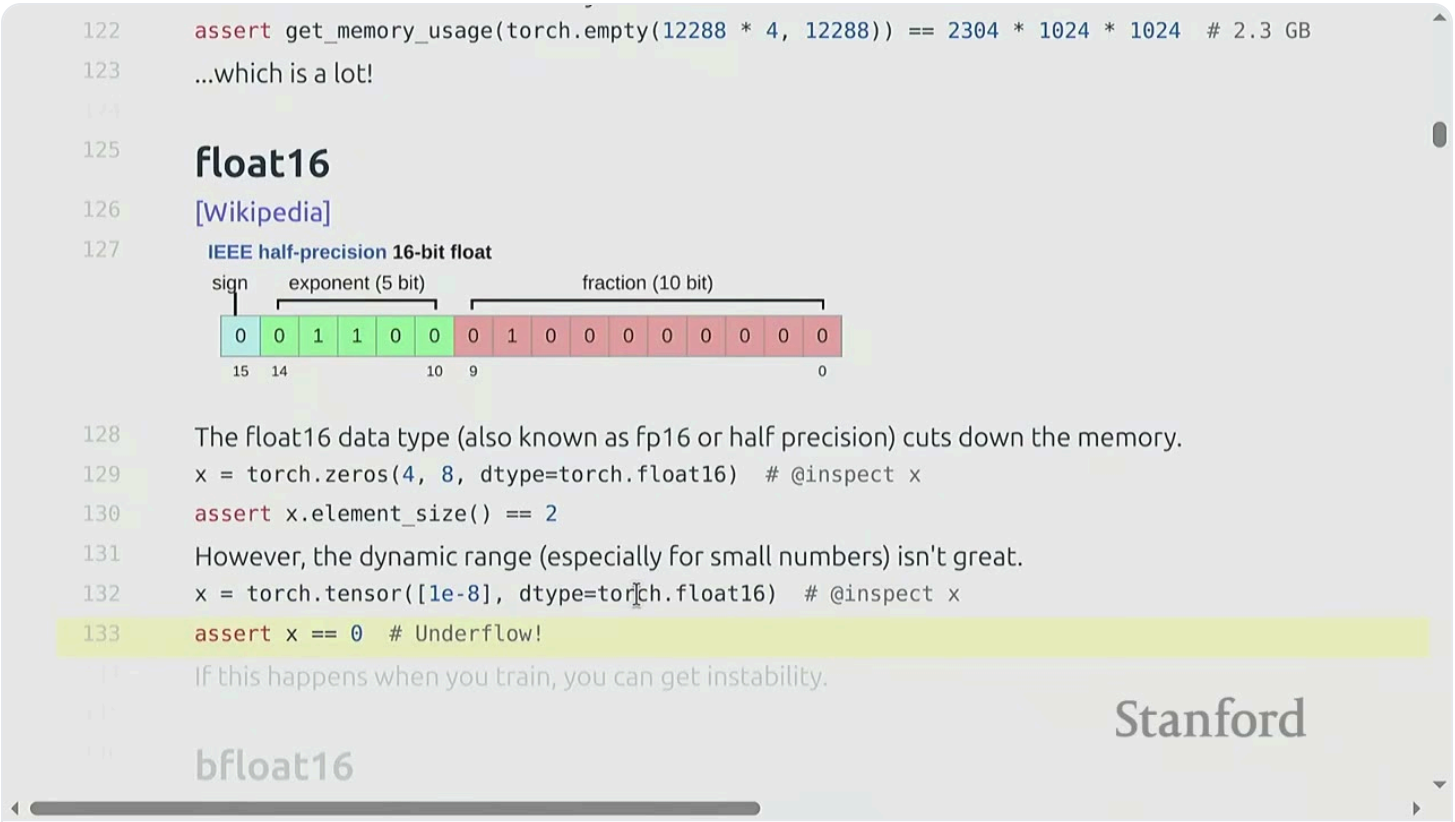
7:14 The slide shows a highlighted statement: "In deep learning, you can be a lot sloppier."



7:40 The slide shows Python code demonstrating memory usage of tensors with float32.



8:28 The slide shows a highlighted statement: "...which is a lot!" regarding memory usage.



9:28 The slide shows a diagram of the float16 data type and code demonstrating its properties.

Almost everything (parameters, gradients, activations, optimizer states) is stored as floating-point numbers.

Float32 (Single-precision) - The default data type in PyTorch is `float32` (also known as FP32 or single precision). - It uses 32 bits: 1 for sign, 8 for exponent, and 23 for fraction. - `float32` is the baseline for deep learning. Scientific computing often uses `float64` (double precision). - Deep learning is "sloppier" and `float32` is usually sufficient.

Memory usage of tensors: Memory usage is determined by: 1. The number of values (elements) in the tensor. 2. The data type of each value.

```
def tensors_memory():
    x = torch.zeros(4, 8) # @inspect x
    assert x.dtype == torch.float32 # Default type
    assert x.size() == torch.Size([4, 8])
    assert x.numel() == 32 # 4 * 8 = 32
    assert x.element_size() == 4 # Float is 4 bytes

    # One matrix in the feedforward layer of GPT-3:
    # assert get_memory_usage(torch.empty(12288 * 4, 12288)) == 2304 * 1024 * 1024 # 2.3 GB
    # ...which is a lot!
```

The memory usage for a `torch.zeros(4, 8)` tensor is $32 \text{ elements} \times 4 \text{ bytes/element} = 128 \text{ bytes}$. A single matrix in a GPT-3 feedforward layer can consume 2.3 GB of memory.

Float16 (Half-precision) - `float16` (also known as FP16 or half-precision) uses 16 bits: 1 for sign, 5 for exponent, and 10 for fraction. - It cuts memory usage by half. - However, its dynamic range is limited, especially for very small or very large numbers. This can lead to underflow/overflow issues and numerical instability during training, especially for large models.

Bfloat16 (Brain float) - Developed by Google in 2018 to address the dynamic range issue of `float16` for deep learning. - `bfloat16` uses 16 bits: 1 for sign, 8 for exponent, and 7 for fraction. - It has the same memory footprint as `float16` but the same dynamic range as `float32` (due to the 8-bit exponent). - The resolution (fraction bits) is worse than `float16`, but this matters less for deep learning. - `bfloat16` is generally preferred over `float16` for deep learning computations due to its better dynamic range.

```
def tensors_memory():
    # float16
    x = torch.zeros(4, 8, dtype=torch.float16) # @inspect x
    assert x.element_size() == 2 # Half precision is 2 bytes

    # Example of underflow with float16
    x = torch.tensor([1e-8], dtype=torch.float16) # @inspect x
    assert x == 0 # Underflow!
    # If this happens when you train, you can get instability.

    # bfloat16
    x = torch.tensor([1e-8], dtype=torch.bfloat16) # @inspect x
    assert x != 0 # No underflow!

    # Compare dynamic ranges and memory usage
    # float32_info = torch.finfo(torch.float32) # @inspect float32_info
    # float16_info = torch.finfo(torch.float16) # @inspect float16_info
    # bfloat16_info = torch.finfo(torch.bfloat16) # @inspect bfloat16_info
```

FP8 (8-bit float) - Developed by Nvidia in 2022, motivated by machine learning workloads. - Uses 8 bits. H100s support two variants of FP8: E4M3 (4-bit exponent, 3-bit mantissa) and E5M2 (5-bit exponent, 2-bit mantissa). - FP8 is very crude and has even more limited precision.

Implications on training: - **Training with `float32`:** Works, but requires lots of memory. It's the safest option. - **Training with `fp8`, `float16`, and `bfloat16`:** Risky, and can lead to instability (underflow/overflow). - **Solution:** Use mixed-precision training. This involves using different

precision types for different parts of the model (e.g., `float32` for attention, `bfloat16` for feedforward passes).

Q&A on mixed precision: The general rule of thumb is to use `float32` for parameters and optimizer states (which accumulate over time and require higher precision) and `bfloat16` for transient computations like activations in forward passes.

[01:20:00] Compute Accounting

[01:20:00] Tensors on GPUs

- By default, tensors are stored in CPU memory.
- To leverage the massive parallelism of GPUs, tensors must be explicitly moved to GPU memory. This involves a data transfer cost over the PCI bus.
- It's crucial to always know where your tensor resides (`.device` attribute).

```
def tensors_on_gpus():
    x = torch.zeros(32, 32)
    assert x.device == torch.device("cpu")

    # Let's first see if we have any GPUs.
    if not torch.cuda.is_available():
        return

    num_gpus = torch.cuda.device_count() # @inspect num_gpus
    # properties = torch.cuda.get_device_properties(i) # @inspect properties

    # Move the tensor to GPU memory (device 0).
    y = x.to("cuda:0")
    assert y.device == torch.device("cuda", 0)

    # Or create a tensor directly on the GPU:
    z = torch.zeros(32, 32, device="cuda:0")

    # memory_allocated = torch.cuda.memory_allocated() # @inspect memory_allocated
    # new_memory_allocated = torch.cuda.memory_allocated()
    # memory_used = new_memory_allocated - memory_allocated
    # assert memory_used == 2 * (32 * 32 * 4) # 2 32x32 matrices of 4-byte floats
```

The example shows how to check for GPU availability, move a tensor to GPU, or create one directly on GPU. It also demonstrates how to verify memory allocation.

[01:25:00] Tensor Operations

Most tensors are created by performing operations on other tensors. Each operation has some memory and compute consequences.

Tensor storage (under the hood): - PyTorch tensors are pointers into allocated memory, with metadata describing how to get to any element. - A 2D tensor (matrix) is stored as a 1D array (underlying storage). - Metadata includes `strides`, which indicate how many elements to skip in the underlying storage to move to the next element along a specific dimension. - For a row-major 4x4 matrix, `stride(0)` (to go to next row) is 4, and `stride(1)` (to go to next column) is 1.

```
def tensor_storage():
    # What are tensors in PyTorch?
    # PyTorch tensors are pointers into allocated memory
    # ...with metadata describing how to get to any element of the tensor.

    x = torch.tensor([
        [0, 1, 2, 3],
        [4, 5, 6, 7],
        [8, 9, 10, 11],
        [12, 13, 14, 15],
    ])

    # To go to the next row (dim 0), skip 4 elements in storage.
    assert x.stride(0) == 4
    # To go to the next column (dim 1), skip 1 element in storage.
    assert x.stride(1) == 1

    # To find an element:
    # r, c = 1, 2
    # index = r * x.stride(0) + c * x.stride(1) # @inspect index
    # assert index == 6
```

Tensor slicing (views): - Many operations (like slicing) simply provide a different view of the tensor. - This does not make a copy, so mutations in one tensor affect the other.

```
def tensor_slicing():
    x = torch.tensor([[1, 2, 3], [4, 5, 6]]) # @inspect x

    # Many operations simply provide a different view of the tensor.
    # This does not make a copy, and therefore mutations in one tensor affects the other.

    # Get row 0:
    y = x[0] # @inspect y
    assert torch.equal(y, torch.tensor([1, 2, 3]))
    # assert same_storage(x, y)
```

```
    # Get column 1:
    y = x[:, 1] # @inspect y
    assert torch.equal(y, torch.tensor([2, 5]))
    # assert same_storage(x, y)

    # View 3x2 matrix as 2x3 matrix:
    y = x.view(3, 2) # @inspect y
    assert torch.equal(y, torch.tensor([[1, 2], [3, 4], [5, 6]]))
    # assert same_storage(x, y)

    # Transpose the matrix:
    y = x.transpose(1, 0) # @inspect y
    assert torch.equal(y, torch.tensor([[1, 4], [2, 5], [3, 6]]))
    # assert same_storage(x, y)

    # Check that mutating x also mutates y.
    x[0][0] = 100 # @inspect x
    assert y[0][0] == 100

    # Note that some views are non-contiguous entries, which means that further views aren't po
    x = torch.tensor([[1, 2, 3], [4, 5, 6]]) # @inspect x
    y = x.transpose(1, 0) # @inspect y
    assert not y.is_contiguous()

    # try:
    #     y.view(2, 3)
    # except RuntimeError as e:
    #     assert "view size is not compatible with input tensor's size and stride" in str(e)

    # One can enforce a tensor to be contiguous first:
    # y = x.transpose(1, 0).contiguous().view(2, 3) # @inspect y
    # assert not same_storage(x, y)

    # Views are free, copying take both (additional) memory and compute.
```

- Views are "free" (don't allocate new memory), making them useful for code readability.
- However, operations like `.contiguous()` or `.reshape()` (which implicitly calls `.contiguous()` if needed) can create a copy, consuming additional memory and compute.
- Non-contiguous tensors cannot be directly viewed into arbitrary shapes. They must be made contiguous first, which involves a copy.

Tensor elementwise operations: - Elementwise operations (e.g., `pow`, `sqrt`, division) create new tensors to store results.

```
def tensor_elementwise():
    # These operations apply some operation to each element of the tensor
    # ...and return a new tensor of the same shape.
```

```
    x = torch.tensor([1, 4, 9])
    # assert torch.equal(x.pow(2), torch.tensor([1, 16, 81]))
    # assert torch.equal(x.sqrt(), torch.tensor([1, 2, 3]))
    # assert torch.equal(x / 2, torch.tensor([0.5, 2, 4.5]))

    # triu takes the upper triangular part of a matrix.
    # x = torch.ones(3, 3) # @inspect x
    # assert torch.equal(x.triu(), torch.tensor([[1, 1, 1], [0, 1, 1], [0, 0, 1]]))
    # This is useful for computing an causal attention mask, where M[i, j] is the contribution (
```

Tensor matrix multiplication (`matmul`): - Matrix multiplication is the "bread and butter" of deep learning.

```
def tensor_matmul():
    # Finally, the bread and butter of deep learning: matrix multiplication.
    x = torch.ones(16, 32)
    w = torch.ones(32, 2)
    y = x @ w
    assert y.size() == torch.Size([16, 2])

    # In general, we perform operations for every example in a batch and token in a sequence.
    # x = torch.ones(4, 8, 16, 32)
    # w = torch.ones(32, 2)
    # y = x @ w
    # assert y.size() == torch.Size([4, 8, 16, 2])
    # In this case, we iterate over values of the first 2 dimensions of x and multiply by w.
```

- In deep learning, operations are often performed on batches of data, where tensors have dimensions like `(batch, sequence, ...)`. PyTorch handles this by iterating over the leading dimensions.

[01:30:00] Tensor Einops

`einops` is a library for manipulating tensors where dimensions are named, inspired by Einstein summation notation.

Motivation (traditional PyTorch code): - Traditional PyTorch code often involves operations like `x @ y.transpose(-2, -1)`, which can be confusing due to the use of negative indices and implicit dimension meanings. - It's easy to mess up dimensions, and comments can get out of sync with the code.

```
def einops_motivation():
    # Traditional PyTorch code:
    # x = torch.ones(2, 2, 3) # batch, sequence, hidden @inspect x
    # y = torch.ones(2, 2, 3) # batch, sequence, hidden @inspect y
    # z = x @ y.transpose(-2, -1) # batch, sequence, sequence @inspect z
    # Easy to mess up the dimensions (what is -2, -1?)
```

Solution: `einops` - `einops` allows you to name dimensions, making tensor operations more explicit and readable.

```
def tensor_einops():
    # Einops is a library for manipulating tensors where dimensions are named.
    # It is inspired by Einstein summation notation (Einstein, 1916).
    # [Einops tutorial]
```

Jax typing basics: - `jax_typing` is a library that provides a way to specify dimension names in type hints, improving documentation.

```
def jaxtyping_basics():
    # How do you keep track of tensor dimensions?
    # Old way:
    # x = torch.ones(2, 1, 3) # batch seq heads hidden @inspect x

    # New (jaxtyping) way:
    # x: Float[torch.Tensor, "batch seq heads hidden"] = torch.ones(2, 1, 3) # @inspect x
    # Note that this is just documentation (no enforcement).
```

`einops_einsum` (generalized matrix multiplication with good bookkeeping): - `einsum` allows you to specify input and output dimensions by name. Dimensions not named in the output are summed over.

```
def einops_einsum():
    # Einsum is generalized matrix multiplication with good bookkeeping.
    # Define two tensors:
    # x: Float[torch.Tensor, "batch seq1 hidden"] = torch.ones(2, 3, 4) # @inspect x
    # y: Float[torch.Tensor, "batch seq2 hidden"] = torch.ones(2, 3, 4) # @inspect y

    # Old way:
    # z = x @ y.transpose(-2, -1) # batch, sequence, sequence @inspect z
```

```
    # New (einops) way:
    # z = einsum(x, y, "batch seq1 hidden, batch seq2 hidden -> batch seq1 seq2") # @inspect z
    # Dimensions that are not named in the output are summed over.

    # Or can use ... to represent broadcasting over any number of dimensions:
    # z = einsum(x, y, "..., seq1 hidden, ... hidden -> ... seq1 seq2") # @inspect z
```

- `einsum` automatically figures out the best way to reduce and optimize the operation. When used with `torch.compile`, it can reuse the same optimized implementation.

`einops_reduce`: - `reduce` operates on one tensor to aggregate dimensions (e.g., sum, mean, max, min).

```
def einops_reduce():
    # You can reduce a single tensor via some operation (e.g., sum, mean, max, min).
    # x: Float[torch.Tensor, "batch seq hidden"] = torch.ones(2, 3, 4) # @inspect x

    # Old way:
    # y = x.mean(dim=-1) # @inspect y

    # New (einops) way:
    # y = reduce(x, "... hidden -> ...", "sum") # @inspect y
```

`einops_rearrange`: - `rearrange` allows reshaping and reordering dimensions, useful when one dimension represents two underlying dimensions that need to be broken up or combined.

```
def einops_rearrange():
    # Sometimes, a dimension represents two dimensions
    # ...and you want to operate on one of them.

    # x: Float[torch.Tensor, "batch seq total_hidden"] = torch.ones(2, 3, 8) # @inspect x
    # ...where total_hidden is a flattened representation of heads * hidden1
    # w: Float[torch.Tensor, "hidden1 hidden2"] = torch.ones(4, 4)

    # Break up total_hidden into two dimensions (heads and hidden1):
    # x = rearrange(x, "..., (heads hidden1) -> ... heads hidden1", heads=2) # @inspect x

    # Perform the transformation by w:
    # x = einsum(x, w, "..., hidden1, hidden1 hidden2 -> ... hidden2") # @inspect x

    # Combine heads and hidden2 back together:
    # x = rearrange(x, "..., heads hidden2 -> (heads hidden2)") # @inspect x
```

- `rearrange` is like a fancier version of `view`, allowing explicit naming of dimensions during reshaping.
- `rearrange` can also be used to combine dimensions back together.

[01:35:00] Tensor Operations FLOPs

Floating-point operation (FLOP): A basic operation like addition ($x + y$) or multiplication ($x \times y$).

Confusing acronyms: - **FLOPs:** Floating-point operations (measure of computation done). - **FLOP/s:** Floating-point operations per second (measure of hardware speed). - The lecture will use FLOP/s to avoid confusion.

Intuitions about FLOPs: - Training GPT-3 (2020) took 3×10^{23} FLOPs. - Training GPT-4 (2023) is speculated to take 2×10^{25} FLOPs. - US executive order required reporting models trained with $\geq 10^{26}$ FLOPs (now revoked). - A100 has a peak performance of 312 TFLOP/s. - H100 has a peak performance of 1979 TFLOP/s with sparsity (50% without). This is for structured sparsity where 2 out of 4 elements are zero. If not sparse, it's half that number.

Example calculation: - 8 H100s for 2 weeks: * Total FLOPs = $8 \times (60 \times 60 \times 24 \times 7 \times 2) \times \text{H100\_flop\_per\_sec} \approx 4.788 \times 10^{21}$ FLOPs.

[01:40:00] Linear Model Example

Let's consider a simple linear model: $y = x \times w$.

```
def tensor_operations_flops():
    # As motivation, suppose you have a linear model.
    # We have n points
    # Each point is d-dimensional
    # The linear model maps each d-dimensional vector to a k outputs

    # B = 16384 # Number of points
    # D = 32768 # Dimension
    # K = 8192 # Number of outputs

    # device = get_device()
    # x = torch.ones(B, D, device=device)
    # w = torch.randn(D, K, device=device)
    # y = x @ w
```

```
    # We have one multiplication (x[i][j] * w[j][k]) and one addition per (i, j, k) triple.
    # actual_num_flops = 2 * B * D * K # @inspect actual_num_flops
```

- For a matrix multiplication of dimensions $B \times D$ and $D \times K$, the output is $B \times K$.
- Each element in the output requires D multiplications and $D - 1$ additions.
- So, for each element in the output, there are approximately $2D$ FLOPs.
- Total FLOPs = $B \times K \times (2D) = 2 \times B \times D \times K$.
- This is a key formula to remember for matrix multiplication FLOPs.

FLOPs of other operations: - Elementwise operations on an $m \times n$ matrix require $O(mn)$ FLOPs. - Addition of two $m \times n$ matrices requires mn FLOPs. - In general, no other operation you'd encounter in deep learning is as expensive as matrix multiplication for large enough matrices. - This is why napkin math often simplifies to only consider matrix multiplications. Hardware is designed for large matrix multiplications, so for efficiency, models should leverage this.

Interpretation: - B is the number of data points (or tokens). - $D \times K$ is the number of parameters. - FLOPs for forward pass is $2 \times (\text{number of tokens}) \times (\text{number of parameters})$. This generalizes to Transformers.

How do our FLOPs calculations translate to wall-clock time (seconds)? - We can time the matrix multiplication and calculate the actual FLOP/s.

```
def tensor_operations_flops():
    # Let us time it!
    # actual_time = time_matmul(x, w) # @inspect actual_time
    # actual_flop_per_sec = actual_num_flops / actual_time # @inspect actual_flop_per_sec
```

- The example shows an actual time of 0.163 seconds for the matrix multiplication, resulting in an actual FLOP/s of 5.4×10^{13}.

Model FLOPs Utilization (MFU): - **Definition:** MFU = (actual FLOP/s) / (promised FLOP/s) (ignoring communication/overhead). - MFU is a measure of how well the model utilizes the theoretical peak performance of the hardware. - Usually, MFU of > 0.5 is considered quite good, and will be higher if matrix multiplications dominate the workload. - For the example, the promised FLOP/s for H100 (float32) is 6.75×10^{13}. - MFU = $(5.4 \times 10^{13})/(6.75 \times 10^{13}) \approx 0.801$. This is a good MFU.

MFU with bfloat16: - If we perform the same matrix multiplication with `bfloat16`, the actual time is 0.032 seconds (much faster). - The actual FLOP/s is 2.735×10^{14} (higher). - The promised FLOP/s for H100 (bfloat16) is 9.895×10^{14}. - MFU = $(2.735 \times 10^{14})/(9.895 \times 10^{14}) \approx 0.276$. - The MFU is lower for `bfloat16` because the promised FLOP/s is much higher, making it harder to fully utilize.

[01:50:00] Gradients FLOPs

We've discussed forward pass FLOPs. Now let's consider the FLOPs for computing gradients (backward pass).

Example: Simple linear model - Loss: $y = 0.5 \times (x \times w - 5)^2$.

```
def gradients_basics():
    # Forward pass: compute loss
    # x = torch.tensor([1., 2, 3])
    # w = torch.tensor([1., 1, 1], requires_grad=True) # Want gradient
    # pred_y = x @ w
    # loss = 0.5 * (pred_y - 5).pow(2)

    # Backward pass: compute gradients
    # loss.backward()
    # assert loss.grad is None
    # assert pred_y.grad is None
    # assert x.grad is None
    # assert torch.equal(w.grad, torch.tensor([1, 2, 3]))
```

FLOPs for computing gradients: - Let's consider a two-layer linear network: $x \to w_1 \to h_1 \to w_2 \to h_2 \to \text{loss}$. - We need to compute gradients with respect to h_1, h_2, w_1, and w_2. - Focus on computing the gradient for w_2 using the chain rule. - The gradient dL/dw_2 involves operations similar to matrix multiplication. - The number of FLOPs for the backward pass (computing dL/dw_2) is $2 \times B \times D \times K$. - Similarly, computing dL/dh_1 also involves matrix multiplication-like operations. - The total FLOPs for the backward pass (for all parameters) is approximately $4 \times (\text{number of data points}) \times (\text{number of parameters})$.

Summary of FLOPs: - **Forward pass:** $2 \times (\text{number of data points}) \times (\text{number of parameters})$ FLOPs. - **Backward pass:** $4 \times (\text{number of data points}) \times (\text{number of parameters})$ FLOPs. - **Total:** $6 \times (\text{number of data points}) \times (\text{number of parameters})$ FLOPs. - This explains the "6" factor in the initial napkin math question.

[02:00:00] Models

[02:00:00] Module Parameters

- Parameters in PyTorch are stored as `nn.Parameter` objects.
- Parameter initialization is crucial for training stability.
- If parameters are initialized with large values, gradients can blow up, leading to unstable training.
- A common technique is to rescale initial weights by $1/\sqrt{\text{num\_inputs}}$ (Xavier initialization).
- For extra safety, weights can be truncated to a specific range (e.g., [-3, 3]).

[02:05:00] Custom Model

Let's build a simple deep linear model using `nn.Parameter`.

```
def custom_model():
    # Let's build up a simple deep linear model using nn.Parameter.
    # D = 64 # Dimension
    # num_layers = 2
    # model = Cruncher(dim=D, num_layers=num_layers).to(get_device())

    # param_sizes = []
    # for name, param in model.state_dict().items():
    #     param_sizes.append((name, param.numel()))

    # assert param_sizes == [('layers.0.weight', D * D), ('layers.0.bias', D),
    #                        ('layers.1.weight', D * D), ('layers.1.bias', D),
    #                        ('final.weight', D), ('final.bias', 1)]

    # num_parameters = get_num_parameters(model)
    # assert num_parameters == (D * D) + D + (D * D) + D + D + 1

    # Remember to move the model to the GPU.
    # device = get_device()
    # model = model.to(device)

    # Run the model on some data.
    # B = 8 # Batch size
    # x = torch.randn(B, D, device=device)
    # y = model(x)
    # assert y.size() == torch.Size([B, 1])
```

- The `Cruncher` model is a custom `nn.Module` with `num_layers` linear layers, each followed by a linear layer for the final output.
- Each `Linear` layer consists of a weight matrix and a bias vector.

- The total number of parameters in this model is $D^2 \times (\text{num\_layers}) + D \times (\text{num\_layers}) + D + 1$.

[02:10:00] Training Loop and Best Practices

Randomness: - Randomness appears in many places: parameter initialization, dropout, data ordering. - For reproducibility, it's recommended to always pass a different random seed for each use of randomness. - Determinism is particularly useful when debugging. - In PyTorch, there are multiple places to set random seeds (torch, numpy, python's random module). It's best to set them all.

Data loading: - In language modeling, data is a sequence of integers (output by the tokenizer). - It's convenient to serialize data as numpy arrays. - For large datasets (e.g., LLaMA data is 2.8 TB), you don't want to load the entire data into memory at once. - Use `np.memmap` to lazily load only the accessed parts into memory. This allows treating a file on disk as a numpy array. - A data loader generates batches of sequences for training.

Optimizer: - We've defined our model. Now let's define the optimizer. - **SGD (Stochastic Gradient Descent):** Computes gradients for a batch and takes a step. - **Momentum:** SGD + exponential averaging of gradients. - **AdaGrad:** SGD + averaging by squared gradients. Scales gradients by the inverse of the square root of past squared gradients. - **RMSprop:** AdaGrad + exponentially averaging of squared gradients. - **Adam:** RMSprop + momentum. Combines exponential averaging of gradients and squared gradients.

```
def optimizer():
    # Recall our deep linear model.
    # B = 2
    # D = 4
    # num_layers = 2
    # model = Cruncher(dim=D, num_layers=num_layers).to(get_device())

    # Let's define the AdaGrad optimizer
    # AdaGrad:
    #   g2 = sum_{t=0}^{t} g_t^2
    #   p_t+1 = p_t - lr * g_t / sqrt(g2 + 1e-5)

    # Compute gradients
    # x = torch.randn([4., D], device=get_device())
    # y = torch.tensor([4., 5.], device=get_device())
    # pred_y = model(x)
    # loss = F.mse_loss(input=pred_y, target=y)
    # loss.backward()
```

```
    # Take a step
    # optimizer = AdaGrad(model.parameters(), lr=0.01)
    # state = model.state_dict() # @inspect state
    # optimizer.step()
    # state = model.state_dict() # @inspect state

    # Free up the memory (optional)
    # optimizer.zero_grad(set_to_none=True)
```

- The `optimizer.step()` method updates the model parameters based on the computed gradients and the optimizer's internal state.
- The `optimizer.zero_grad()` method clears the gradients of all optimized tensors.

Implementation of AdaGrad: - To implement a custom optimizer in PyTorch, you override the `torch.optim.Optimizer` class. - The `__init__` method takes the model parameters and learning rate. - The `step()` method contains the logic for updating parameters. - It accesses the optimizer's internal state (e.g., `g2` for AdaGrad) for each parameter. - It updates `g2` by adding the square of the current gradient. - It then updates the parameter using the AdaGrad formula: $p_{t+1} = p_t - \text{lr} \times g_t / \sqrt{g2 + 10^{-5}}$.

Memory for optimizer state: - The optimizer state (e.g., `g2` in AdaGrad, or momentum terms in Adam) also consumes memory. - For AdaGrad, it stores one copy of squared gradients for each parameter. So, `num_optimizer_states = num_parameters`. - For Adam, it stores two copies (momentum and squared gradients), so `num_optimizer_states = 2 * num_parameters`.

Total memory for training: - Total memory = $4 \times (\text{num\_parameters} + \text{num\_activations} + \text{num\_gradients} + \text{num\_optimizer\_states})$. - Assuming `float32` (4 bytes per value). - `num_parameters`: $D^2 \times (\text{num\_layers}) + D \times (\text{num\_layers}) + D + 1$. - `num_activations`: $B \times D \times (\text{num\_layers})$. - `num_gradients`: `num_parameters`. - `num_optimizer_states`: `num_parameters` (for AdaGrad).

Compute for one step: - FLOPs for one step = $6 \times B \times \text{num\_parameters}$.

[02:20:00] Training Loop

```
def train_loop():
    # Generate data from linear function with weights (0, 1, 2, ..., D-1).
    # true_w = torch.arange(D, dtype=torch.float32, device=get_device())
    # x = torch.randn(B, D, device=get_device())
```

```
    # true_y = x @ true_w

    # Let's do a basic run
    # train("simple", get_batch, D=D, num_layers=0, B=B, num_train_steps=10, lr=0.01)

    # Do some hyperparameter tuning
    # train("simple", get_batch, D=D, num_layers=0, B=B, num_train_steps=10, lr=0.1)

def train(name: str, get_batch, D: int, num_layers: int, lr: float):
    # model = Cruncher(dim=D, num_layers=num_layers).to(get_device())
    # optimizer = SGD(model.parameters(), lr=lr)

    # for t in range(num_train_steps):
    #     x, y = get_batch(B=B)
    #     pred_y = model(x)
    #     loss = F.mse_loss(pred_y, y)
    #     loss.backward()
    #     optimizer.step()
    #     optimizer.zero_grad(set_to_none=True)
```

Checkpointing: - Training large language models takes a long time and will certainly crash. - To avoid losing progress, it's useful to periodically save your model and optimizer state to disk. - When saving, store both `model.state_dict()` and `optimizer.state_dict()`.

```
def checkpointing():
    # Training language models take a long time and will certainly crash.
    # You don't want to lose all your progress.

    # During training, it is useful to periodically save your model and optimizer state to disk.
    # model = Cruncher(dim=64, num_layers=3).to(get_device())
    # optimizer = AdaGrad(model.parameters(), lr=0.01)

    # Save the checkpoint
    # checkpoint = {
    #     "model": model.state_dict(),
    #     "optimizer": optimizer.state_dict(),
    # }
    # torch.save(checkpoint, "model_checkpoint.pt")

    # Load the checkpoint
    # loaded_checkpoint = torch.load("model_checkpoint.pt")
```

Mixed-precision training: - The choice of data type (float32, bfloat16, fp8) has tradeoffs: - Higher precision: more accurate/stable, more memory, more compute. - Lower precision: less accurate/stable, less memory, less compute. - **Solution:** Use `float32` by default, but use `bfloat16` or `fp8` when possible. - **Concrete plan:** - Use `{bfloat16, fp8}` for the forward

pass (activations). - Use `float32` for the rest (parameters, gradients). - This is the idea behind mixed-precision training. - PyTorch has an automatic mixed precision (AMP) library (`torch.cuda.amp`). - Nvidia's Transformer Engine supports FP8 for linear layers.

Final comment: - People are pushing the envelope on what precision is needed. - The challenge with lower precision is numerical instability. Tricks are used to control numerics during training. - Model design and system design are synergistic: models are designed to leverage hardware properties (e.g., Nvidia chips are optimized for lower precision).

← Lec 01

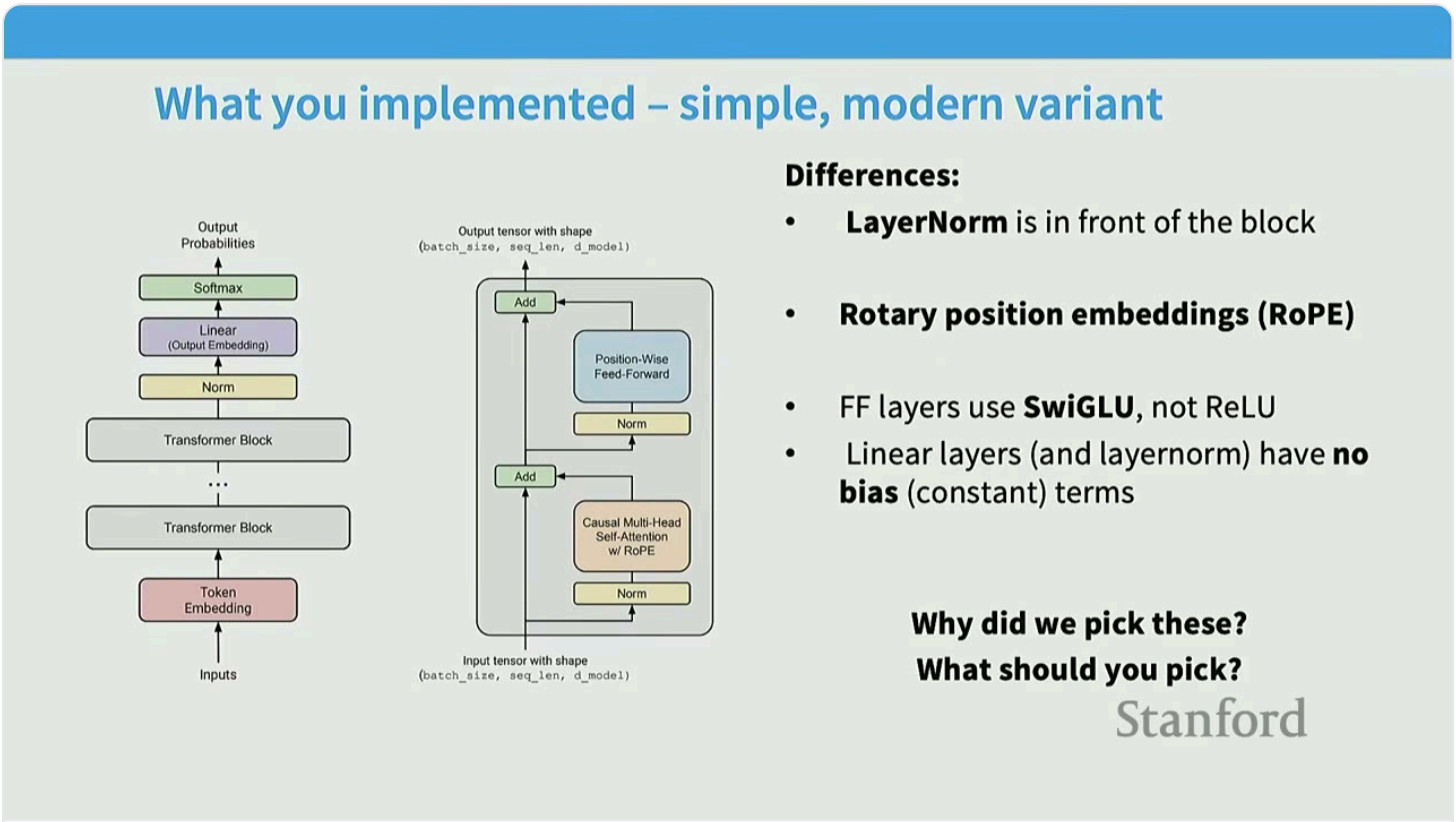
Lec 03 →

Lecture 03: Architectures, Hyperparameters

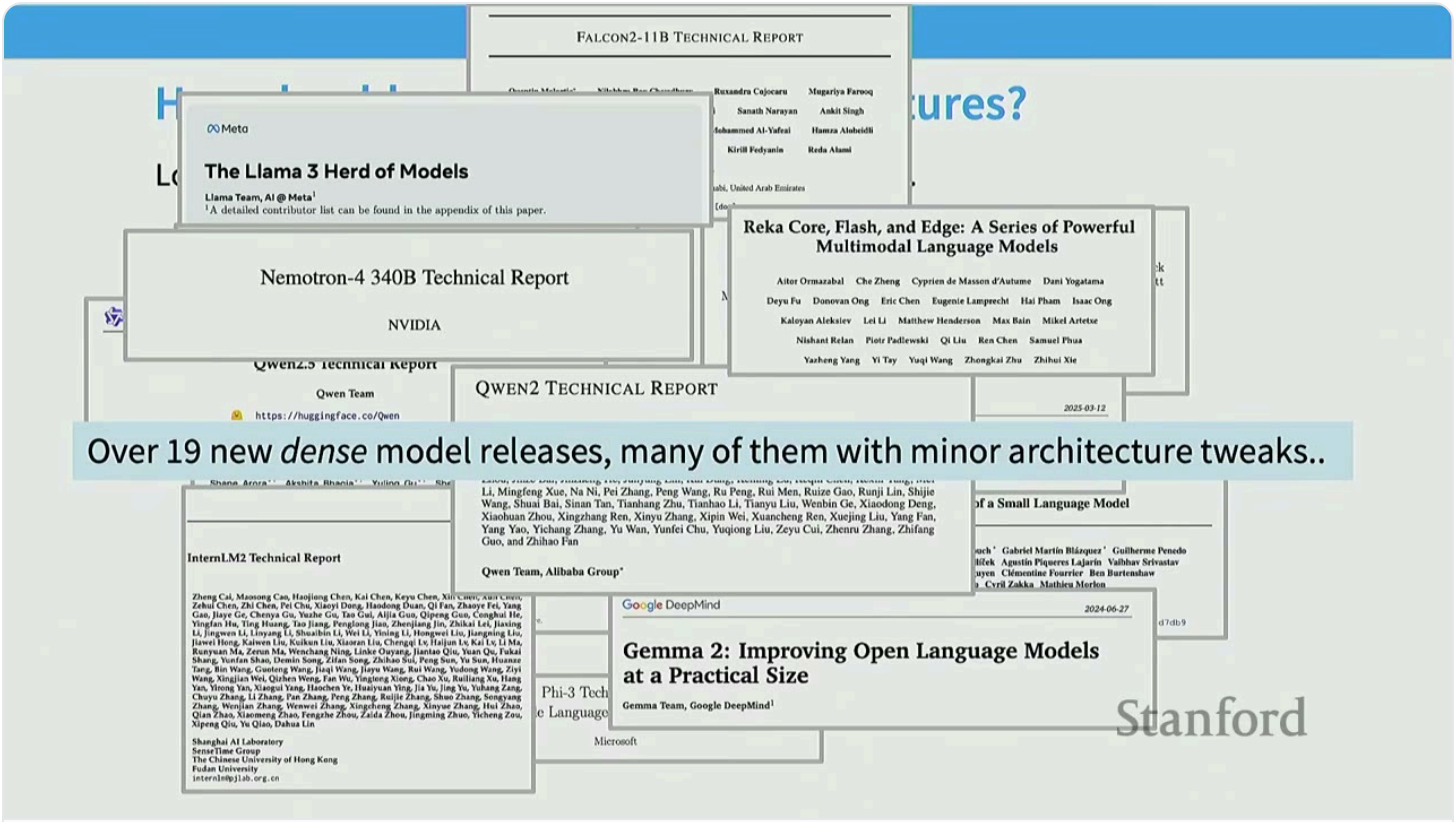
Stanford CS336 · Spring 2025 · 1:27:03 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.





2:23 Slide: "What you implemented - simple, modern variant" with a transformer block diagram and differences listed.



3:37 Slide: "How to pick architectures?" with many research paper titles on language models.

Architecture variations..

Let's think about the core architecture piece

Name	Year	Norm	Parallel Layer	Pre-norm	Position embedding	Activations
Original transformer	2017	LayerNorm	Serial	☐	Sine	ReLU
GPT	2018	LayerNorm	Serial	☐	Absolute	GeLU
T5 (11B)	2019	RMSNorm	Serial	☑	Relative	ReLU
GPT2	2019	LayerNorm	Serial	☑	Sine	GeLU
T5 (XXL 11B) v1.1	2020	RMSNorm	Serial	☑	Relative	GeGLU
mT5	2020	RMSNorm	Serial	☑	Relative	GeGLU
GPT3 (175B)	2020	LayerNorm	Serial	☑	Sine	GeLU
GPTJ	2021	LayerNorm	Parallel	☑	RoPE	GeLU
LaMDA	2021			☑	Relative	GeGLU
Gopher (280B)	2021	RMSNorm	Serial	☑	Relative	ReLU
GPT-NeoX	2022	LayerNorm	Parallel	☑	RoPE	GeLU
BLOOM (176B)	2022	LayerNorm	Serial	☑	AliBi	GeLU
OPT (175B)	2022	LayerNorm	Serial	☑	Absolute	ReLU
PaLM (540B)	2022	RMSNorm	Parallel	☑	RoPE	SwiGLU
Chinchilla	2022	RMSNorm	Serial	☑	Relative	ReLU
Mistral (7B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU
LLaMA2 (70B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU
LLaMA (65B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU
Qwen (14B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU
DeepSeek (67B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU
Yi (34B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU

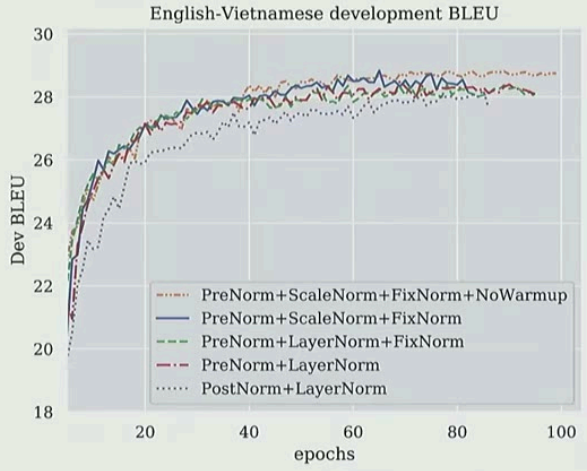
High level view:

- Low consensus (except pre-norm)
- Trends toward 'LLaMA-like' architectures

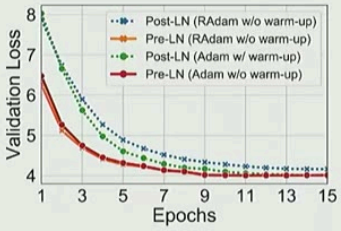
Stanford

5:42 Slide: "Architecture variations.." showing a table comparing different model architectures and trends.

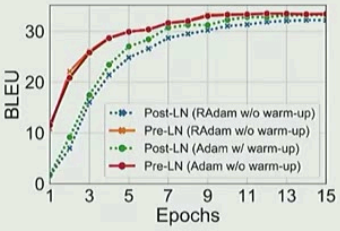
Pre-vs-post-norm, the data



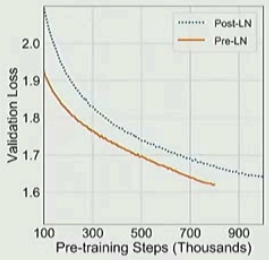
Salazar and Nguyen 2019



(a) Validation Loss (IWSLT)



(b) BLEU (IWSLT)



(a) Validation Loss on BERT

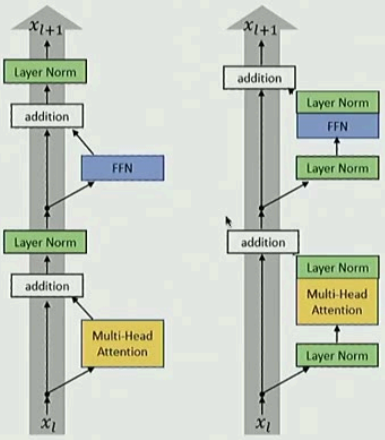
Stanford

Figure from Xiong 2020

7:24 Slide: "Pre-vs-post-norm, the data" with graphs comparing pre-norm and post-norm performance.

New things – 'double' norm.

If putting LayerNorms in residual streams is bad.. Why not post-norm outside the stream?



Stanford

Recent models: Grok, Gemma 2. Olmo 2 *only* does non-residual post norm

9:54 Slide: "New things - 'double' norm." showing diagrams of two different normalization architectures.

LayerNorm vs RMSNorm

Original transformer: **LayerNorm** – normalizes the mean and variance across d_{model}

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Notable models:

GPT3/2/1, OPT, GPT-J, BLOOM

Many modern LMs: **RMSNorm** – does not subtract mean or add a bias term

$$y = \frac{x}{\sqrt{||x||_2^2 + \varepsilon}} * \gamma$$

Notable models:

LLaMA-family, PaLM, Chinchilla, T5

Stanford

11:53 Slide: "LayerNorm vs RMSNorm" with equations for both normalization types and notable models.

Why RMSNorm?

Modern explanation – it's faster (and just as good).

- **Fewer operations** (no mean calculation)
- **Fewer parameters** (no bias term to store)

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Does this explanation make sense?

Operator class	% flop
△ Tensor contraction	99.80
□ Stat. normalization	0.17
○ Element-wise	0.03

Matrix multiplies are the *vast* majority of FLOPs (and memory)

Stanford

[Ivanov et al 2023]

13:30 Slide: "Why RMSNorm?" explaining fewer operations/parameters, with a table of flop percentages.

RMSNorm - validation

RMSNorm runtime (and surprisingly, perf) gains have been seen in papers

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	$223M$	$11.1T$	3.50	2.182 ± 0.005	1.838	71.66	17.78	23.02	26.62
RMS Norm	$223M$	$11.1T$	3.68	2.167 ± 0.008	**1.821**	**75.45**	**17.94**	**24.07**	**27.14**
Rezero	$223M$	$11.1T$	3.51	2.262 ± 0.003	1.939	61.69	15.64	20.90	26.37
Rezero + LayerNorm	$223M$	$11.1T$	3.26	2.223 ± 0.006	1.858	70.42	17.58	23.02	26.29
Rezero + RMS Norm	$223M$	$11.1T$	3.34	2.221 ± 0.009	1.875	70.33	17.32	23.02	26.19
Fixup	$223M$	$11.1T$	2.95	2.382 ± 0.012	2.067	58.56	14.42	23.02	26.31

Narang et al 2020

Stanford

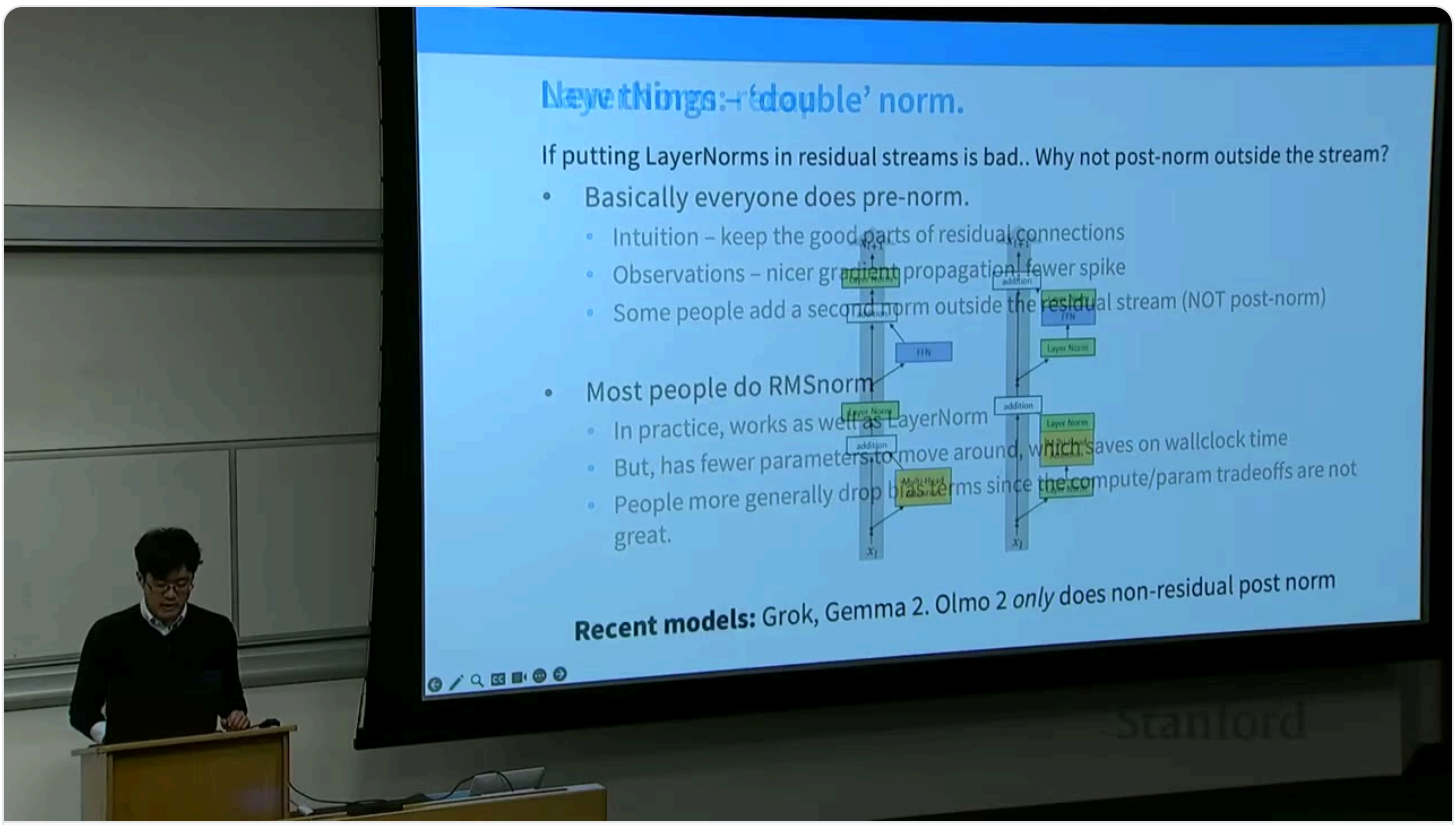
15:14 Slide: "RMSNorm - validation" showing a table comparing RMSNorm performance with other models.

LayerNorm: recap

- Basically everyone does pre-norm.
  - Intuition – keep the good parts of residual connections
  - Observations – nicer gradient propagation, fewer spike
  - Some people add a second norm outside the residual stream (NOT post-norm)
- Most people do RMSnorm
  - In practice, works as well as LayerNorm
  - But, has fewer parameters to move around, which saves on wallclock time
  - People more generally drop bias terms since the compute/param tradeoffs are not great.

Stanford

17:24 Slide: "LayerNorm: recap" summarizing pre-norm, RMSNorm, and bias term practices.

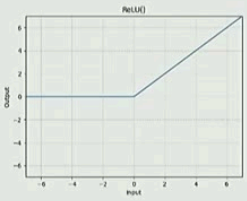


18:39 A speaker presenting in front of a screen displaying the "New things - 'double' norm" slide.

A few of the common activations

ReLU

$$FF(x) = \max(0, xW_1)\, W_2$$

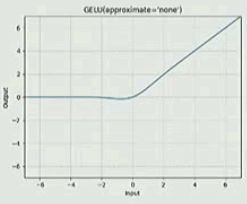


Notable models:

Original transformer, T5, Gopher, Chinchilla, OPT

GeLU

$$FF(x) = \text{GELU}(xW_1)W_2$$
$$GELU(x) \coloneqq x\Phi(x)$$



Notable models:

GPT1/2/3, GPTJ, GPT-Neox, BLOOM

SwiGLU / GeGLU (next slide..)

Notable models:

Llama, PaLM,T5 v1.1, *most* models post 2023

Stanford

20:38 Slide: "A few of the common activations" showing ReLU and GeLU equations, graphs, and notable models.

Gated variants of standard FF layers

GeGLU

$$\text{FFN}_{\text{GEGLU}}(x, W, V, W_2) = (\text{GELU}(xW) \otimes xV)W_2$$

Notable models:

T5 v1.1, mT5, LaMDA, Phi3, Gemma 2, Gemma 3

SwiGLU (swish is $x * \text{sigmoid}(x)$)

$$\text{FFN}_{\text{SwiGLU}}(x, W, V, W_2) = (\text{Swish}_1(xW) \otimes xV)W_2$$

Notable models:

LLaMa 1/2/3, PaLM, Mistral, OlMo, *most models post 2023*

Note: Gated models use smaller dimensions for the d_{ff} by 2/3

Stanford

23:20 Slide: "Gated variants of standard FF layers" showing GeGLU and SwiGLU equations and notable models.

TL;DR - Modern transformer architectures have converged on pre-norm, RMSNorm, and gated linear units (GLUs) for better stability and performance. - Hyperparameter choices, especially the feed-forward dimension to model dimension ratio, show surprising consensus (4x for ReLU, 2.66x for GLU). - Weight decay in large language models (LLMs) is used more for optimization dynamics than for controlling overfitting. - Stability tricks, particularly related to softmax operations, are crucial for training large models. - Innovations like Multi-Query Attention (MQA) and Grouped-Query Attention (GQA) improve inference efficiency, while sparse attention patterns (e.g., sliding window) enable longer context windows.

Key concepts - Pre-norm vs. Post-norm - LayerNorm vs. RMSNorm - Gated Linear Units (GLUs) - Feed-forward dimension ratio - Head dimension to model dimension ratio - Aspect ratio (model dimension vs. number of layers) - Vocabulary sizes - Regularization (Dropout, Weight Decay) - Stability tricks (Z-loss, QK norm, Logit soft-capping) - Attention heads (GQA/MQA, Sparse/Sliding Window Attention) - KV cache

---

[00:00] **Lecture 3: Architectures, Hyperparameters**

This lecture will delve into the "nitty-gritty details" of Language Model (LM) architecture and training, covering aspects often omitted in other courses. The goal is to learn from the collective experience of those who have trained many LLMs.

[00:46] **Outline and Goals** 1. Quick recap of the "standard" transformer (what you implement). 2. What do most of the large LMs have in common? 3. What are common variations to the architecture/training process?

Today's theme: The best way to learn is hands-on experience; the second best way is to try to learn from others' experience.

[01:42] **Starting Point: The 'Original' Transformer**

The original Transformer architecture (from "Attention Is All You Need") consists of: - **Position Embeddings**: Sine and cosine functions. - **Feed-Forward Network (FFN)**: ReLU activation, $FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$. - **Normalization**: Post-LayerNorm (LayerNorm applied *after* the residual connection).

We will examine variations in these components, leading to the most modern Transformer variants.

[02:17] **What You Implemented - Simple, Modern Variant**

The Transformer variant implemented in the assignment is a modern version, not the original "vanilla" Transformer. Key differences include: - **LayerNorm**: Applied *in front* of the block (pre-norm). - **Position Embeddings**: Rotary Position Embeddings (RoPE). - **FFN Activation**: Uses SwiGLU, not ReLU. - **Linear Layers**: Linear layers (and LayerNorm) have no bias (constant) terms.

The question arises: Why these specific choices? We will explore these decisions based on empirical evidence from various LLMs.

[03:07] **How Should We Think About Architectures?**

The field of LLM architectures is rapidly evolving. In the last year alone, there have been over 19 new dense model releases, many with minor architectural tweaks. This proliferation of models (e.g., Command A, OLMo, Gemma, Qwen, Mistral, Falcon) presents a challenge but also a wealth of information. By analyzing what these models have in common and what parts vary, we can understand which architectural choices are truly important.

[04:30] **What Are We Going to Cover?** 1. **Common architecture variations**: Activations, FFN, Attention variants, Position embeddings. 2. **Hyperparameters that (do or don't) matter**: What is `ff_dim`? Do `multi_head_dims` always sum to `model_dim`? How many vocab elements? 3. **Stability tricks**.

[05:15] **Architecture Variations**

A high-level overview of various LLMs (from 2017 to 2024) reveals: - **Low consensus** on many architectural choices, with the exception of pre-norm. - **Trends towards 'LLaMA-like' architectures** in recent years.

[05:54] **Pre-vs-Post Norm**

The original Transformer used **post-norm**, where LayerNorm is applied *after* the residual connection.

$$x_{l+1} = LayerNorm(x_l + MultiHeadAtt(x_l))$$

$$x_{l+1} = LayerNorm(x_l + FFN(x_l))$$

However, very early on, researchers found that moving LayerNorm to *before* the block (pre-norm) led to much better results.

$$x_{l+1} = x_l + MultiHeadAtt(LayerNorm(x_l))$$

$$x_{l+1} = x_l + FFN(LayerNorm(x_l))$$

Almost all modern LMs use pre-norm, with OPT-350M being a notable exception.

[07:13] **Pre-vs-Post Norm, the Data**

Early papers (e.g., Salazar and Nguyen 2019, Xiong 2020) demonstrated the benefits of pre-norm. - **Pre-norm + Stability tricks** (like ScaleNorm + FixNorm) allowed models to train without warm-up, achieving comparable or better performance than post-norm with careful warm-up. - This was observed across various tasks, including machine translation (Dev BLEU) and language modeling (Validation Loss on TWSLT and BERT).

[08:10] **Pre-vs-Post Norm, Explanations?**

- **Gradient Attenuation (Xiong 2020)**: Pre-norm helps maintain constant gradient sizes across layers, preventing them from exploding or vanishing. Post-norm, without warm-up, can lead to exploding gradients.
- **Gradient Spikes (Salazar and Nguyen 2019)**: Pre-norm leads to more stable training with fewer and smaller gradient spikes compared to post-norm.

Today, pre-norm and other LayerNorm tricks are primarily used as **stability-inducing aids** for training large neural networks, especially with larger learning rates.

[09:14] **New Things - 'Double Norm'**

A recent innovation (not present in last year's lectures) is the "double norm" approach. - If putting LayerNorms *in* residual streams is bad, why not put them *outside* the stream? - Models like Grok and Gemma 2 apply LayerNorms both *before* and *after* the attention and FFN blocks (i.e., in front of the residual stream and after the main block output). - OLMo 2 uses LayerNorms *only after* the attention and FFN blocks (non-residual post-norm). - This approach is argued to be even more stable and easier to train for larger models.

[11:59] **LayerNorm vs RMSNorm**

- **Original Transformer: LayerNorm** normalizes the mean and variance across the model dimension (d_{model}).

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} \cdot \gamma + \beta$$

Notable models using LayerNorm include GPT-2/3/OPT-J, BLOOM.

- **Many Modern LMs: RMSNorm** does not subtract the mean or add a bias term.

$$y = \frac{x}{\sqrt{\frac{1}{D}\sum x_i^2 + \epsilon}} \cdot \gamma$$

Notable models using RMSNorm include LLaMA-family, PaLM, Chinchilla, T5.

[12:38] **Why RMSNorm?**

- **Modern explanation**: It's faster and just as good.
  - **Fewer operations**: No mean calculation.
  - **Fewer parameters**: No bias term to store.
- **Does this explanation make sense?**
  - Tensor contractions (matrix multiplies) account for 99.80% of FLOPs, while statistical normalization (LayerNorm/RMSNorm) accounts for only 0.17%. Saving 0.17% of FLOPs doesn't seem like a huge win.
  - **Important lesson**: FLOPs are not runtime! (We will discuss this in far more detail later).
  - While normalization is only 0.17% of FLOPs, it accounts for 25.5% of runtime.
  - RMSNorm can still matter due to the importance of **data movement**. Memory movement overhead is a significant factor in runtime, not just FLOPs.

[15:06] **RMSNorm - Validation**

Narang et al. (2020) showed that RMSNorm provides runtime improvements and, surprisingly, performance gains. - **Vanilla Transformer**: 3.50 steps/s, Final loss 1.838. - **RMSNorm**: 3.68 steps/s, Final loss 1.821. - This is a win-win: faster runtime and lower loss.

[15:58] **More Generally: Dropping Bias Terms**

Most modern Transformers do not have bias terms. - **Original Transformer FFN**: $FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$. - **Most Implementations FFN (if not gated)**: $FFN(x) = \sigma(xW_1)W_2$. - **Reasons**: Similar to RMSNorm, it saves memory and improves optimization stability. Dropping bias terms has been empirically observed to stabilize training.

[17:12] **LayerNorm: Recap** - **Basically everyone does pre-norm.** - **Intuition**: Keep the good parts of residual connections. - **Observations**: Nicer gradient propagation, fewer spikes. - **Some people** add a second norm outside the residual stream (not post-norm). - **Most people do RMSNorm.** - In practice, works as well as LayerNorm. - But, has fewer parameters to move

around, which saves on wallclock time. - People more generally drop bias terms since the compute/param tradeoffs are not great.

[18:05] **Activations**

There's a "whole zoo" of activations: ReLU, GeLU, Swish, ELU, GLU, GeGLU, ReGLU, SeLU, SwiGLU, LiGLU. - It really does matter which activation function is chosen. SwiGLU and other GLU variants consistently work well.

[20:07] **A Few of the Common Activations** - **ReLU**: $FFN(x) = \max(0, xW_1)W_2$. - Notable models: Original Transformer, T5, Gopher, Chinchilla, OPT. - **GeLU**: $FFN(x) = GeLU(xW_1)W_2 = x\Phi(x)W_2$. - Notable models: GPT-1/2/3, GPT-Neox, BLOOM. - **SwiGLU / GeGLU** (next slide). - Notable models: LLaMa 1/2/3, PaLM, Mistral, OLMo, most models post 2023.

[21:24] **Gated Activations (*GLU)ivations**

GLUs modify the "first part" of an FF layer. - **Original FFN (with ReLU)**: $FFN(x) = \max(0, xW_1)W_2$. - **Instead of a linear + ReLU, augment the above with an (entrywise) linear term**:

$$\max(0, xW_1) \to \max(0, xW_1) \otimes (xV)$$

- **This gives the gated variant (ReGLU)**:

$$FFN_{ReGLU}(x) = (\max(0, xW_1) \otimes xV)W_2$$

- Note that we have an extra parameter (V).

[22:47] **Gated Variants of Standard FF Layers** - **GeGLU**: $FFN_{GeGLU}(x, W, V, W_2) = (GeLU(xW_1) \otimes xV)W_2$. - Notable models: T5 v1.1, mT5, LLaMDA, Phi3, Gemma 2, Gemma 3. - **SwiGLU**: $FFN_{SwiGLU}(x, W, V, W_2) = (Swish(xW_1) \otimes xV)W_2$. - Notable models: LLaMa 1/2/3, PaLM, Mistral, OLMo, most models post 2023. - **Note**: Gated models use smaller dimensions for the d_{ff} by 2/3 to keep the total parameter count similar to non-gated counterparts.

[25:49] **Do Gated Linear Units Work?**

Yes, fairly consistently so. - Shazeer (2020) showed that GLU variants consistently outperform ReLU on various tasks (e.g., CoLA, SST-2). - FFN$_{ReGLU}$ achieved the highest average score

(84.67) and accuracy (94.38) among all tested FFN variants. - Narang et al. (2020) corroborated these findings, showing that GLU variants consistently achieve lower losses.

[27:54] **Gating, Activations** - Many variations (ReLU, GeLU, GLU) across models. - GLU isn't *necessary* for a good model (see GPT3), but it's probably helpful. - Recent outlier models like Nemotron 340B (Squared ReLU) and Falcon 2 11B (ReLU) also achieve high performance. - But evidence points towards somewhat consistent gains from SwiGLU/GeGLU.

[28:51] **Serial vs Parallel Layers**

Normal Transformer blocks are **serial**: they compute attention, then the MLP. - Input comes in, attention is computed, result is passed to MLP, MLP is computed, result is passed forward. - This serial nature can limit parallelism across GPUs.

[29:40] **Parallel Layers**

A few models (GPT-J, PaLM, GPT-NeoX) do parallel layers. Originally in GPT-J. - **Parallel Layers**: Instead of serial computation, attention and MLP are computed *in parallel* and then added to the residual stream. - Standard (serial) formulation: $y = x + MLP(LayerNorm(x)) + Attention(LayerNorm(x))$. - Parallel formulation: $y = x + MLP(LayerNorm(x)) + Attention(LayerNorm(x))$. (Note: The slide shows the same formula for both serial and parallel, but the key difference is that in parallel, MLP and Attention are computed from the *same* input x and their outputs are summed *before* adding to x). - The parallel formulation can result in roughly 15% faster training speed at large scales, since the MLP and Attention input matrix multiplications can be fused. - If implemented right, LayerNorm can be shared, and matrix multiplies can be fused for systems efficiency. - **Recent Models**: Cohere Command A, Falcon 2 11B, Command R. - However, most models since then have reverted to serial layers, with only a few exceptions.

[30:43] **Summary: Architectures** - **Pre-vs-Post norm**: Everyone does pre-norm (except OPT350M), likely with good reason. - **Layer vs RMSNorm**: RMSNorm has clear compute wins, sometimes even performance wins. - **Gating**: GLUs seem generally better, though differences are small. - **Serial vs parallel layers**: No extremely serious ablations, but has a compute win.

[32:34] **Many Variations in Position Embeddings**

- **Sine embeddings**: Add sines and cosines that enable localization.
  - $PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$
  - $PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$

    - Notable models: Original Transformer.
- **Absolute embeddings**: Add a position vector to the embedding.
    - $Embed(x, i) = V_x + U_i$
    - Notable models: GPT-1/2/3, OPT.
- **Relative embeddings**: Add a vector to the attention computation.
    - $e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij})^T}{\sqrt{d_z}}$
    - Notable models: T5, Gopher, Chinchilla.
- **RoPE embeddings** (next slides).
    - Notable models: GPTJ, PaLM, most 2024+ models.

[33:29] **RoPE: Rotary Position Embeddings**

- **High level thought process**: A relative position embedding should be some function $f(x, i)$ such that:

$$\langle f(x, i), f(y, j) \rangle = g(x, y, i - j)$$

    - That is, the attention function only gets to depend on the relative position $(i - j)$.
    - Existing embeddings do not fully get this property:
        - **Sine**: Has various cross-terms that are not relative.
        - **Absolute**: Obviously not relative.
        - **Relative embeddings**: Is relative, but is not an inner product.

[34:41] **RoPE: Rotary Position Embeddings - How can we solve this problem?**

- We want our embeddings to be invariant to absolute position.
- We know that inner products are invariant to arbitrary rotation.
- **Key Idea**: Rotate the word embeddings based on their position.
    - If "we" is at position 0, and "know" is at position 1, we rotate "know" by 1 unit.
    - If "we" is at position 2, and "know" is at position 3, we rotate "we" by 2 units and "know" by 3 units.
    - The relative angle between the rotated vectors remains the same if their relative positions are the same, preserving the inner product.

[36:16] **RoPE: Rotary Position Embeddings - There are many rotations, which one do you pick?**

- It's easy to think about rotations in 2D. In high-dimensional space, it's not obvious how to rotate.
- **RoPE's solution**: Pair up the coordinates and rotate them in 2D (motivation: complex numbers).
    - Take a high-dimensional vector (e.g., $D = 256$).
    - Cut it into blocks of 2 dimensions.
    - Rotate each 2D block by a specific angle ($\theta_1, \theta_2, \ldots$).
    - The rotation angle for each pair of dimensions is chosen to capture different frequency ranges (some rotate quickly for local context, others slowly for long-range context).

[37:24] **The Actual RoPE Math**

- **Multiply with sines and cosines**.
    - The rotation matrix $R^d_{\Theta,m}$ is a block diagonal matrix where each 2x2 block is a standard 2D rotation matrix.
    - This is applied to the query and key vectors.
- **Difference with sine embeddings**: Not additive, no cross terms. This makes it purely relative.

[37:52] **Implementation and Code for RoPE**

- **Multiply with sines and cosines**.
- **Usual attention stuff**: Query, Key, Value projections.
- **Get the RoPE matrix cos/sin inputs**: Compute rotation angles based on position.
- **Multiply query/key by cos/sin**: Apply the rotations to the query and key vectors.
- **Same stuff as the usual multi-head self attention below**.
- **Note**: Embedding at each attention layer to enforce position invariance. RoPE is applied *within* the attention mechanism, not as an initial embedding.

[39:03] **Hyperparameters**

Transformer hyperparameter questions you might have had in 224n: - How much bigger should the feedforward size be compared to hidden size? - How many heads, and should num_heads always divide hidden size? - What should my vocab size be? - Do people even regularize these LMs? - How do people scale these models - very deep or very wide?

[40:03] **Surprising (?) Consensus Hyperparameter 1 - Feedforward**

- **Feedforward - model dimension ratio**.
  - $FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$.
  - There are two dimensions that are relevant: the feedforward dim (d_{ff}) and model dim (d_{model}).
  - **Relationship**: $d_{ff} = 4 \cdot d_{model}$.
  - This is almost always true. There's just a few exceptions.

[41:00] **Exception #1 - GLU Variants**

- Remember that GLU variants scale down by 2/3. This means most GLU variants have $d_{ff} = \frac{8}{3}d_{model}$. This is mostly what happens.
- **Table of d_{ff}/d_{model} ratios for various models**: | Model | d_{ff}/d_{model} | | :---------- | :---------- | | PaLM | 4 | | Mistral 7B | 3.5 | | LLaMA 2 70B | 3.5 | | LLaMA 70B | 2.68 | | Qwen 14B | 2.67 | | DeepSeek 67B | 2.68 | | Yi 34B | 2.85 | | T5 v1.1 | 2.5 |
- Models are roughly in this range, though PaLM, LLaMA2 and Mistral are slightly larger.
- The 2.66 ratio for GLU variants ensures a similar parameter count to the 4x ratio for ReLU variants.

[42:00] **Exception #2 - T5**

- As we have (and will) see, most LMs are have boring, conservative hyperparameters.
- One exception is T5 (Raffel et al. 2020) which has some very bold settings.
- **In particular, for the 11B model, they set**:
  - $d_{ff} = 65,536$
  - $d_{model} = 1024$
  - This results in an astounding **64-times multiplier** for d_{ff}/d_{model}.
- The T5 paper states: "For '11B' we use $d_{ff} = 65,536$ with 128-headed attention producing a model with about 11 billion parameters. We chose to scale up d_{ff} specifically because modern accelerators (such as the TPUs we train our models on) are most efficient for large dense matrix multiplications like those in the Transformer's feed-forward networks."
- Other recent exceptions (Gemma 2 (8x), SmallLM/Gemma 3 (4x, GLU)) have also used larger multipliers.

[43:00] **Why This Range of Multipliers?**

- Empirically, there's a basin between 1-10 where this hyperparameter is near-optimal.
- Kaplan et al. (2020) showed that the loss increase is minimal for d_{ff}/d_{model} ratios between 1 and 10.

- The 4x multiplier is a reasonable choice within this optimal range.

[44:00] **What Can We Learn from the Model-Dim Hyperparam?**

- The 'default' choices of $d_{ff} = 4d_{model}$ and $d_{ff} = 2.66d_{model}$ have worked well for nearly all modern LLMs.
- But T5 does show that even radical choices of $64d_{model}$ can work. This hyperparameter choice isn't written in stone.
- T5 has a follow-up model (T5 v1.1) that is 'improved' and uses a much more standard 2.5 multiplier on GeGLU, so the 64-times multiplier is likely suboptimal.

[45:00] **Surprising (?) Consensus Hyperparameter 2 - Multi-Head Self-Attention**

- **Head-dim * num-heads to model-dim ratio**.
- Even though we compute h many attention heads, it's not really more costly.
  - We compute $XQ \in \mathbb{R}^{n \times d'}$ and then reshape to $\mathbb{R}^{n \times h \times d/h}$. (Likewise for XK, XV).
  - Then we transpose to $\mathbb{R}^{h \times n \times d/h}$, now the head axis is like a batch axis.
- Almost everything else is identical, and the matrices are the same sizes.
- This doesn't *have* to be true: we can have head-dimensions > model-dim / num-heads.
- But most models do follow this guideline.

[46:00] **How Many Heads, What's the Model Dim?**

- **Table of ratios for various models**: | Model | Num heads | Head dim | Model dim | Ratio ($d_{model}/(h \cdot d_h)$) | | :---------- | :-------- | :------- | :-------- | :---------- | | GPT3 | 96 | 128 | 12288 | 1 | | T5 | 128 | 128 | 1024 | 16 | | T5 v1.1 | 64 | 64 | 4096 | 1 | | LLaMDA | 64 | 128 | 8192 | 1 | | PaLM | 48 | 258 | 18432 | 1.48 | | LLaMa 2 | 64 | 128 | 8192 | 1 |
- Most models have ratios around 1 - notable exceptions by some Google models.

[47:00] **Evidence for 1-1 Ratio?**

- There have been papers against the 1-1 ratio (Bhojanapalli et al. 2020).
- They argued that having very few dimensions per head (due to a 1-1 ratio with many heads) can lead to low-rank bottlenecks, affecting expressiveness.
- But we don't seem to be seeing significant 'low rank bottlenecks' in practice. Most models with a 1-1 ratio perform well.

[47:00] **Aspect Ratios**

- Should my model be deep or wide? How deep and how wide?

- Most models are surprisingly consistent on this one too!
- **Table of d_{model}/n_{layer} ratios for various models**: | Model | d_{model}/n_{layer} | | :---------- | :---------- | | BLOOM | 205 | | T5 v1.1 | 171 | | PaLM (540B) | 156 | | GPT3/OPT/Mistral/Qwen | 128 | | LLaMA/LLaMA2/Chinchilla | 102 | | T5 (11B) | 43 | | GPT2 | 33 |
- There's a sweet spot around 100-200 for this ratio.

[48:00] **Considerations About Aspect Ratio**

- **The Limits of Depth vs Width**: Scaling depth is an obvious limiter. Deep models are harder to parallelize across devices and have higher latency.
- Scaling depth is an obvious limiter, i.e., they are non-parallelizable across different machines or devices and every computation has to always wait for the previous layer. This is unlike width, which can be easily parallelizable over thousands or hundreds of thousands of devices.
- Within the limitation of scaling, a wide range of architectures achieve similar performance.
- **Evidence on aspect ratio scaling (Kaplan et al. 2020)**:
  - Plots of loss increase vs. aspect ratio (d_{model}/n_{layer}) for models of different sizes (50M, 274M, 1.5B parameters).
  - A sweet spot for aspect ratio is observed around 100, where loss increase is minimized.
  - This sweet spot remains relatively consistent across different model sizes.
- **Impact of depth vs. width on upstream vs. downstream performance**:
  - E.K. et al. (Google) found that for upstream tasks (e.g., negative log-perplexity), parameter count is the only thing that matters; deeper models don't help.
  - For downstream tasks (e.g., SuperGLUE accuracy), the story is less clear, but deeper models might be better for the same FLOPs.

[49:00] **What Are Typical Vocabulary Sizes?**

- **Monolingual models**: 30-50k vocab.
  - Original Transformer: 37000
  - GPT: 40257
  - GPT-2/3: 50257
  - T5/T5v1.1: 32128
  - LLaMA: 32000
- **Multilingual / production systems**: 100-250k.

    - mT5: 250000
    - PaLM: 256000
    - GPT4: 100276
    - Command A: 255000
    - DeepSeek: 152064
    - Yi: 64000
- Monolingual vocabs don't need to be huge, but multilingual ones do.
- As LLMs are deployed more widely and interact with diverse languages and modalities (emojis, etc.), larger vocabularies become necessary.
- Larger models can make good use of larger vocabularies.

[50:00] **Dropout and Other Regularization**

- **Do we need regularization during pretraining?**
    - **Arguments against**:
        - There is a lot of data (trillions of tokens), more than parameters.
        - SGD only does a single pass on a corpus (hard to memorize).
    - This is all quite reasonable... but what do people do in practice?

[51:00] **Dropout and Weight Decay in Practice**

- **Table of Dropout and Weight Decay for various models**: | Model | Dropout* | Weight decay | | :---------- | :------- | :---------- | | Original Transformer | 0.1 | 0 | | GPT2 | 0.1 | 0 | | T5 | 0.1 | 0 | | GPT3 | 0.1 | 0 | | T5 v1.1 | 0 | 0.1 | | PaLM | 0 | (variable) | | OPT | 0.1 | 0.1 | | LLaMA | 0 | 0.1 | | Qwen 14B | 0 | 0.1 |
- Many older models used dropout during pretraining.
- Newer models (except Qwen) rely only on weight decay.
- *Most of the time papers just don't discuss dropout. On open models, this closely matches not doing dropout. This may not be true of closed models.*

[52:00] **Why Weight Decay LLMs?**

- Andriushchenko et al. (2023) has interesting observations about LLM weight decay.
- **It's not to control overfitting.**
    - Different amounts of weight decay don't significantly change the ratio of training loss to validation loss.
    - If you train long enough, you end up with the same train-to-val loss gap regardless of weight decay.

- **Weight decay interacts with learning rates (cosine schedule).**
  - With a constant learning rate, models with weight decay start slow but drop rapidly when the learning rate is decreased.
  - With a cosine learning rate schedule, models with high weight decay start slow but optimize very rapidly as the learning rate cools down.
- There's a complex interaction between the optimizer, weight decay, and implicit acceleration at the tail end of training that leads to better models.
- The purpose of weight decay in LLMs is to achieve **better training losses**, not primarily to regularize against overfitting.

[53:00] **Summary: Hyperparameters** - **Feedforward**: Factor-of-4 rule of thumb (8/3 for GLUs) is standard (with some evidence). - **Head dim**: Head dim * Num head = D model is standard. - **Aspect ratio**: Wide range of 'good' values (100-200). Systems concerns dictate the value. - **Regularization**: You still 'regularize' LMs but its effects are primarily on optimization dynamics.

[54:00] **Stability Tricks**

Recently, lots of attention on stable training. - As models get bigger and are trained longer, stability issues become more prominent. - A common problem is exploding gradients, which leads to unstable training and divergence. - The goal is to turn an unstable training curve (like the blue one with high gradient spikes) into a stable one (like the orange one with low gradient norms).

[55:00] **Where Do the Issues Arise? Beware of Softmaxes!**

- Softmaxes can be ill-behaved due to exponentials / division by zero.
- There are two softmaxes in a Transformer:
  1. The output softmax (at the very end of the decoder).
  2. The softmax within the self-attention mechanism.

[56:00] **Output Softmax Stability - The 'Z-loss'**

- Recall the softmax calculation:

$$P(x) = \frac{e^{U_r(x)}}{Z(x)}$$

$$Z(x) = \sum_{r'=1}^{V} e^{U_{r'}(x)}$$

$$\log P(x) = U_r(x) - \log Z(x)$$

- **This is useful for stability!** PaLM pioneered this 'z-loss' trick.
  - They added an auxiliary loss of $Z_{loss} = 10^{-4} \sum_i (\log Z(x_i) - 0)^2$ to encourage the softmax normalizer $\log Z$ to be close to 0.
  - This forces $Z(x)$ to be close to 1.
  - If $Z(x)$ is close to 1, then $\log Z(x)$ is close to 0.
  - This makes $\log P(x) \approx U_r(x)$, effectively removing the problematic exponential and division.
  - This increased the stability of training.
- **Other examples**: Baichuan 2 (2023), DCLM (2024), OLMo 2 (2025).

[57:00] **Attention Softmax Stability - The 'QK Norm'**

- The other softmax we have to deal with is in the attention operation.
- The query and keys are Layer (RMS) normed before going into the softmax operation.
- This is an innovation from the vision and multimodal model community.
  - Dehgani (2023) trained very large vision transformers.
  - IDefCS, Chameleon (Hugging Face) used these tricks for their multimodal components.
- It was then picked up by several others: DCLM, OLMo 2, Gemma 2.
- This technique stabilizes the softmax by bounding the inputs, naturally controlling bad behavior.

[58:00] **Logit Soft-Capping**

- Soft-capping the logits to some maximum value via Tanh.
- Bello et al. (2016) in each attention layer and the final layer ensure that the value of the logits stays between -soft_cap and +soft_cap.
- The logits are passed through: $logits \leftarrow soft\_cap \cdot \tanh(logits / soft\_cap)$.
- Set the soft_cap parameter to 50.0 for the self-attention layers and to 30.0 for the final layer.
- This prevents logits from blowing up, but also might have perf issues.
- This technique hasn't been as widely adopted as QK norm.

[59:00] **Attention Heads**

- Most models don't touch the attention heads much at all with a few minor exceptions.
- **GQA / MQA**: Saving inference costs by reducing the number of heads.

- **Sparse or sliding window attention (GPT4/Mistral)**: Restricting the attention pattern to reduce compute cost.
- **Exotic SSM stuff** (Jamba, Falcon 3, etc): Not covered (sorry!).

[1:00:00] **GQA/MQA - Reducing Attention Head Cost**

- Let's think about the compute involved for attention.
  - Total arithmetic operations: $O(bnd^2)$.
  - Total memory accesses: $O(bnd + bhn^2 + d^2)$.
  - Arithmetic intensity is high $O(\frac{n}{d} + \frac{1}{b})^{-1}$ - need large batches + short seq length (n) or big model dimensions (d) - we can keep our GPUs running.
- **What about the incremental case when we generate text?**
  - **Key difference**: Can't parallelize the generation process - needs to be step by step.
  - In this case - we need to incrementally re-compute/update attention via the 'KV cache'.
  - When generating text, we process one token at a time, conditioning on previous tokens.
  - The KV cache stores the keys and values from previous tokens, so they don't need to be recomputed.
  - This is essential for efficient inference.

[1:02:00] **GQA/MQA - Reducing Attention Head Cost (cont.)**

- **What's the incremental arithmetic intensity?**
  - When using the KV cache, the total arithmetic operations remain $O(bnd^2)$.
  - However, the memory access pattern changes significantly. We are repeatedly loading in matrices (keys, values) from memory.
  - The arithmetic intensity becomes $O(\frac{n}{d} + \frac{1}{b})^{-1}$.
  - This means arithmetic intensity is not good. We need large batches and short sequence lengths, or big model dimensions.
  - The n/d term is difficult to reduce.

[1:03:00] **MQA - Just Have Fewer Key Dimensions.**

- **Key idea**: Have multiple queries, but just one dimension for keys and values.
- Instead of having h separate key and value heads, we have a single key and value head that is shared across all query heads.
- This significantly reduces the size of the KV cache, as we only store one set of keys and values.

- We have much fewer items to move in and out of memory (KV Cache).
- Total memory access: $O(bnd + bn^2 + d^2)$.
- Arithmetic intensity: $O(\frac{n}{d} + \frac{1}{b})^{-1}$.
- This improves inference efficiency, especially for long sequence lengths.

[1:04:00] **Recent Extension - GQA**

- Don't go all the way to one dimension of KV - have fewer dims.
- Instead of a single key and value head (MQA), Grouped-Query Attention (GQA) groups query heads and shares a key/value head among each group.
- This provides a simple knob to control expressiveness (key-query ratio) and inference efficiency.

[1:04:00] **Does MQA Hurt? Sometimes.**

- Some work shows that MQA can lead to a small perplexity hit (Shazeer 2019).
- Other work shows low/no hit with GQA (Ainslie 2023).
- The trade-off is between inference speed and model quality.

[1:04:00] **Sparse / Sliding Window Attention**

- Attending to the entire context can be expensive (quadratic).
- Build sparse / structured attention that trades off expressiveness vs runtime (GPT3).
- Instead of computing attention for all pairs of tokens, we restrict the attention pattern to a local window or a strided pattern.
- This reduces the computational cost from $O(N^2)$ to $O(N \cdot \text{window\_size})$.

[1:04:00] **Sliding Window Attention**

- Another variation on this idea - sliding window attention.
- At each layer, only pay attention to a small region around your current position.
- This also controls the total amount of resources needed for longer contexts.
- Your effective receptive field is now the local one times kind of the layers.

[1:04:00] **Current Standard Trick - Interleave 'Full' and 'LR' Attention**

- From Cohere Command A - Every 4th layer is a full attention.
- This is a very clever trick to both control the systems aspects of things (full attention only happens every now and then) and the length extrapolation aspect.

- Rope only deals with local context windows, and anything that's really really long range has no position embeddings at all.
- This allows for very aggressive extrapolation, as there's no position extrapolation to deal with.

[1:04:00] **Recap, Conclusion, etc.**

- Many aspects (architecture, hyperparameters) of Transformers are in parallel in common across the big LMs.
- **Major differences**: Position embeddings, tokenization.

Practical Takeaways - When building LLMs, prioritize pre-norm for stability. - RMSNorm is generally preferred over LayerNorm for efficiency without sacrificing performance. - GLU variants (GeGLU, SwiGLU) are the current state-of-the-art for FFN activations. - Follow established hyperparameter ratios (e.g., $d_{ff} = 4d_{model}$ or $8/3d_{model}$ for GLUs, $d_{model}/(h \cdot d_h) \approx 1$). - Weight decay is crucial for optimizing LLMs, even if not for traditional overfitting control. - Implement stability tricks, especially for softmax operations (Z-loss, QK norm, logit soft-capping). - Consider MQA/GQA for inference efficiency and sparse attention patterns for longer context windows.

Open Questions / Things to Remember - The exact theoretical reasons for the superior stability of pre-norm and RMSNorm are still being actively researched, but empirical evidence is strong. - The interaction between weight decay and learning rate schedules is complex and crucial for optimal training. - The field is still evolving, with new architectural tweaks and stability tricks emerging regularly (e.g., double norm, interleaved attention patterns). - System-level considerations (memory movement, parallelism constraints) increasingly influence architectural choices.

← Lec 02

Lec 04 →

Lecture 04: Mixture of Experts

Stanford CS336 · Spring 2025 · 1:22:04 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



TL;DR

- **Mixture of Experts (MoE) models** are becoming the standard for high-performance large language models (LLMs) due to their ability to achieve better performance for the same computational cost (FLOPs) during training and inference.
- The core idea of MoE is to replace dense feed-forward networks (FFNs) in a transformer block with multiple smaller FFNs (experts) and a **router** that selectively activates a subset of these experts for each token.
- MoE models offer **more parameters per FLOP**, leading to better memorization and performance, and enable **expert parallelism**, allowing models to be distributed across many

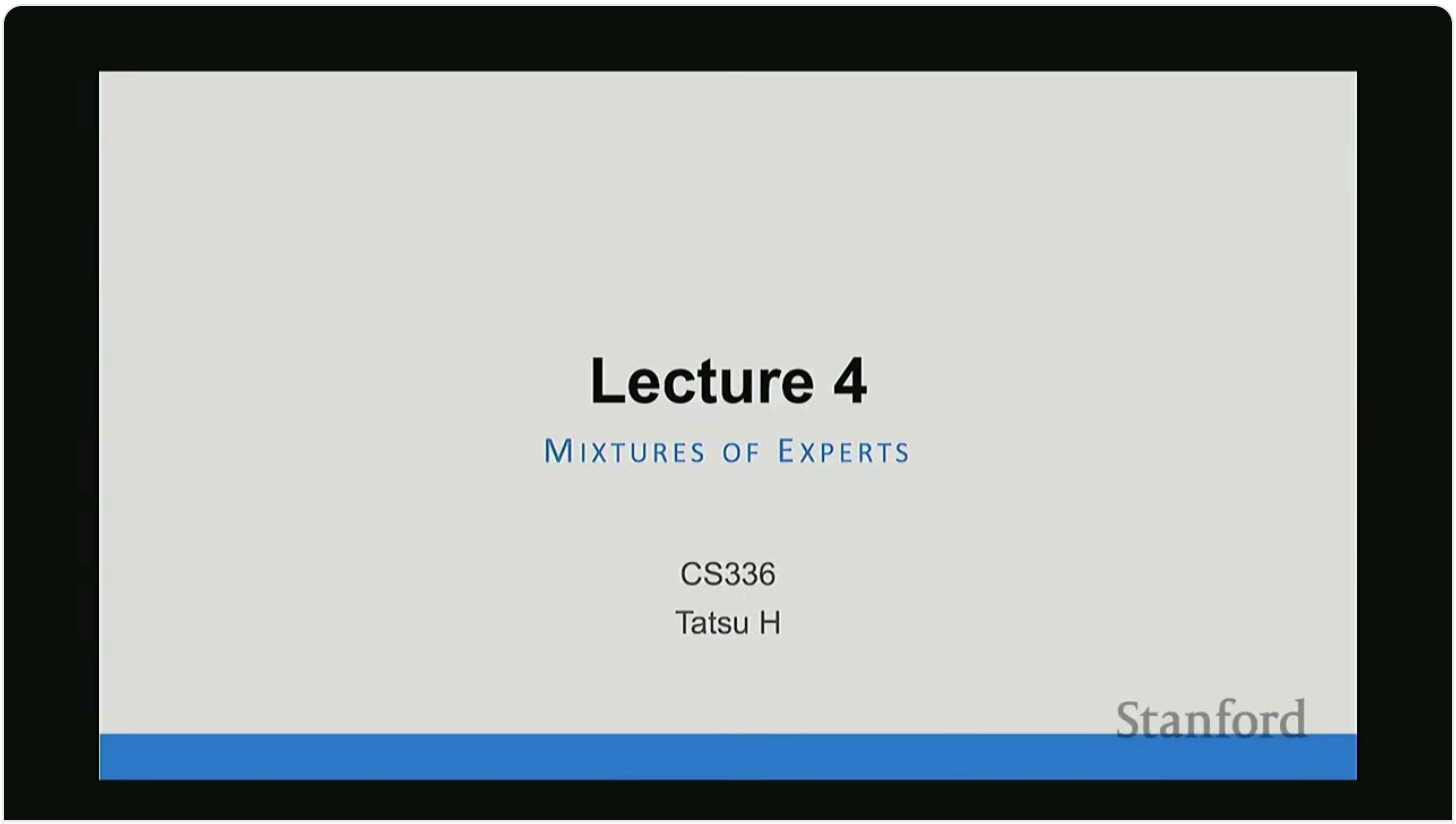
devices.
- Training MoEs is challenging due to the **non-differentiable nature of routing decisions** and potential **instability**. Practical solutions involve heuristic balancing losses and careful architectural choices like fine-grained and shared experts.
- Recent advancements, particularly from DeepSeek, have refined MoE architectures, incorporating **fine-grained experts, shared experts, and sophisticated routing mechanisms** to optimize for both performance and system efficiency.

Key Concepts

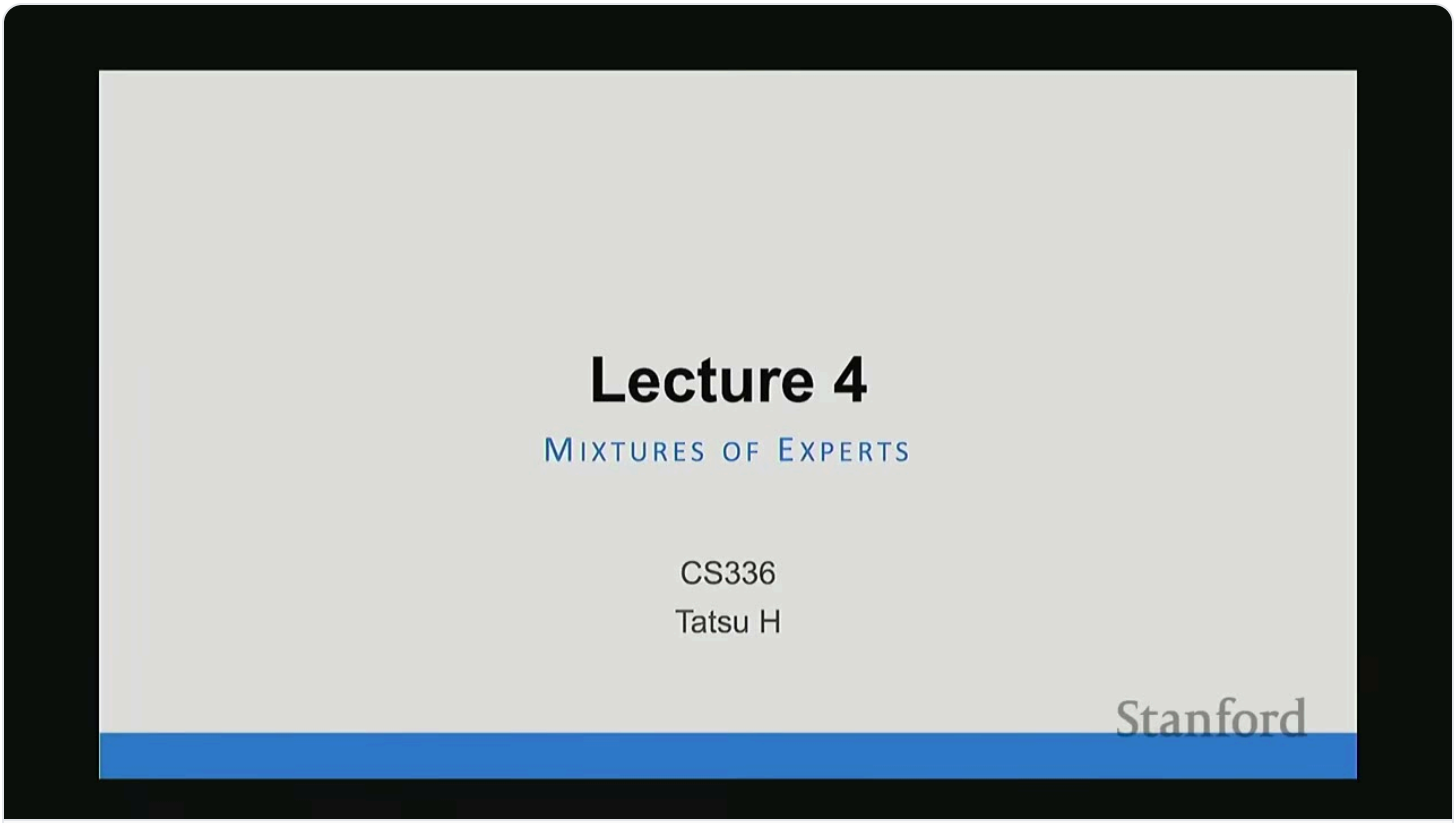
- **Mixture of Experts (MoE)**: A neural network architecture where different "expert" sub-networks are specialized to process different inputs or parts of inputs.
- **Sparse Activation**: Only a subset of experts are activated for a given input, reducing computational cost compared to dense models with similar parameter counts.
- **Router (Gating Network)**: A component that determines which experts are activated for each input token.
- **Experts**: Typically smaller feed-forward networks (FFNs) within a transformer block.
- **FLOPs (Floating Point Operations)**: A measure of computational cost. MoEs aim to increase parameters without increasing FLOPs.
- **Parameters**: The total number of weights and biases in a model. MoEs allow for a massive increase in parameters.
- **Expert Parallelism**: Distributing different experts across different computational devices (GPUs/TPUs) to speed up training and inference.
- **Top-K Routing**: A common routing strategy where the router selects the K experts with the highest scores for each token.
- **Fine-grained Experts**: Dividing the original FFN into smaller, more numerous experts to increase the total parameter count without increasing FLOPs.
- **Shared Experts**: Experts that are always active for all tokens, handling common processing tasks.
- **Load Balancing Losses**: Heuristic losses added to the training objective to encourage the router to distribute tokens evenly among experts, preventing "dead" experts.
- **Stochastic Approximations**: Adding noise or perturbations to routing decisions to encourage exploration and robustness.
- **Upcycling**: A training method where a pre-trained dense model is used to initialize an MoE, leveraging existing knowledge to accelerate MoE training.

- **Multi-head Latent Attention (MLA)**: An attention optimization that projects keys and values into a lower-dimensional "latent" space for caching, reducing KV cache size.

[0:00] Introduction to Mixture of Experts (MoE)



0:12 Lecture 4 title slide: Mixtures of Experts, CS336, Tatsu H, Stanford.



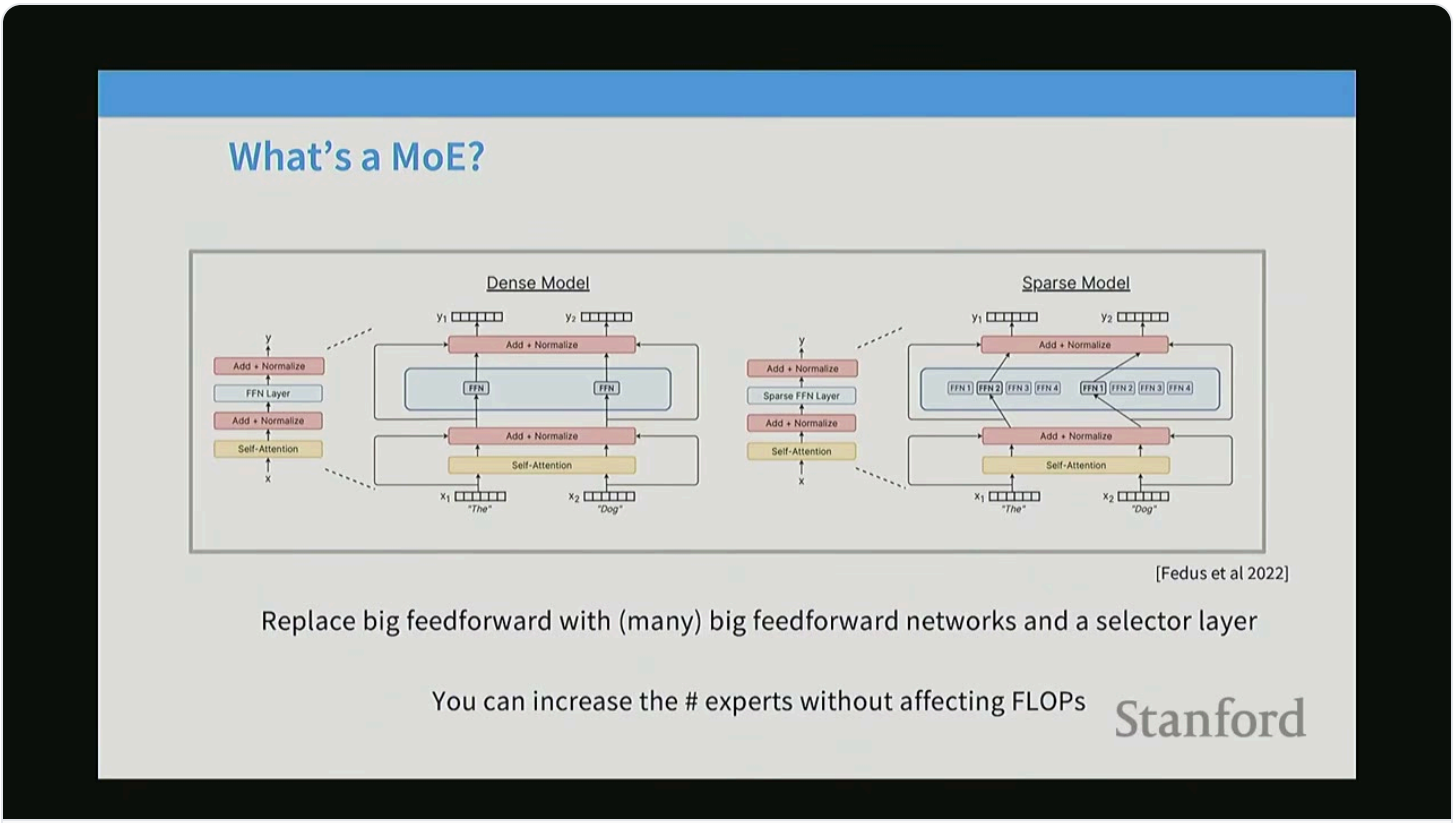
0:30 Diagram comparing dense and sparse models, illustrating how MoE replaces big feedforward networks.

The lecture begins by introducing Mixture of Experts (MoE) as a critical and increasingly popular topic in large language model (LLM) development. Initially a "bonus" topic, MoEs have become central to modern high-performance systems.

Recent LLMs adopting MoE architectures include: * Mistral AI * Grok * DeepSeek * Llama 4 (Maverick) * GPT-4 (speculated as GPT-MoE-1.8T)

The advantage of MoEs over dense architectures is now "very much clear," offering benefits across all compute scales. Understanding MoEs is crucial for building state-of-the-art models within given computational budgets (FLOPs).

[1:34] What is an MoE?



2:14 Graph showing faster training for MoE models, with 7x speedup and improved perplexity.

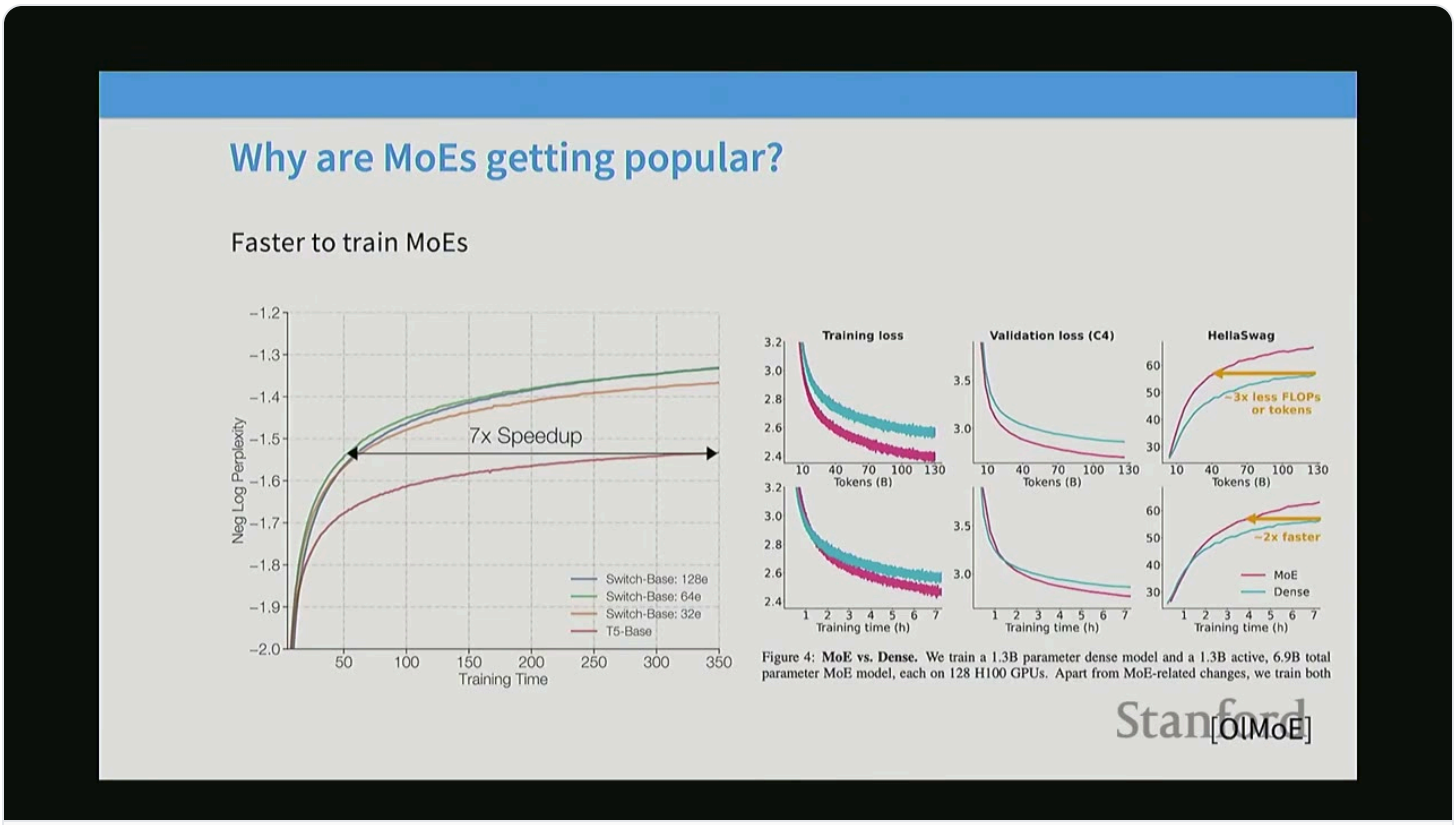
An MoE is a type of neural network architecture where different "expert" sub-networks are activated sparsely. The name "Mixture of Experts" can be misleading; it doesn't necessarily imply specialization in different domains (e.g., a "coding expert" and an "English expert"). Instead, it refers to a specific architectural pattern.

The core idea of MoE primarily applies to the **Multi-Layer Perceptrons (MLPs)** or **Feed-Forward Networks (FFNs)** within a transformer block.

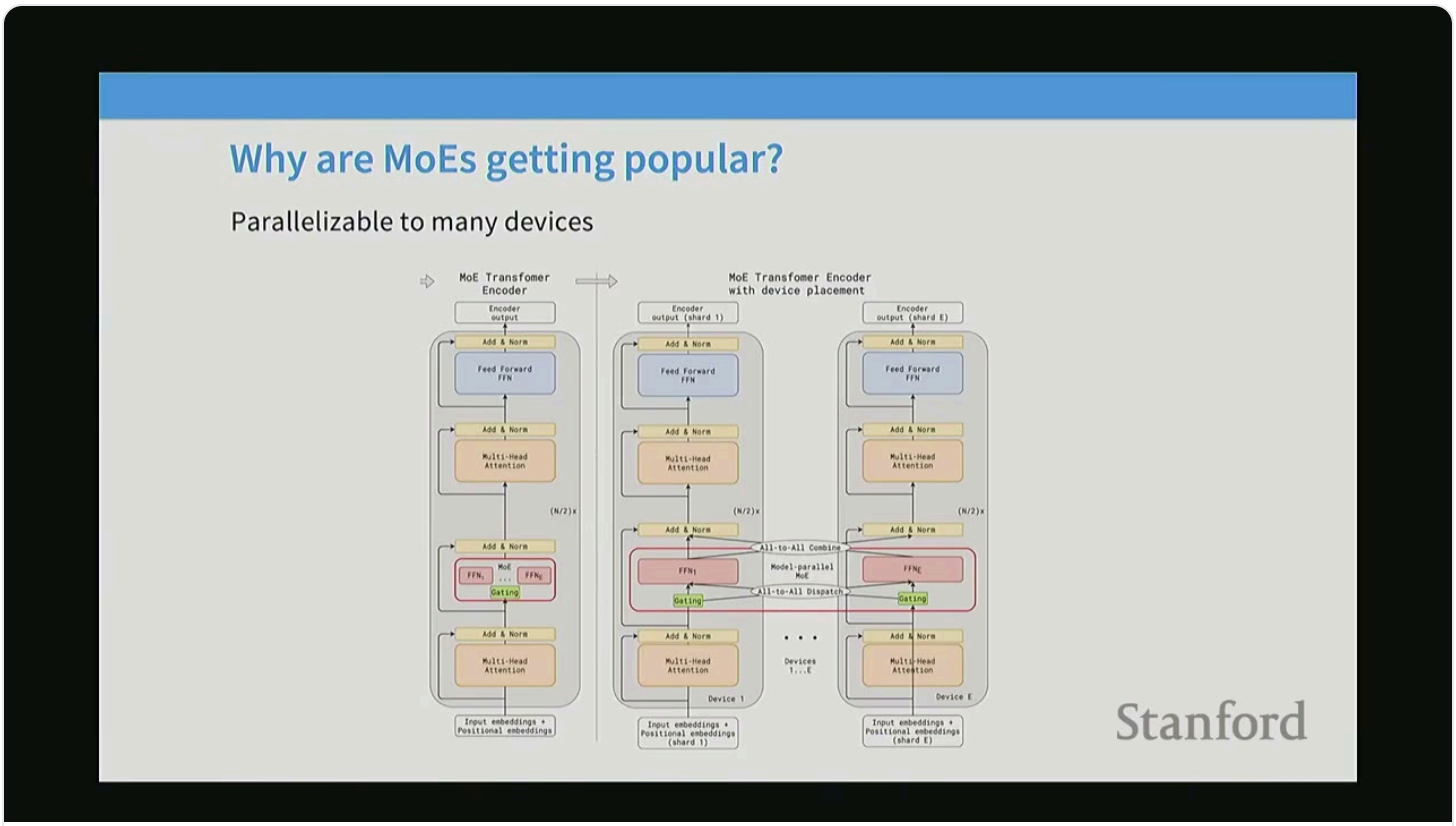
Dense Model vs. Sparse Model (MoE): * **Dense Model (Left):** A standard transformer block contains a self-attention layer followed by a single, large FFN. * **Sparse Model (Right):** In an MoE, the single FFN is replaced by *multiple* FFNs (experts). A **router** (or gating network) is introduced to select a *smaller number* of these experts for each forward pass or inference step.

The key benefit is that you can have **many more parameters** (by having many experts) **without increasing the FLOPs** for a single forward pass, as only a subset of experts are activated. If an expert in an MoE is the same size as the FFN in a dense model, and only one expert is activated, the FLOPs are identical.

[3:34] Why are MoEs getting popular?



5:34 Diagram illustrating MoE Transformer Encoder with device placement for parallelization.



8:29 Bar chart comparing recent MoE results from DeepSeek v3 against other models.

MoEs are gaining popularity because they offer better performance for the same computational cost.

Key advantages: 1. **Same FLOPs, more parameters, better performance:** For a fixed FLOP budget during training, increasing the number of experts in an MoE consistently leads to lower test loss and better perplexity. This means MoEs can learn more effectively by having more parameters to memorize facts about the world. * **Graph (Fedus et al. 2022):** Shows that as the number of experts increases (from 1e to 256e), the test loss decreases significantly for the same FLOPs. * **Graph (Fedus et al. 2022):** Shows that models with more experts (e.g., Switch-Base 128e) achieve better perplexity faster during training compared to models with fewer experts or dense models (T5-Base). This indicates faster convergence and better final performance. * **Graph (AI2 OLMo):** Confirms these findings, showing that MoEs (pink line) achieve lower training and validation loss much faster than dense models (teal line) for the same training time.

2. **Faster to train MoEs:** Due to sparse activation, MoEs can achieve a given performance level with significantly less training time compared to dense models.
    - **Graph (Fedus et al. 2022):** Illustrates a 7x speedup in training time for MoEs to reach the same perplexity as a dense model.
    - **Graph (AI2 OLMo):** Shows that MoEs achieve 3x less FLOPs or 2x faster training time for similar performance.
3. **Highly competitive vs. dense equivalents:** MoEs consistently outperform dense models when comparing performance against "activated parameters" (parameters actually used in computation for a given inference step).
    - **Graph (DeepSeek V2 paper):** Plots performance (MMLU) against activated parameters (billions). MoE models like DeepSeek V2, Mistral 8x22B, and Grok-1 achieve significantly higher MMLU scores for a given number of activated parameters compared to dense Llama models. This highlights the efficiency of MoEs in leveraging parameters for performance.
4. **Parallelizable to many devices:** MoEs offer a natural axis of parallelism, which is crucial for scaling up large models.
    - **Diagram (MoE Transformer Encoder vs. Encoder with Device Placement):**

- In a dense transformer, the FFN is a single block.
- In an MoE, the FFN is split into multiple experts.
- Each expert can be placed on a different computational device (e.g., GPU). The router then directs tokens to the appropriate device, allowing for efficient distribution of computation. This is called **expert parallelism**. This makes MoEs particularly attractive for very large-scale models that require multi-node training.

[9:03] MoE Results from the West and China

MoE development has seen significant contributions from both Western and Chinese research groups. * While Google (a Western company) has been a pioneer in MoE research, many early open-source breakthroughs came from China. * **Qwen and DeepSeek** (Chinese companies) were early adopters and contributors to large-scale, well-documented MoE models. * **Qwen 1.5** was one of the first models to demonstrate a "clever trick" to **upcycle** a dense model into an MoE, showing significant gains in compute efficiency. * **DeepSeek** papers provided foundational work, carefully comparing dense, naive MoE, and switch-routed MoE architectures, consistently showing performance improvements for a fixed FLOP budget. * More recently, Western open-source groups like **Mistral, Grok, and Llama 4** have also adopted MoE architectures, confirming the trend. * **DeepSeek V3** is presented as a culmination of this work, demonstrating remarkable performance. The speaker notes that architecturally, DeepSeek V3 is not drastically different from earlier DeepSeek MoE versions, suggesting that the core engineering principles were established early on.

[2:16] Why haven't MoEs been more popular?

Despite their advantages, MoEs have historically been less prevalent due to several challenges:

1. **Infrastructure Complexity:**
    - MoEs are complex to implement, especially when leveraging multiple accelerators (GPUs/TPUs).
    - Their main advantage (sparsity) comes with the additional complexity of managing sparse data.
    - Training typically involves data parallelism, where different machines get different slices of the training/inference data.

- For sparse models, data needs to be routed to specific machines hosting the activated experts, which requires sophisticated systems for efficient high-throughput serving.
- The biggest benefits of MoEs are realized in multi-node training environments where model splitting is already necessary.

2. **Training Objectives are Heuristic and Unstable:**

    - The decision of which expert to route a token to is a **discrete, non-differentiable** choice. This makes direct gradient-based optimization difficult.
    - Training objectives for MoEs are often heuristic (e.g., using proxy losses) and can lead to unstable training, making them harder to optimize compared to dense models.

[3:50] What MoEs generally look like

The typical MoE architecture involves replacing the FFNs in a transformer block with an MoE layer.

- **Typical MoE (FFN-based):**

    - A standard transformer block has self-attention followed by an FFN.
    - In an MoE, the FFN is replaced by multiple FFNs (experts) and a router.
    - The router selects a subset of experts for each token.
    - This is the most common and successful approach.

- **Less Common MoE (Attention-based):**

    - It's theoretically possible to apply MoE to attention heads, where a router would select specific attention heads for different tokens.
    - Some research has explored this, but it's generally found to be even more unstable and difficult to train consistently. Major model releases rarely use attention-based MoEs.

[4:45] MoE - what varies?

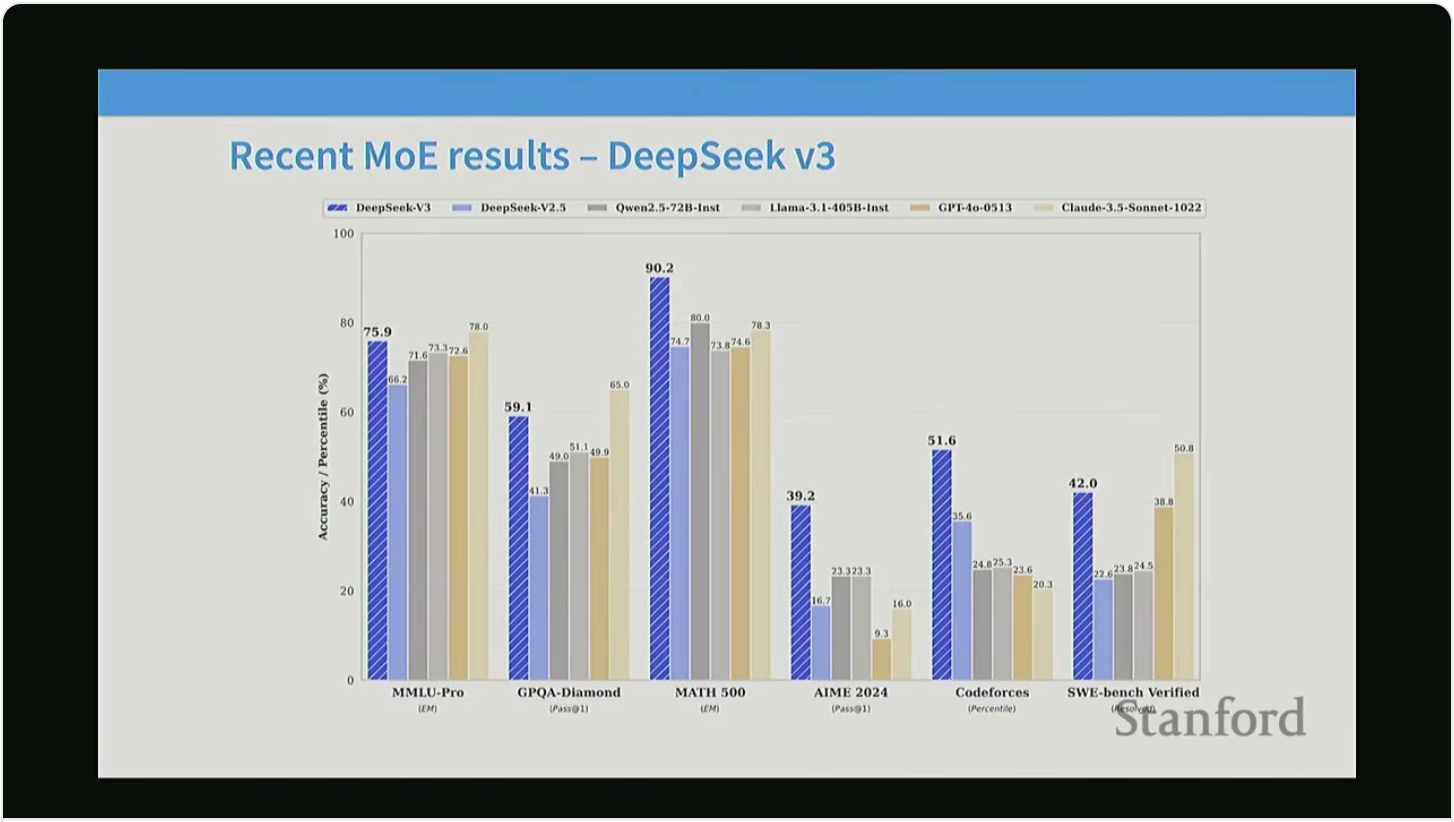
When designing an MoE, several key choices need to be made:

1. **Routing Function:** How are tokens assigned to experts?
2. **Expert Sizes:** How many experts are there, and how large is each expert?

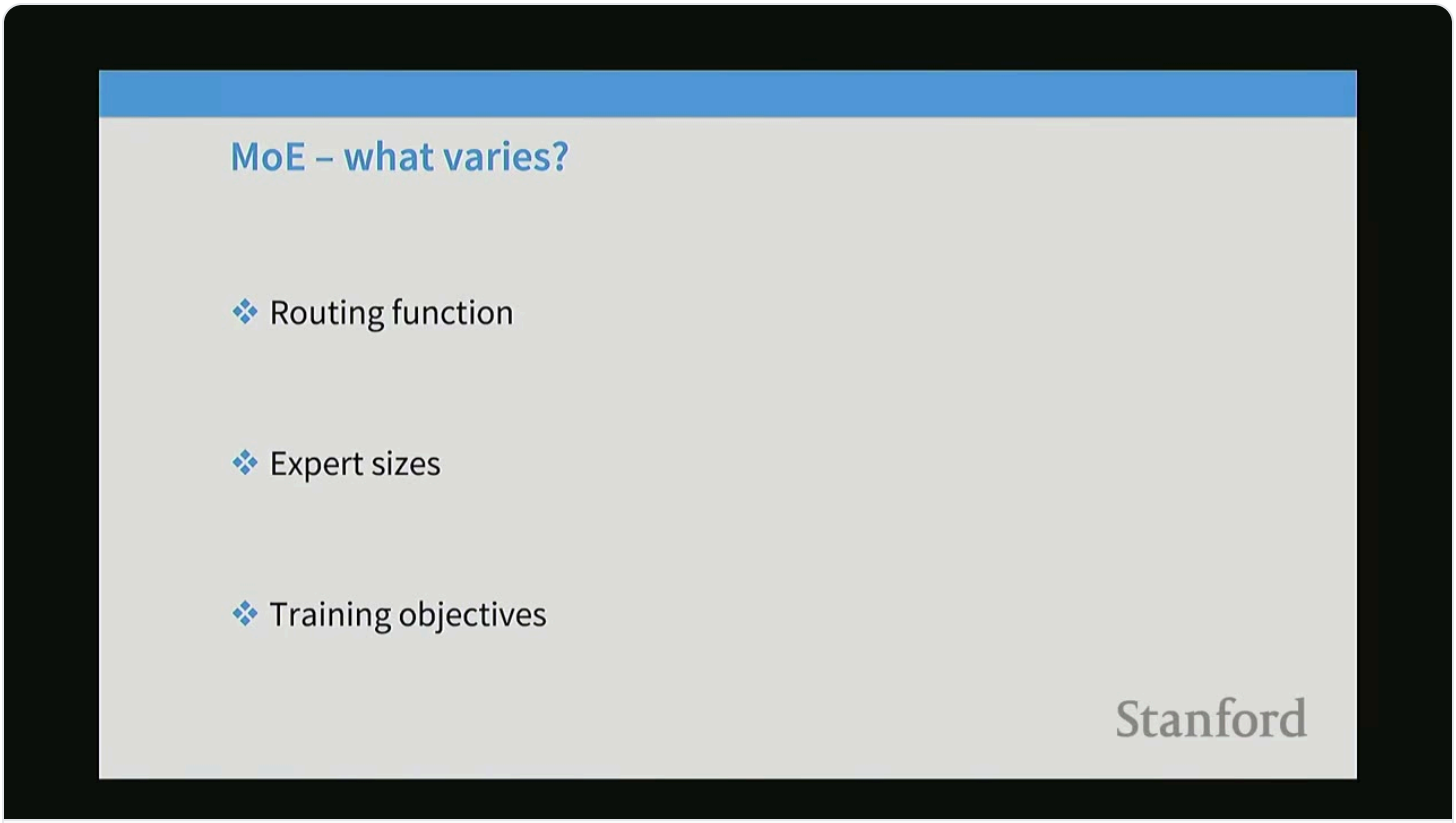
3. **Training Objectives:** How is the router trained, especially given the non-differentiable routing decisions?

These design questions define the vast design space of MoEs.

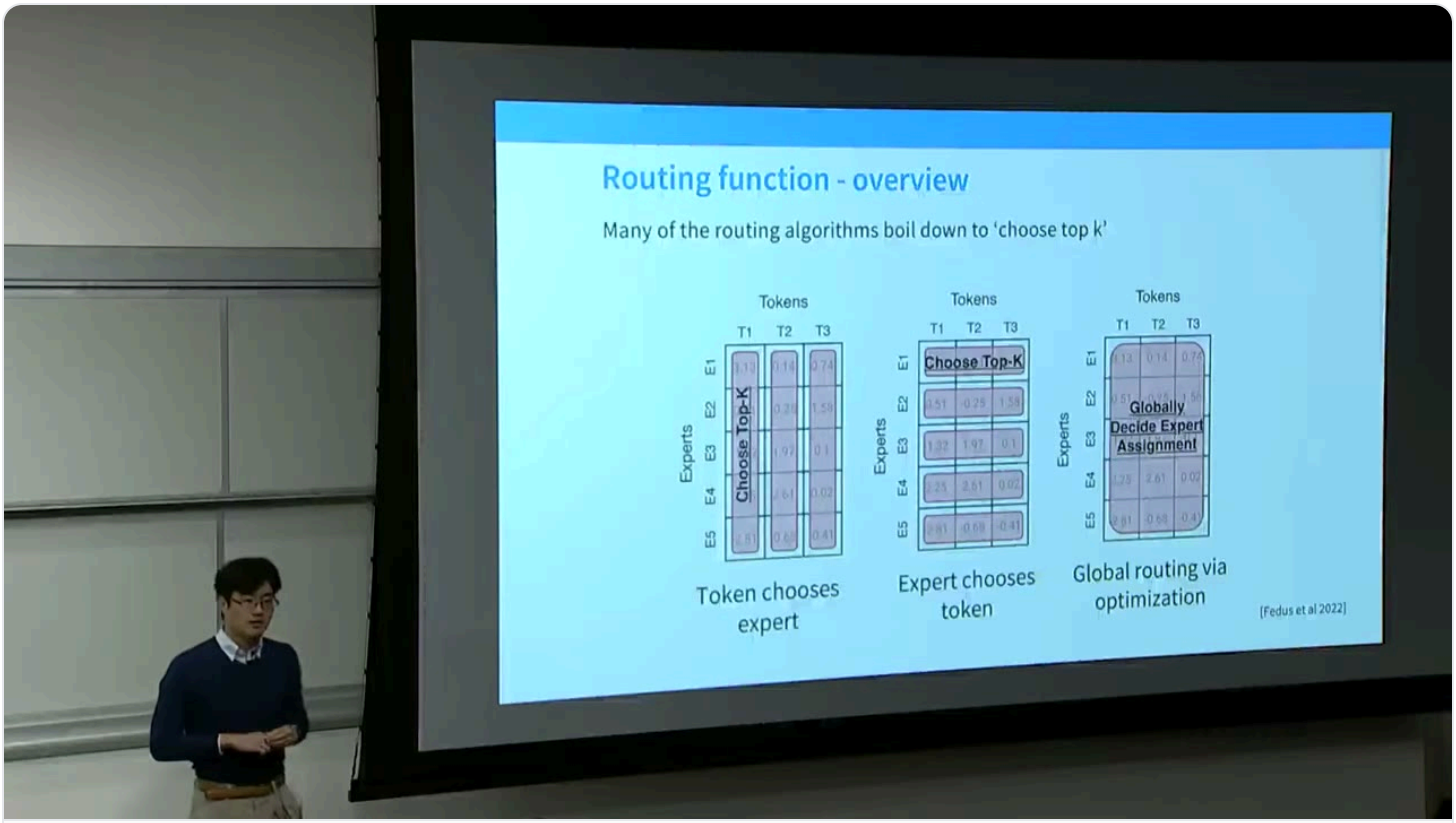
[5:34] Routing Function - Overview



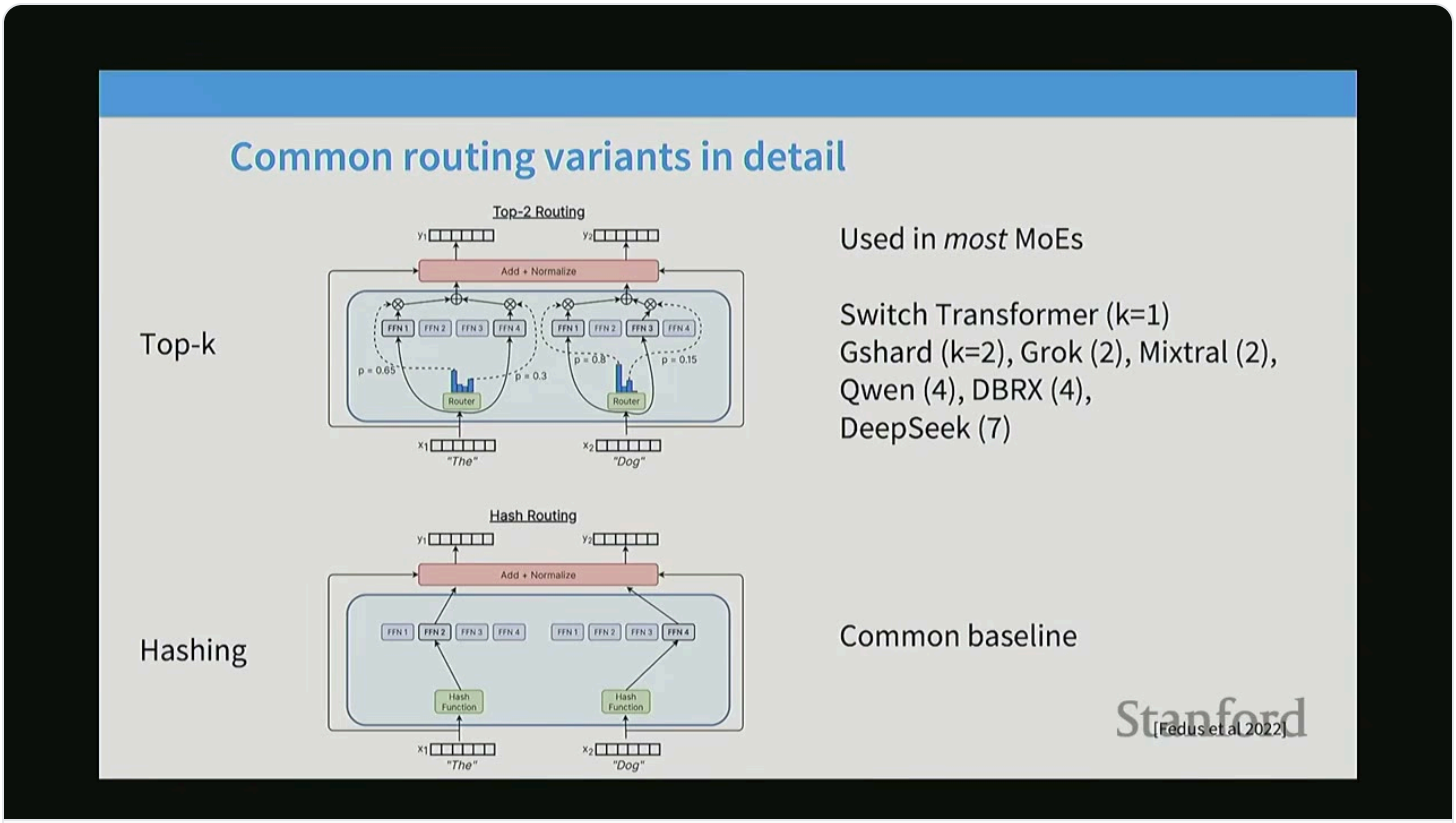
11:39 Slide listing routing function, expert sizes, and training objectives as MoE variables.



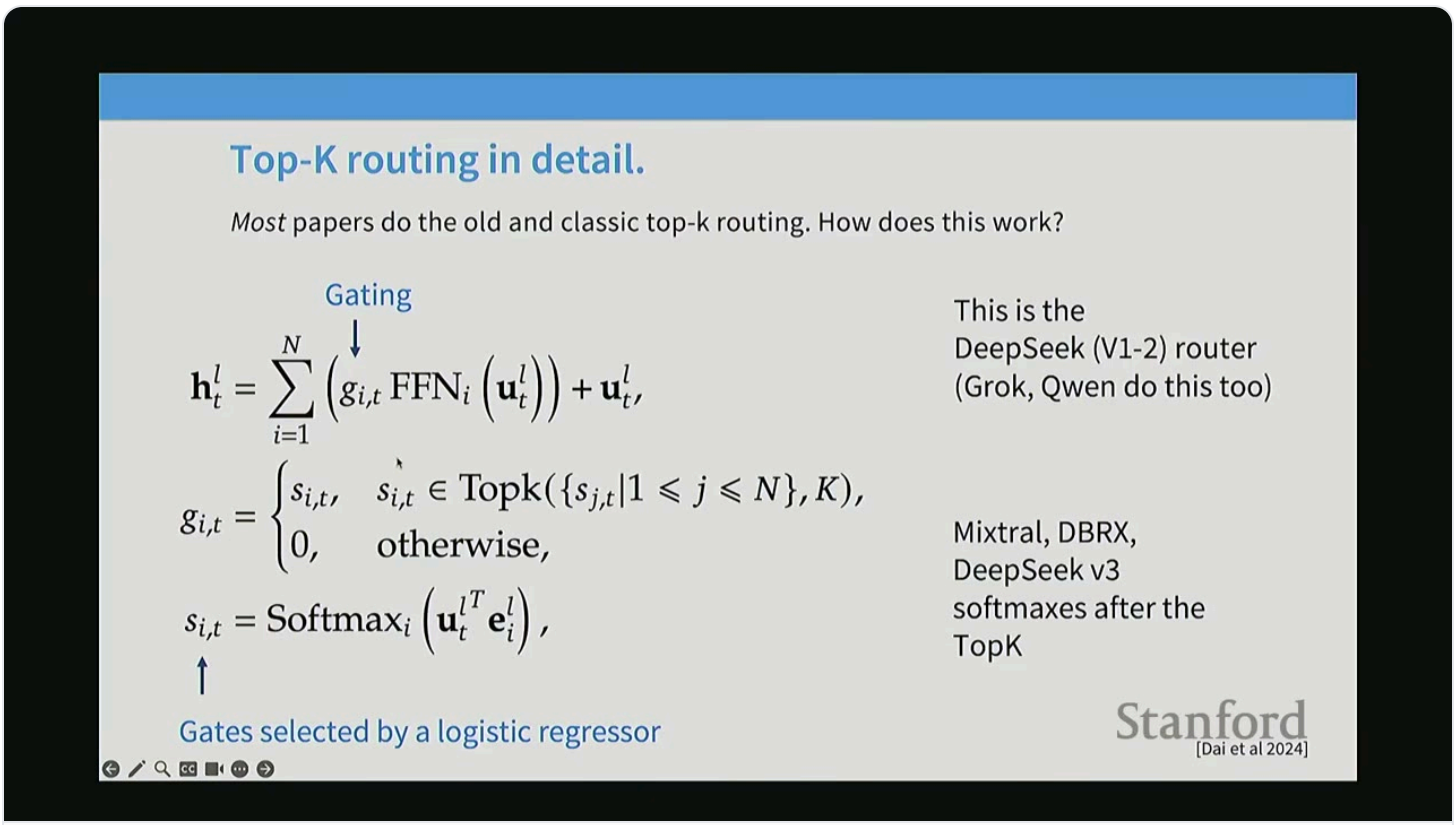
15:01 Speaker presenting a slide on routing function overview, showing token-expert assignment methods.



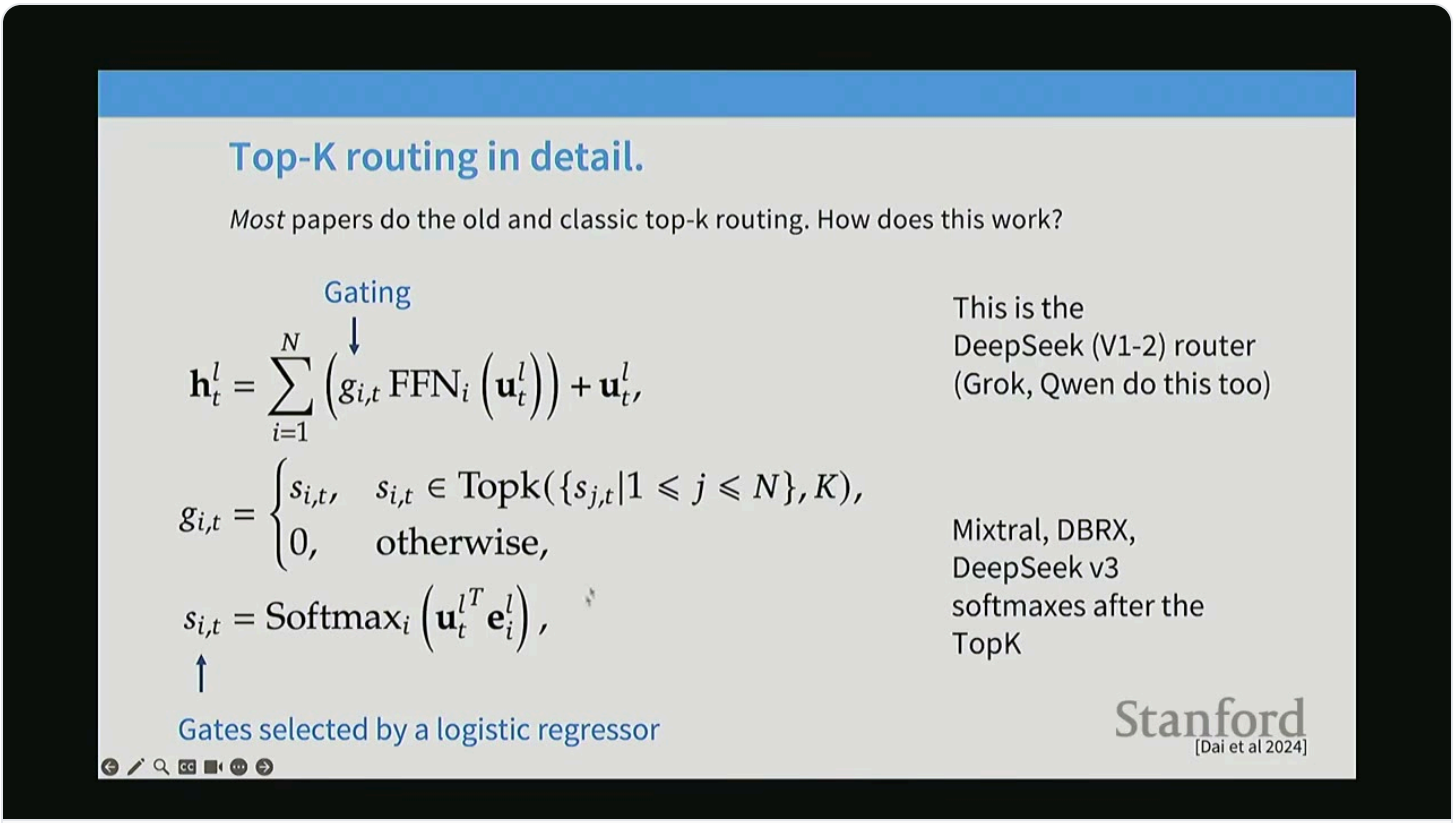
19:19 Diagram detailing Top-k and Hashing routing variants, with examples of models using them.



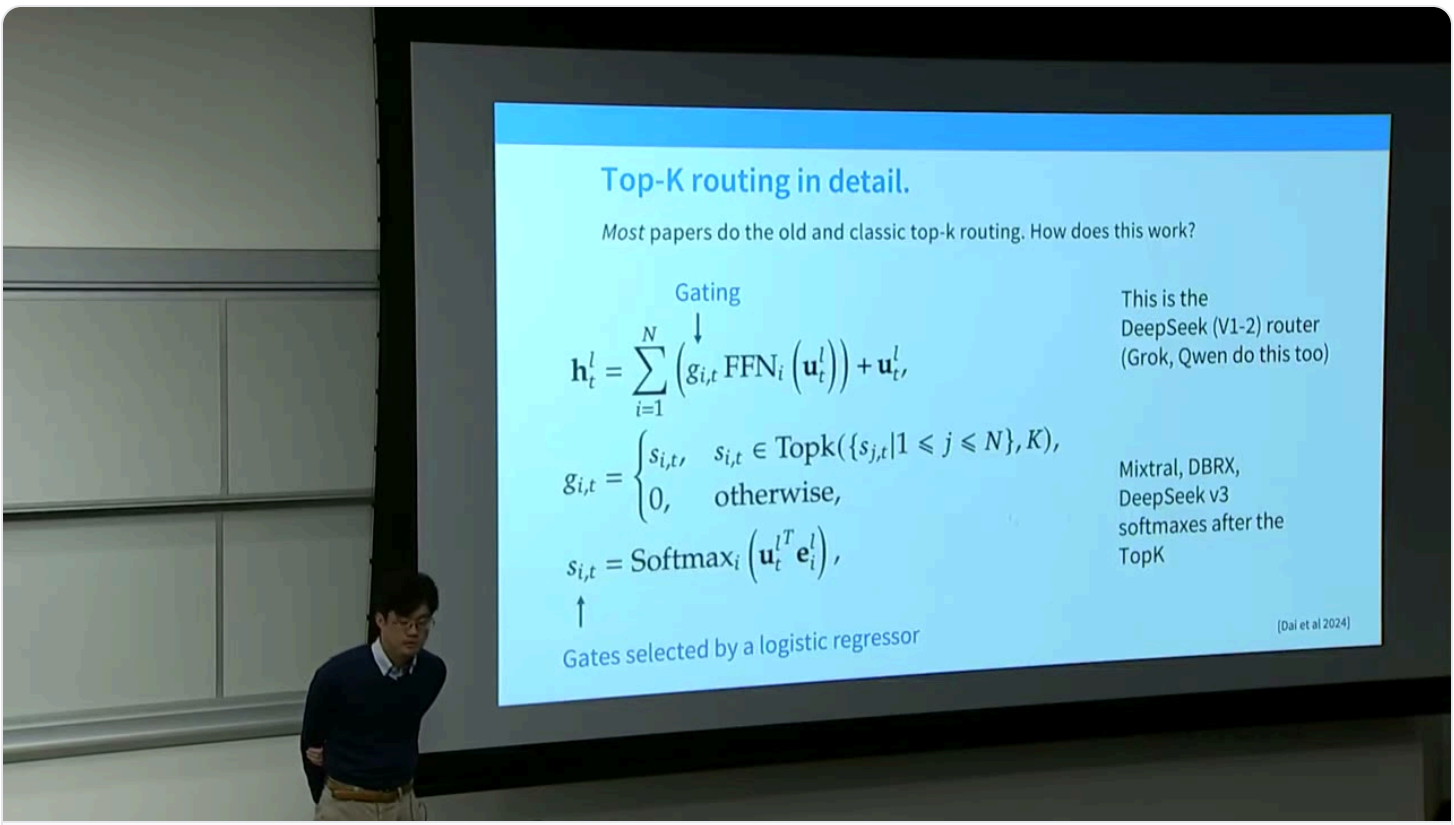
22:10 Equation for Top-K routing in detail, showing gating and softmax functions.



25:33 Equation for Top-K routing in detail, showing gating and softmax functions.



26:56 Speaker presenting a slide on Top-K routing in detail, showing gating and softmax functions.



28:30 Speaker presenting a slide on Top-K routing in detail, showing gating and softmax functions.

The core of an MoE is how it matches tokens to experts. Not all experts process every token; this is the essence of sparse routing.

Three main types of routing algorithms:

1. **Token Chooses Expert (Token Choice):**
    - Each token has a preference score for each expert.
    - The router selects the **top-K** experts with the highest scores for each token.
    - **Benefit:** Each token is processed by the most relevant experts.
    - **Drawback:** Can lead to imbalanced expert utilization if some experts are consistently preferred.
2. **Expert Chooses Token (Expert Choice):**
    - Each expert has a preference score for each token.
    - The router selects the **top-K** tokens with the highest scores for each expert.
    - **Benefit:** Ensures balanced utilization of experts, as each expert processes a fixed number of tokens.
    - **Drawback:** A token might not be processed by its "best" expert if that expert is already "full."
3. **Global Routing via Optimization:**
    - A more complex approach that involves solving an optimization problem (e.g., linear assignment) to globally decide expert assignments, aiming to balance both token relevance and expert utilization.
    - **Benefit:** Theoretically optimal balance.
    - **Drawback:** High computational cost and complexity.

Practical Choice: * Almost all modern MoEs use **Token Choice Top-K routing**. * Early MoE research explored various routing strategies, but recent large-scale models have converged to this method. * **Ablations (OLMo):** Show that Token Choice (TC) routing generally leads to better performance (lower validation loss, higher MMLU Var) compared to Expert Choice (EC) routing.

Q&A on Routing: * **How does a token know which expert is best?** The router, which is a lightweight neural network, learns to predict scores for each expert based on the token's hidden state. It's typically a simple linear projection followed by a non-linearity (e.g., sigmoid or softmax). * **What is K?** K is a hyperparameter representing the number of experts activated per token. Early

MoE papers suggested K > 1 (e.g., K=2) to encourage exploration (avoid always picking the "best" arm and missing out on potentially better ones). K=2 remains a common choice. * **What happens if K > 1?** If K > 1, multiple experts process the token, and their outputs are combined (e.g., summed or weighted average) before being passed to the next layer.

[1:16:00] Common Routing Variants in Detail

The lecture details the two most common routing variants:

1. **Top-K Routing (Token Choice Top-K):**
    - **Mechanism:** For each token, the router computes scores for all experts. The top-K scoring experts are selected. Their outputs are combined (e.g., weighted sum based on router scores).
    - **Usage:** Used in most modern MoEs, including Switch Transformer (K=1), GShard (K=2), Grok (K=2), Mixtral (2), Qwen (4), DBRX (7), and DeepSeek (7).
    - **Diagram:** Shows an input `x` going through a `Router` that outputs probabilities `p1, p2, p3, p4`. If K=2, the top two (e.g., p1=0.6, p3=0.3) are selected, and their corresponding FFNs (Expert-1, Expert-3) are activated. Their outputs are then combined.
2. **Hashing:**
    - **Mechanism:** A simple hash function maps each token to a specific expert. No learned router is involved.
    - **Usage:** Serves as a common baseline. Surprisingly, even this non-semantic routing can yield performance improvements over dense models.
    - **Intuition for Hashing's Effectiveness:** Even random assignment creates specialized subsets of data for each expert. If the data distribution is Zipfian (like natural language), some experts might inadvertently specialize in very frequent tokens/patterns, leading to some form of implicit specialization.

[2:50] Other Routing Methods

- **Reinforcement Learning (RL) to learn routes:**
    - **Idea:** Treat routing as a discrete decision problem and use RL to learn optimal gating policies. This is conceptually appealing as RL excels at discrete decision-making.

    - **Usage:** Used in some of the earliest MoE work (e.g., Bengio 2013).
    - **Drawback:** Not commonly used now due to high computational cost and existing stability issues with MoEs. RL adds significant complexity without a clear performance win.

- **BASE Routing (Solving a Matching Problem):**

    - **Idea:** Formulate routing as a linear assignment problem or optimal transport problem, aiming for balanced and efficient assignments.
    - **Usage:** Explored in some recent papers (e.g., Clark '22).
    - **Drawback:** Computationally expensive, often outweighing the benefits in practice.

[3:45] Top-K Routing in Detail (Mathematical Formulation)

This section provides the mathematical details of the standard Top-K routing used in most MoEs.

$$h'_t = \sum_{i=1}^{N} g_{i,t} \text{FFN}_i(u_t^L) + u_t^L$$

Where: * u_t^L: Input residual stream for token t at layer L. * FFN_i: The i-th expert (feed-forward network). * $g_{i,t}$: The gating weight for expert i and token t. * N: Total number of experts. * The sum is over the activated experts.

The gating weights $g_{i,t}$ are determined as follows:

$$g_{i,t} = \begin{cases} s_{i,t} & \text{if } s_{j,t} \in \text{TopK}(\{s_{j,t} | 1 \le j \le N\}, K) \\ 0 & \text{otherwise} \end{cases}$$

Where: * $s_{i,t}$: The raw score for expert i and token t. * $\text{TopK}(\cdot, K)$: A function that selects the top K values. * This means only the top K experts receive non-zero gating weights.

The raw scores $s_{i,t}$ are typically computed using a logistic regressor (a simple linear layer followed by a softmax):

$$s_{i,t} = \text{Softmax}(u_t^L e_i^T)$$

Where: * e_i: A learned vector (expert embedding) representing expert i. This vector is *not* tied to the FFN parameters.

Key points: * **Gating:** The process of selecting experts and weighting their outputs. * **Softmax:** The softmax function normalizes the raw scores $s_{i,t}$ to sum to 1, effectively turning them into probabilities. * **Top-K Selection:** After computing softmax scores, only the top K experts are chosen. The gating weights for non-selected experts are set to 0. * **System Efficiency:** The Top-K selection is crucial for achieving sparse activation during both training and inference. Without it, activating all experts would incur the full FLOPs cost of a dense model with N times the parameters. * **Non-differentiability:** The Top-K operation is non-differentiable, making direct gradient-based optimization challenging.

Q&A on Top-K Routing: * **Softmax then Top-K:** The softmax is applied first to get normalized scores, then Top-K selects the highest. The resulting $g_{i,t}$ values (after zeroing out non-top-K experts) no longer sum to 1. This is acceptable because the next layers (e.g., LayerNorm) can adjust the scale. * **E-vectors:** The e_i vectors are learned parameters of the router, separate from the FFN parameters. They represent what kind of input each expert is "looking for." * **Learning Challenge:** The non-differentiable nature of Top-K makes learning the router's parameters (the e_i vectors) difficult. The gradients for non-selected experts are zero.

[2:00] Recent variations from DeepSeek and other Chinese LLMs

DeepSeek introduced several architectural innovations that were quickly adopted by other MoE models.

1. **Fine-grained Experts:**
    - **Idea:** Instead of having a few large experts, have many smaller experts. This allows for a massive increase in the total number of parameters without increasing FLOPs (since each expert is smaller).
    - **Mechanism:** If a standard FFN multiplies the hidden dimension by 4 (e.g., $H \to 4H$), fine-grained experts might multiply by 2 (e.g., $H \to 2H$) and then have twice as many experts. This can be extended to $H \to H/X$ for X times more experts.
    - **Benefit:** More parameters for the same FLOPs, leading to better performance.
    - **Diagram (b):** Shows a "fine-grained expert segmentation" where the FFNs are smaller (e.g., 2W instead of 4W), allowing for more experts (e.g., N experts instead of N/2).
2. **Shared Experts:**

- **Idea:** Include a few experts that are always active for all tokens, regardless of the router's decision.
- **Benefit:** These shared experts can learn common, universally applicable patterns or processing tasks, reducing the burden on specialized experts and potentially improving overall efficiency.
- **Diagram (c):** Shows "shared expert isolation" where some experts (green) are always active, while others (blue) are routed.

Ablations from DeepSeek paper: * **Graph:** Shows normalized performance across various benchmarks (HellaSwag, PIQA, ARC-Easy, ARC-Challenge, TriviaQA, NaturalQuestions). * **Findings:** * Adding **one shared expert** (orange bar) provides a significant boost on some tasks. * Adding **fine-grained experts** (green and yellow bars) provides further boosts. * Combining these innovations (e.g., 1 shared expert + fine-grained experts) consistently leads to the best performance across most benchmarks. * **Conclusion:** More experts (especially fine-grained) and shared experts generally help.

Ablations from OLMoE (AI2): * **Graphs:** Corroborate DeepSeek's findings. * **Fine-grained experts:** Increasing the number of fine-grained experts (from 8 to 64) leads to clear improvements in training loss, validation loss, and MMLU Var. * **Shared experts:** In the OLMo setup, shared experts (purple vs. teal) did not show significant gains, suggesting this innovation might be more context-dependent or require specific tuning. OLMo ultimately opted for no shared experts.

[7:07] Expert Routing Setups for Recent MoEs

A table summarizes the routing configurations of various prominent MoE models:

Model	Routed	Active	Shared	Fine-grained ratio
GShard	2048	2	0	0
Switch Transformer	64	1	0	0
ST-MoE	64	2	0	0
Mixtral	8	2	0	0
DBRX	16	2	0	0

Model	Routed	Active	Shared	Fine-grained ratio
Grok	8	2	0	0
DeepSeek v1	64	6	2	1/4
Qwen 1.5	64	4	1	1/14
DeepSeek v3	256	8	2	1/8
OLMoE	32	8	0	1/8
MiniMax	64	8	0	~1/4
Llama 4 (maverick)	128	1	1/2	0

Observations: * **Early Google models (GShard, Switch Transformer, ST-MoE):** Explored a wide range of "routed" experts (up to 2048), often with K=1 or K=2 active experts, and no explicit shared experts. * **Mid-range models (Mixtral, DBRX, Grok):** Tend to use fewer total experts (8-16) with K=2 active, and no shared experts. * **DeepSeek and Qwen:** Pioneered the use of **fine-grained experts** (indicated by ratios like 1/4, 1/14, 1/8) and **shared experts**. * DeepSeek v1: 64 routed, 6 active, 2 shared, 1/4 fine-grained ratio. * Qwen 1.5: 64 routed, 4 active, 1 shared, 1/14 fine-grained ratio. * DeepSeek v3: 256 routed, 8 active, 2 shared, 1/8 fine-grained ratio. * **Recent models (OLMoE, MiniMax, Llama 4):** Continue to use fine-grained experts and often shared experts, with varying numbers of total and active experts. * Llama 4 is unique in having a 1/2 shared expert, implying the shared expert is half the size of a standard FFN.

Key takeaway: The trend is towards using **fine-grained experts** and often **shared experts** to maximize parameter count and performance for a given FLOP budget. For very large models like DeepSeek and Llama 4, the number of total experts can be very high (hundreds).

[1:50:00] How do we train MoEs?

The major challenge in training MoEs is achieving **training-time efficiency** while dealing with **non-differentiable sparse gating decisions**.

Potential Solutions: 1. **Reinforcement Learning (RL) to optimize gating policies:** * **Idea:** Treat the router as an agent making discrete decisions and use RL algorithms (e.g., REINFORCE) to learn optimal gating policies. * **Pros:** Principled approach for non-differentiable decisions. * **Cons:**

In practice, RL for MoEs hasn't shown significant improvements over simpler methods. High gradient variance and complexity make it finicky and not widely used at scale.

2. **Stochastic perturbations:**

    - **Idea:** Introduce noise or randomness into the routing decisions to make them "soft" and encourage exploration.
    - **Mechanism (Shazeer et al. 2017):** Instead of directly computing $s_{i,t} = \text{Softmax}(u_t^L e_i^T)$, they add Gaussian noise to the logits: $H(x)_i = (x \cdot W_g)_i + \text{StandardNormal}() \cdot \text{Softplus}((x \cdot W_{noise})_i)$. The W_{noise} matrix is learned to control the amount of noise.
    - **Benefits:** Naturally leads to experts that are a bit more robust (less specialized). The softmax (after Top-K) helps the model learn to rank experts.
    - **Drawbacks:** Stochasticity can lead to less specialization and loss of efficiency. This approach was tried in early Google papers but often abandoned for heuristic loss-based methods.

3. **Heuristic "balancing" losses:**

    - **Idea:** Add auxiliary loss terms to the training objective that encourage the router to distribute tokens evenly among experts. This prevents "dead" experts (experts that are never chosen) and ensures all parameters are utilized.
    - **Mechanism (Switch Transformer, Fedus et al. 2022):** The loss function is augmented with a term that penalizes imbalanced expert utilization.
        - $L_{aux} = \alpha \sum_{i=1}^{N} f_i P_i$
        - f_i: Fraction of tokens dispatched to expert i (actual utilization).
        - P_i: Fraction of router probability allocated to expert i (intended utilization).
        - The derivative with respect to P_i shows that experts with higher P_i (more frequently used) are downweighted more strongly, encouraging the router to explore other experts.
    - **Types of Balancing:**
        - **Per-expert balancing:** Ensures each expert receives a roughly equal number of tokens per batch.
        - **Per-device balancing:** Aggregates expert utilization by device, ensuring each device (e.g., GPU) receives a balanced workload. This is crucial for system efficiency.
    - **DeepSeek V3 Variation (Per-expert biases):**

- DeepSeek V3 replaces the per-expert balancing loss with a simpler mechanism: **per-expert biases** (b_i).
- The router's scores are adjusted by these biases: $g'_{i,t} = s_{i,t} + b_i$.
- The biases b_i are learned through a simple online gradient scheme: if an expert isn't getting enough tokens, its b_i is increased; if it's getting too many, its b_i is decreased.
- This is called "auxiliary loss free balancing" because it avoids adding a complex auxiliary loss term.
- However, DeepSeek V3 also reintroduces a "complementary sequence-wise auxiliary loss" (which is essentially the balancing loss) to handle imbalances at the sequence level, not just the batch level. This suggests that even with clever bias adjustments, explicit balancing losses are often still necessary.

[5:54:00] Issues with MoEs - Stability

MoEs are known to be difficult to train and fine-tune, often exhibiting instability.

- **Problem:** MoEs can "blow up" during training, especially when using lower precision (e.g., bfloat16). This is often due to numerical instability in the softmax operation, where large logits can lead to overflow or underflow, causing incorrect outputs and unstable gradients.
- **Solution (Zoph et al. 2022):**
  - **Use Float32 for the expert router:** Performing router computations (especially softmax) in higher precision (Float32) can significantly improve stability, even if the rest of the model uses lower precision (e.g., bfloat16).
  - **Auxiliary Z-loss:** Adding an auxiliary Z-loss (log-sum-exp of logits squared) to the router's objective helps keep the logits bounded, preventing numerical instability.
  - **Graph (Router Z-loss stability):** Shows that removing the Z-loss from the router leads to large, unstable spikes in training and validation loss, indicating severe instability. The Z-loss effectively "tames" the router's logits.

[1:07:00] Issues with MoEs - Fine-tuning

MoEs can also be challenging to fine-tune, especially on smaller datasets.

- **Problem:** Sparse MoEs, with their massive parameter counts, are prone to **overfitting** when fine-tuned on smaller, downstream tasks. This leads to a large gap between training and validation performance.
- **Graph (SuperGLUE CB Task):** Shows that sparse MoEs (blue and orange lines) exhibit a much larger train-test gap (overfitting) compared to dense models (green and red lines) when fine-tuned on a small task.
- **Solutions:**
    1. **Hybrid Architectures:** Architect MoEs such that not every layer is an MoE layer. For example, alternate dense layers with MoE layers. When fine-tuning, only fine-tune the dense layers, which behave more stably.
    2. **Use lots of data:** If overfitting is the problem, the solution is to use more data. DeepSeek MoE, for instance, used 1.4 million training examples for fine-tuning, mitigating overfitting concerns.

[1:08:25] Other training methods - Upcycling

Upcycling is a clever trick to initialize an MoE from a pre-trained dense model, saving significant training costs.

- **Idea:** Instead of training an MoE from scratch, leverage an existing, well-trained dense model.
- **Mechanism:**
    1. Take a pre-trained dense model.
    2. Duplicate its FFN weights to create multiple experts.
    3. Perturb the copied weights slightly to encourage differentiation among experts.
    4. Initialize the router from scratch.
    5. Continue training the resulting MoE.
- **Benefit:** This method is very cost-effective, allowing for the creation of high-performance MoEs without the immense computational cost of training from scratch. It effectively "transforms" a dense model into an MoE.
- **Graph (Upcycling performance):** Shows that upcycled MoEs (red dots) achieve significantly better performance (validation token accuracy) for the same pretraining time compared to dense models (blue dots). This is particularly true for larger models.
- **Example (MiniCPM):** MiniCPM (a Chinese open LLM) successfully used upcycling to create MoEs that outperformed larger dense models, demonstrating the effectiveness of this

technique.
- **Example (Qwen MoE):** Qwen also used upcycling to build their MoE models, achieving strong performance relative to smaller models.

[1:10:10] DeepSeek MoE v1-v2-v3 Architecture Walkthrough

This section provides a detailed walkthrough of the evolution of DeepSeek's MoE architecture, highlighting key changes and innovations.

DeepSeek MoE v1 (16B - 2.8B active): * **Architecture:** Shared + Fine-grained (64/4) experts. This means 2 shared experts and 64 fine-grained experts, where each fine-grained expert is 1/4 the size of a standard FFN. * **Routing:** Standard Top-K routing (softmax then Top-K). * **Balancing:** Auxiliary loss balancing (expert + device level). * **Key takeaway:** This early version already incorporated the core ideas of fine-grained and shared experts, along with balancing losses.

DeepSeek MoE v2 (236B - 21B active): * **Architecture:** Shared + Fine-grained (160/10) experts. This means 10 shared experts and 160 fine-grained experts, with 6 active experts. * **New Things:** * **Top-M device routing:** To address the communication costs of fine-grained experts, DeepSeek v2 introduced a two-stage routing process. First, the router selects the top M devices. Then, within those M devices, it selects the top K experts. This reduces communication by concentrating token traffic to a smaller set of devices. * **Communication balancing loss:** To balance communication both in and out, they added a communication balancing loss that considers both input and output communication costs, ensuring efficient data flow across devices.

DeepSeek MoE v3 (671B - 37B active): * **Architecture:** Shared + Fine-grained (256/8) experts. This means 8 shared experts and 256 fine-grained experts, with 8 active experts. * **New Things:** * **Sigmoid + Softmax + TopM routing:** The routing mechanism was further refined. Instead of a simple softmax, they use a sigmoid function for the initial gating decisions, followed by a softmax and TopM selection. This allows for more flexible gating. * **Aux-loss-free + Seq-wise aux loss:** They moved away from the explicit auxiliary balancing loss, instead relying on the per-expert biases (b_i) learned through an online gradient scheme (as discussed earlier). However, they also added a "sequence-wise auxiliary loss" to ensure balancing at the individual sequence level, addressing potential issues with out-of-distribution sequences at inference time.

[1:16:17] Bonus: What else do you need to make DeepSeek MoE v3?

Beyond the MoE-specific innovations, DeepSeek V3 incorporates other advanced techniques common in modern LLMs.

1. **Multi-head Latent Attention (MLA):**
    - **Basic Idea:** Express the Query (Q), Key (K), and Value (V) as functions of a lower-dimensional "latent" activation.
    - **Mechanism:** Instead of directly generating K and V from the hidden state H_t, DeepSeek V3 first projects H_t into a lower-dimensional "latent" representation C. This C is then cached (KV cache). When K and V are needed, they are up-projected from C.
    - **Benefits:** Reduces the size of the KV cache, which can be a significant memory bottleneck during inference.
    - **Complexity:** This involves an extra matrix multiplication (W^{UK}) to project H_t to C, and another to project C back to K/V. However, this W^{UK} matrix can be merged with the existing Q projection matrix, avoiding an extra FLOPs cost.
    - **RoPE Compatibility:** This MLA approach is compatible with RoPE (Rotary Positional Embeddings), although special care is needed to handle the rotation matrices.
2. **Multi-token Prediction (MTP):**
    - **Idea:** Have small, lightweight models that can predict multiple steps ahead.
    - **Mechanism:** In addition to the main transformer predicting the next token, DeepSeek V3 uses a lightweight, one-layer transformer (MTP Module 2) to predict the token *after* the next token.
    - **Benefit:** This allows the model to "look ahead" during generation, potentially improving coherence and quality.

[1:21:13] MoE Summary

- **MoEs take advantage of sparsity:** Not all inputs need the full model. This is the fundamental efficiency gain.
- **Discrete routing is hard, but Top-K heuristics seem to work:** While theoretically challenging, practical heuristic approaches for routing have proven effective.

- **Lots of empirical evidence now that MoEs work and are cost-effective:** The widespread adoption and strong benchmark results confirm the benefits of MoEs.

Practical Takeaways

- MoEs are the current state-of-the-art for building large, high-performing LLMs due to their parameter efficiency and parallelization capabilities.
- The key to MoE success lies in carefully designed routing functions (Top-K is dominant), fine-grained experts, and effective load balancing strategies.
- Training stability and overfitting are significant challenges, often addressed through techniques like higher precision for routers, Z-loss, and upcycling.
- System-level optimizations, including expert parallelism and communication balancing, are crucial for deploying MoEs at scale.

Open Questions / Things to Remember

- The "expert" in MoE doesn't necessarily imply semantic specialization, but rather a mechanism for sparse activation.
- The non-differentiable nature of Top-K routing is a core challenge, leading to the development of heuristic balancing losses.
- Upcycling dense models to MoEs is a powerful technique for cost-effective initialization.
- DeepSeek V3 showcases a highly engineered MoE, integrating fine-grained experts, shared experts, sophisticated routing (Top-M device routing), and various balancing losses to achieve state-of-the-art performance.

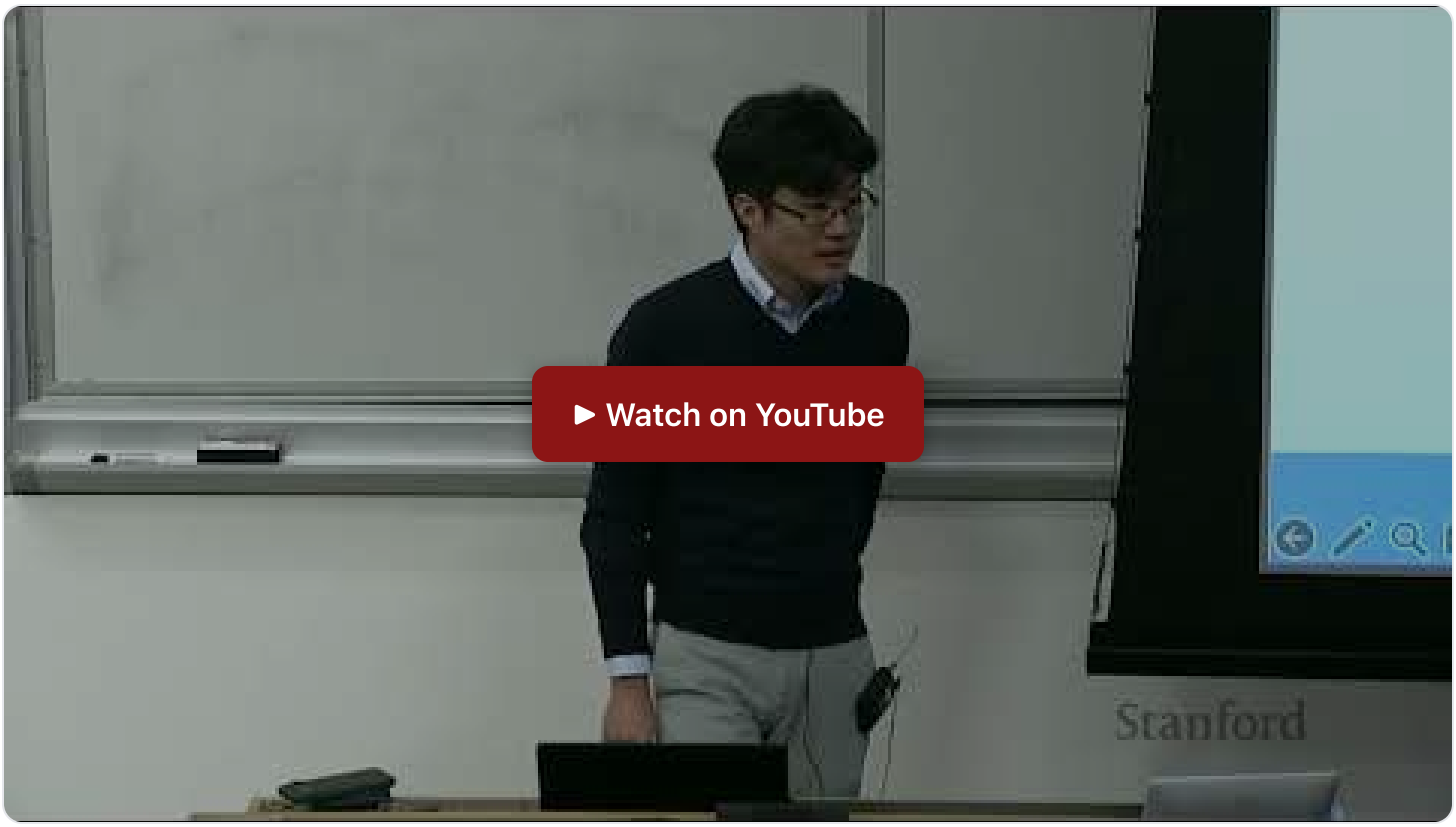
← Lec 03

Lec 05 →

Lecture 05: GPUs

Stanford CS336 · Spring 2025 · 1:14:21 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



TL;DR - GPUs are massively parallel processors optimized for throughput, not latency, by having many simple compute units (SMs) orchestrated by minimal control logic. - Compute (especially matrix multiplication) has scaled much faster than memory bandwidth, making memory access the primary bottleneck for modern ML workloads. - Optimizing GPU performance requires careful management of the memory hierarchy, leveraging fast on-chip memory (shared memory, registers) and minimizing slow global memory (DRAM) access. - Key techniques for optimizing GPU performance include: reducing memory accesses (coalescing, fusion, tiling), trading compute for memory (quantization, recomputation), and aligning data access patterns with hardware specifics (bursts, tile divisibility). - Flash Attention (and Flash Attention 2) dramatically accelerates attention by applying these low-level GPU optimization techniques, particularly online softmax computation and recomputation, to reduce HBM accesses.

Key Concepts - GPU Architecture (SMs, SPs, Tensor Cores) - CPU vs. GPU Design Goals (Latency vs. Throughput) - Memory Hierarchy (Registers, Shared Memory, L1/L2 Cache, Global Memory/HBM) - Execution Model (SIMT, Warps, Blocks, Threads) - Dennard Scaling vs. Parallel Scaling - Roofline Model (Compute-bound vs. Memory-bound) - GPU Optimization Techniques: - Control Divergence - Low Precision Computation (Quantization) - Operator Fusion - Recomputation - Memory Coalescing - Tiling - Flash Attention

---

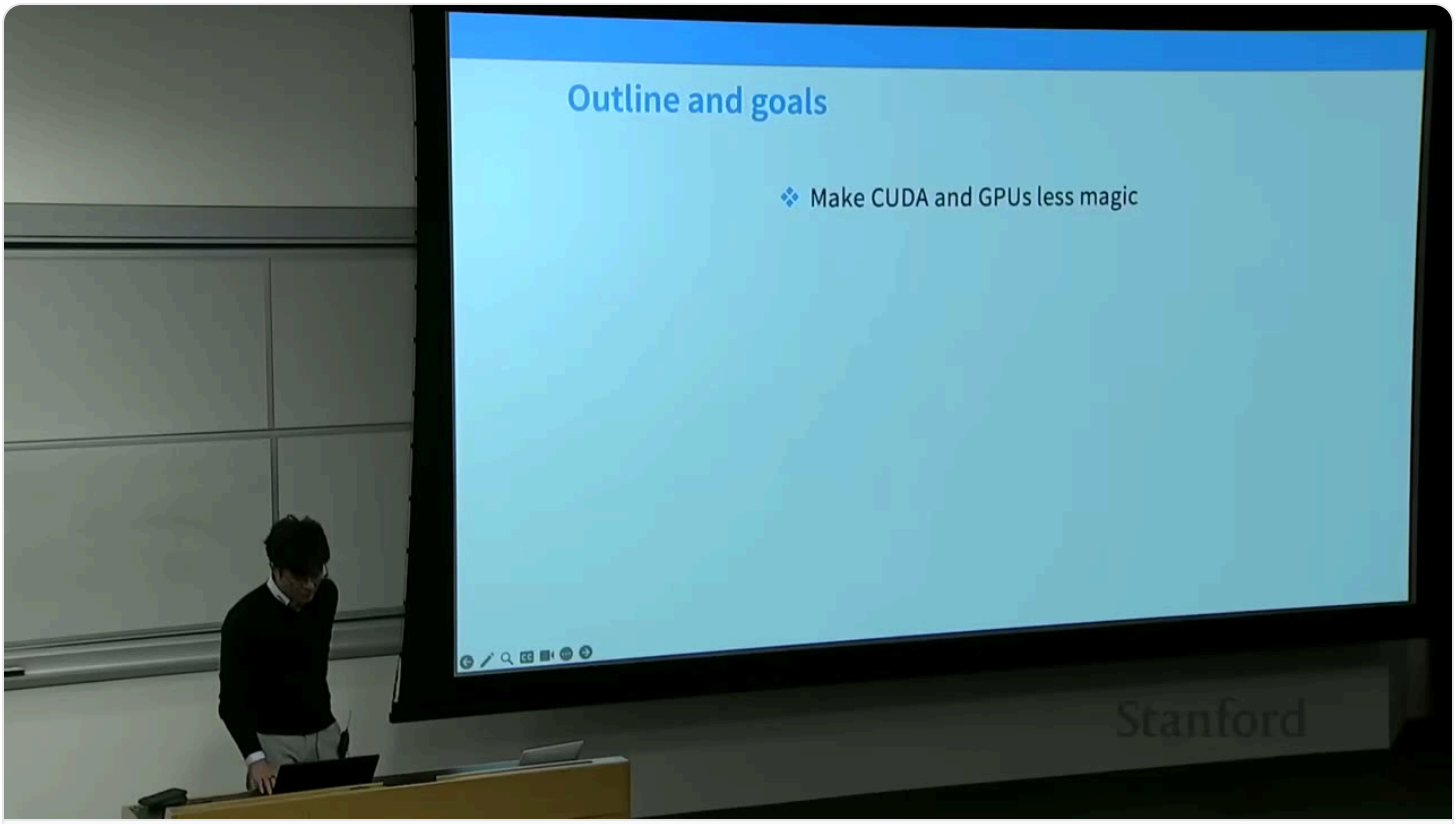
[0:00] Introduction and Course Updates

- Assignment 1 is due tonight. Extensions are available upon request.
- Assignment 2 will be released soon, focusing on implementing parts of Flash Attention 2.

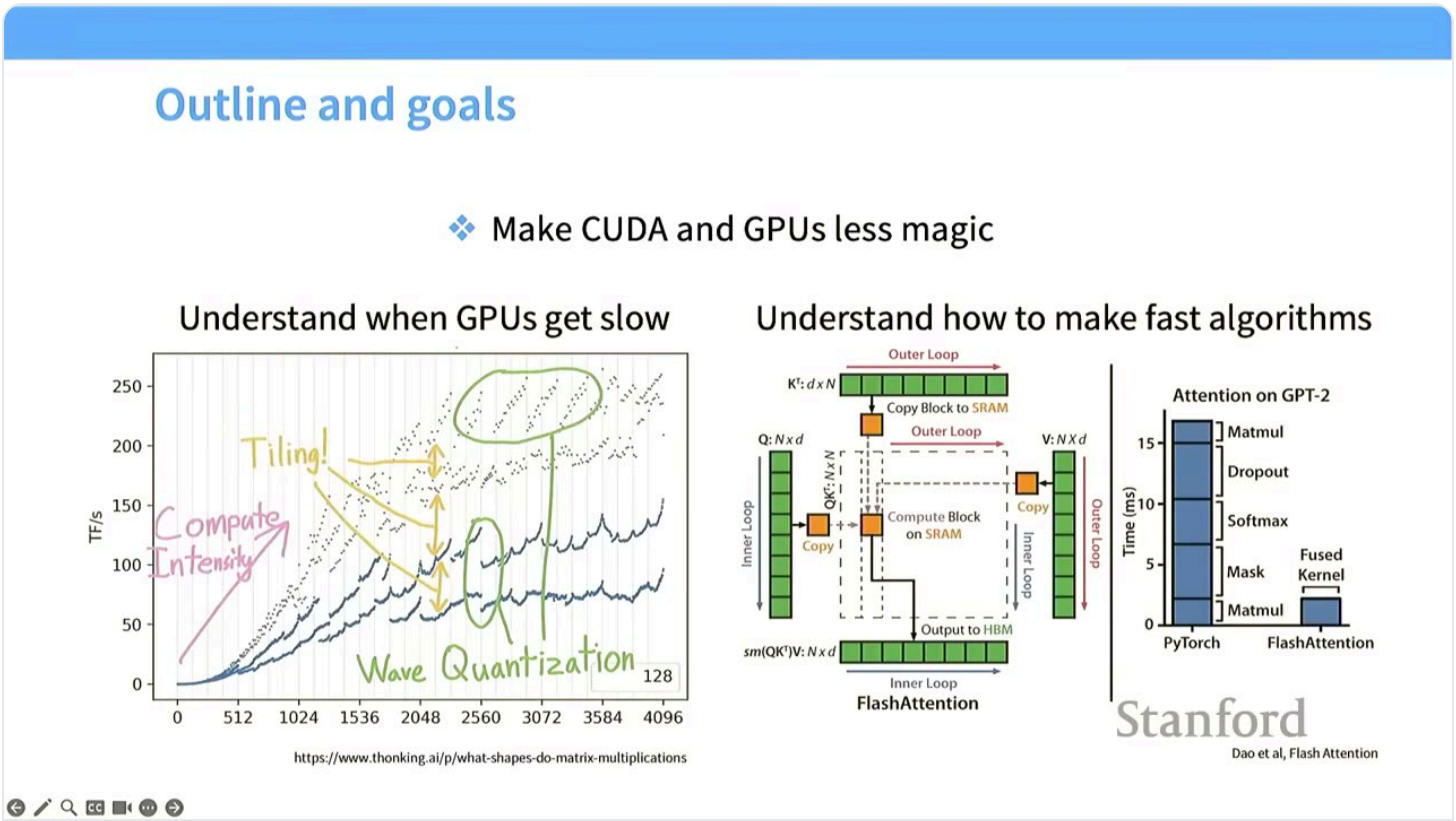
[0:05] Outline and Goals

- **Goal 1: Make CUDA and GPUs less magical.**
  - Understand why GPUs can be slow and exhibit mysterious performance patterns (e.g., wave-like patterns in matrix multiplication throughput).
  - Understand the underlying hardware and execution model.
- **Goal 2: Understand how to make fast algorithms.**
  - Learn the primitives and components needed to accelerate algorithms like Flash Attention.
  - Feel comfortable accelerating new architectures with CUDA.

[0:27] Why GPUs Seem Mysterious



0:28 The slide shows the outline and goals of the lecture, focusing on making CUDA and GPUs less magical.



0:50 The slide details the lecture's goals: understanding GPU slowdown, and how to make fast algorithms.

- GPUs can show unpredictable performance. For example, when increasing matrix multiply sizes, throughput (TFlops/s) doesn't scale smoothly but shows wave-like patterns.
- We will explore concepts like tiling, compute intensity, and wave quantization to demystify these behaviors.

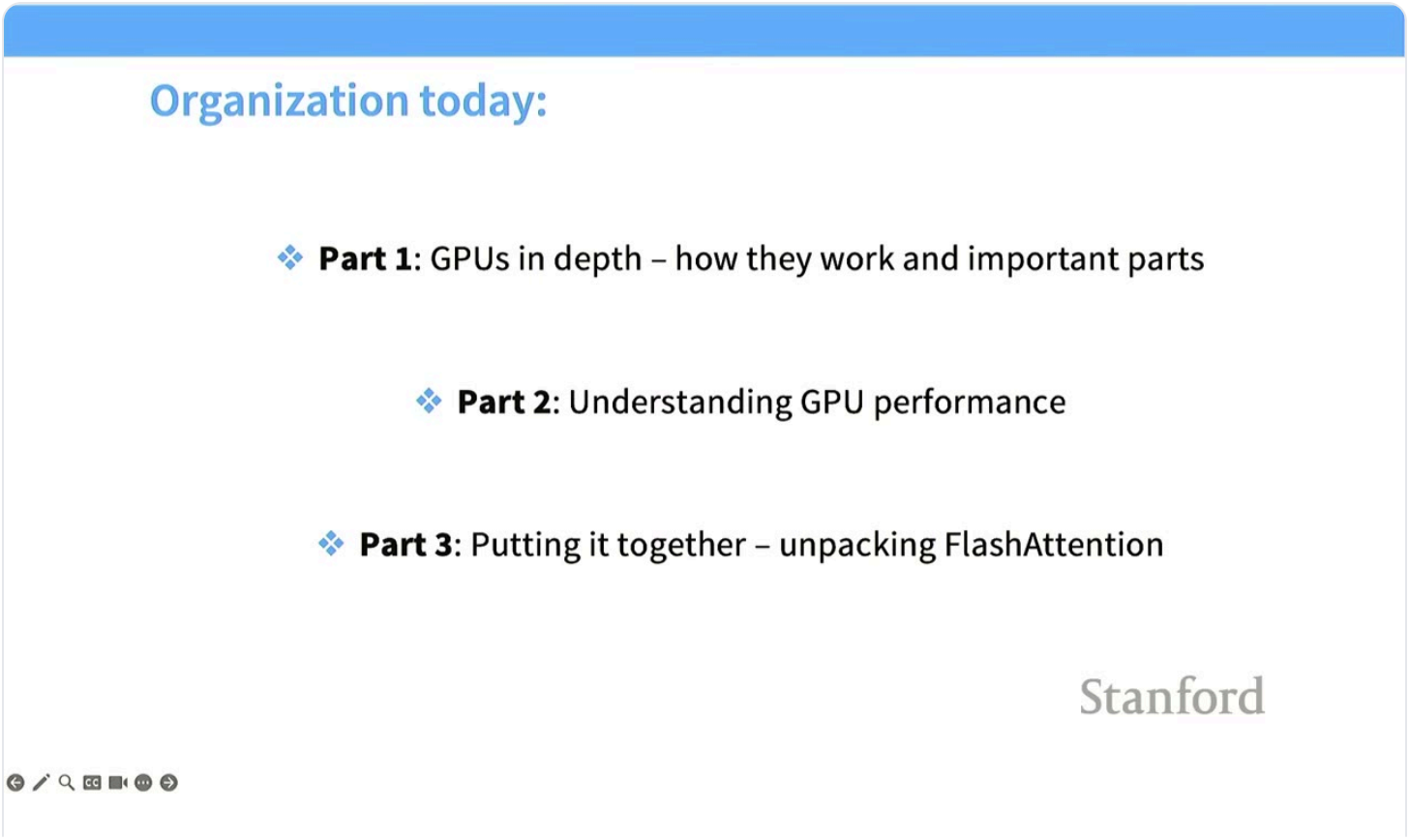
[2:11] Acknowledgements

- Substantial credit to resources like Horace He's blog, CUDA Mode group, and the TPU book from Google. These resources offer deeper dives into GPU and hardware specifics.

[2:44] Organization of Today's Lecture

1. **GPUs in depth:** How they work and important parts.
2. **Understanding GPU performance:** What makes GPUs fast or slow.
3. **Putting it together:** Unpacking Flash Attention.

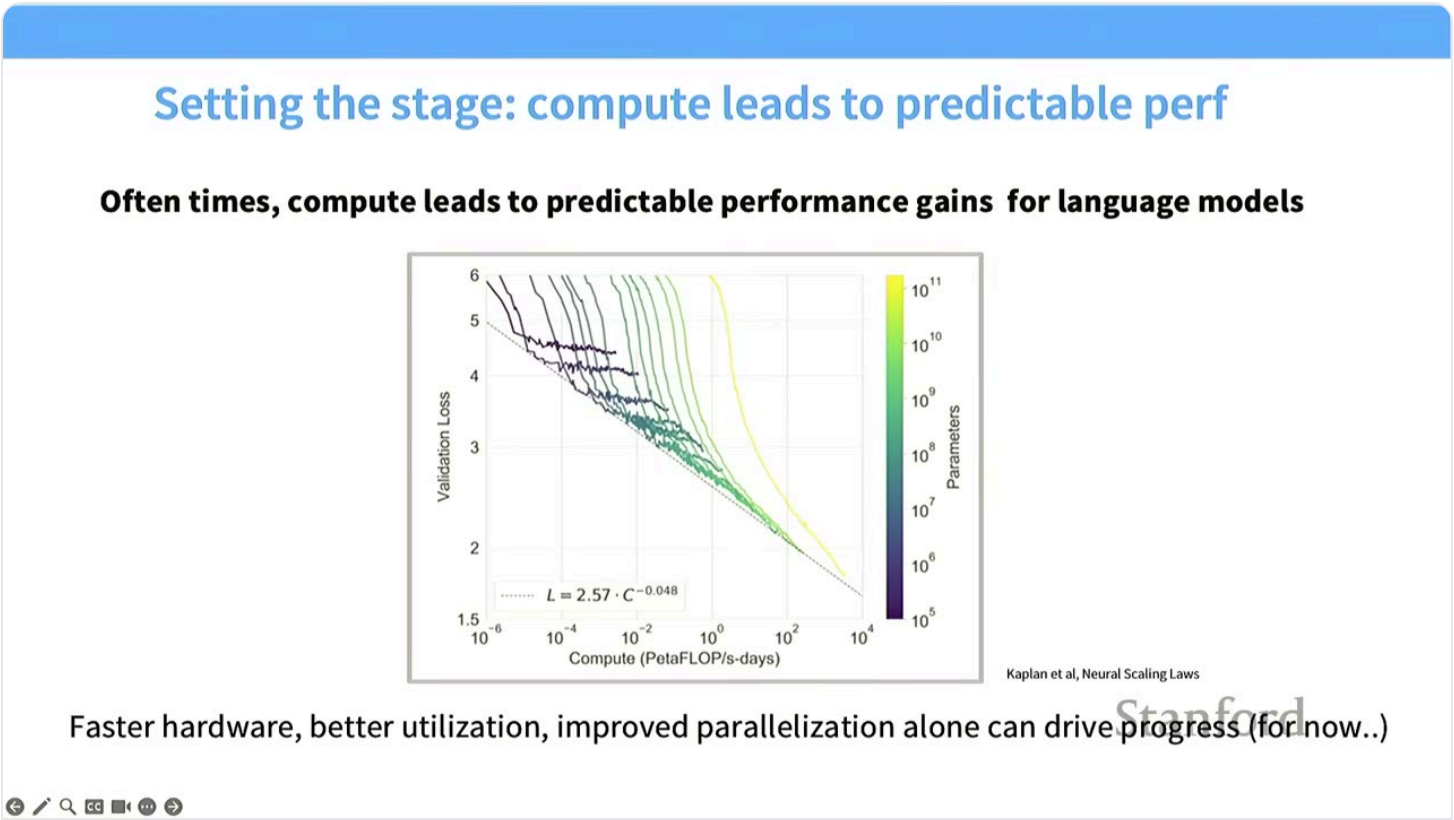
[3:35] Setting the Stage: Compute Leads to Predictable Performance



3:32 The slide outlines the lecture's organization into three parts: GPUs, GPU performance, and FlashAttention.

- **Compute is critical for Language Models:** More compute allows for processing more data, training larger models, and ultimately leads to improved performance.
- **Performance drivers:** Faster hardware, better utilization, and improved parallelization are key.

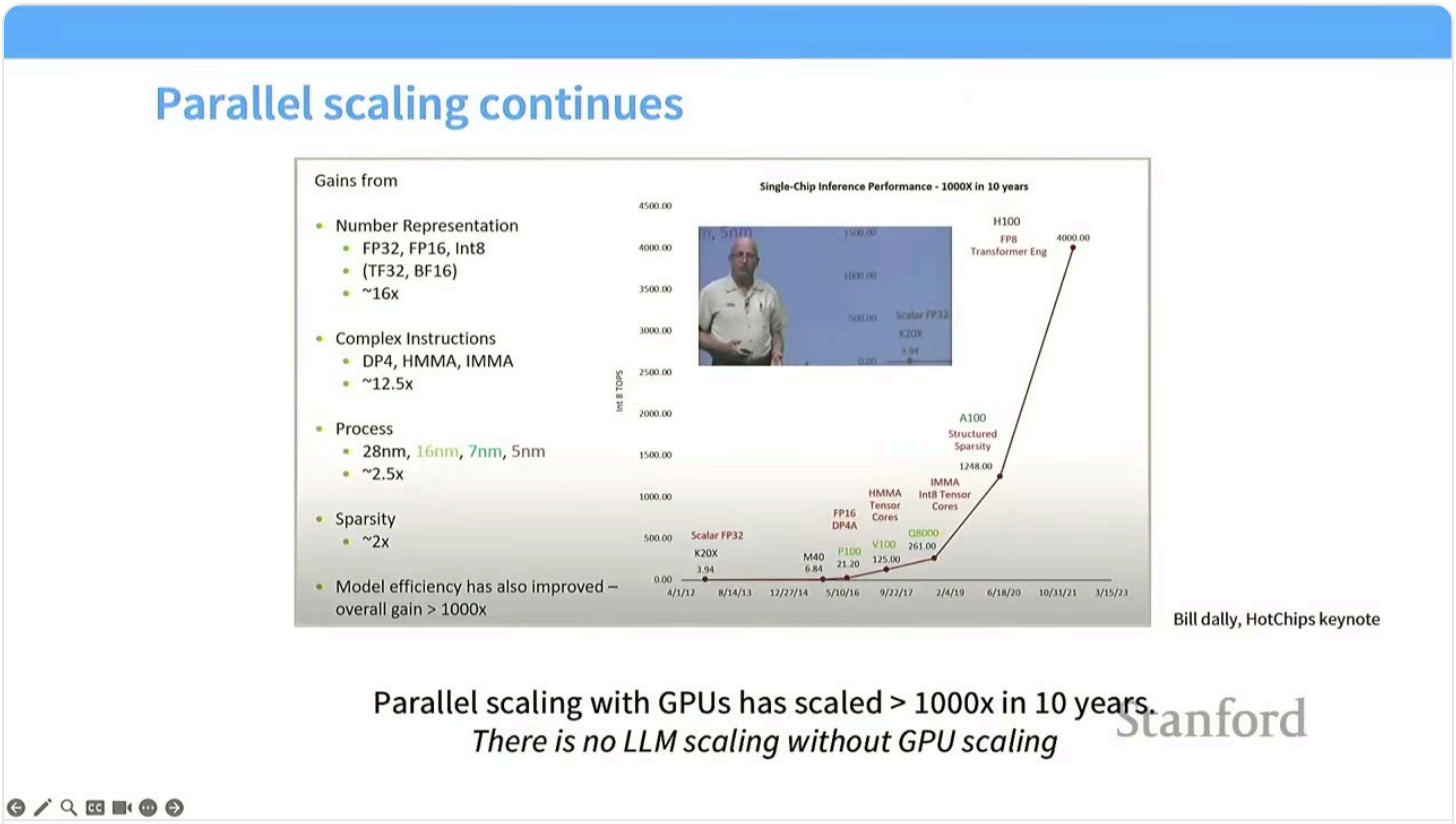
[4:20] How We Get Compute Scaling: Early On - Dennard Scaling



4:04 The slide presents a graph showing that compute leads to predictable performance gains for language models.

- **Dennard Scaling (1980s-2000s):** In the early days of semiconductor scaling, CPUs got faster by packing more transistors onto a chip (Moore's Law). Smaller transistors could run at higher clock speeds with lower power, leading to increased single-thread performance.
- **End of Dennard Scaling:** Around the 2000s, single-thread performance started to taper off (blue dots in the chart). While transistor counts continued to increase, this no longer translated to proportional gains in single-thread throughput.

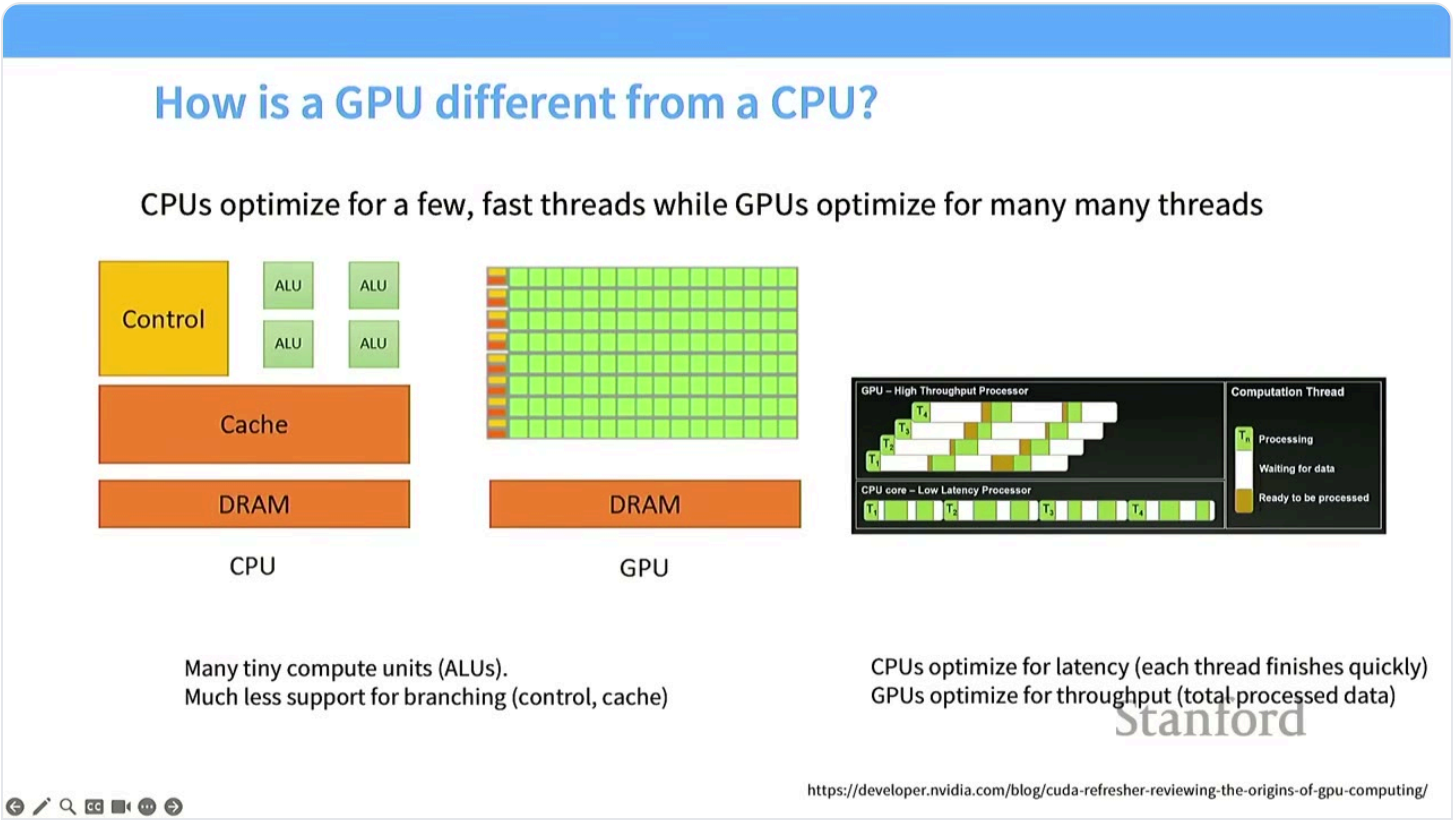
[5:30] Parallel Scaling Continues



5:35 The slide illustrates how parallel scaling with GPUs has improved performance over 1000x in 10 years.

- **Shift to Parallel Scaling:** The story of scaling for deep learning and neural networks moved from single-thread scaling (doing computation faster in absolute terms) to parallel scaling (performing many workloads simultaneously).
- **GPU Performance Growth:** GPUs have shown super-exponential growth in integer operations per second (INT8 TOPS), as illustrated by Bill Dally's keynote chart. This growth is driven by:
  - **Number Representation:** Moving from FP32 to FP16, BF16, and INT8 (up to 16x gain).
  - **Complex Instructions:** Introducing specialized instructions like DP4, HMMA, IMMA (up to 12.5x gain).
  - **Process:** Smaller manufacturing processes (28nm down to 5nm) (up to 2.5x gain).
  - **Sparsity:** Leveraging sparsity in models (up to 2x gain).
  - **Model Efficiency:** Overall gains exceeding 1000x.
- To achieve high performance in language models, we must understand and leverage this parallel scaling.

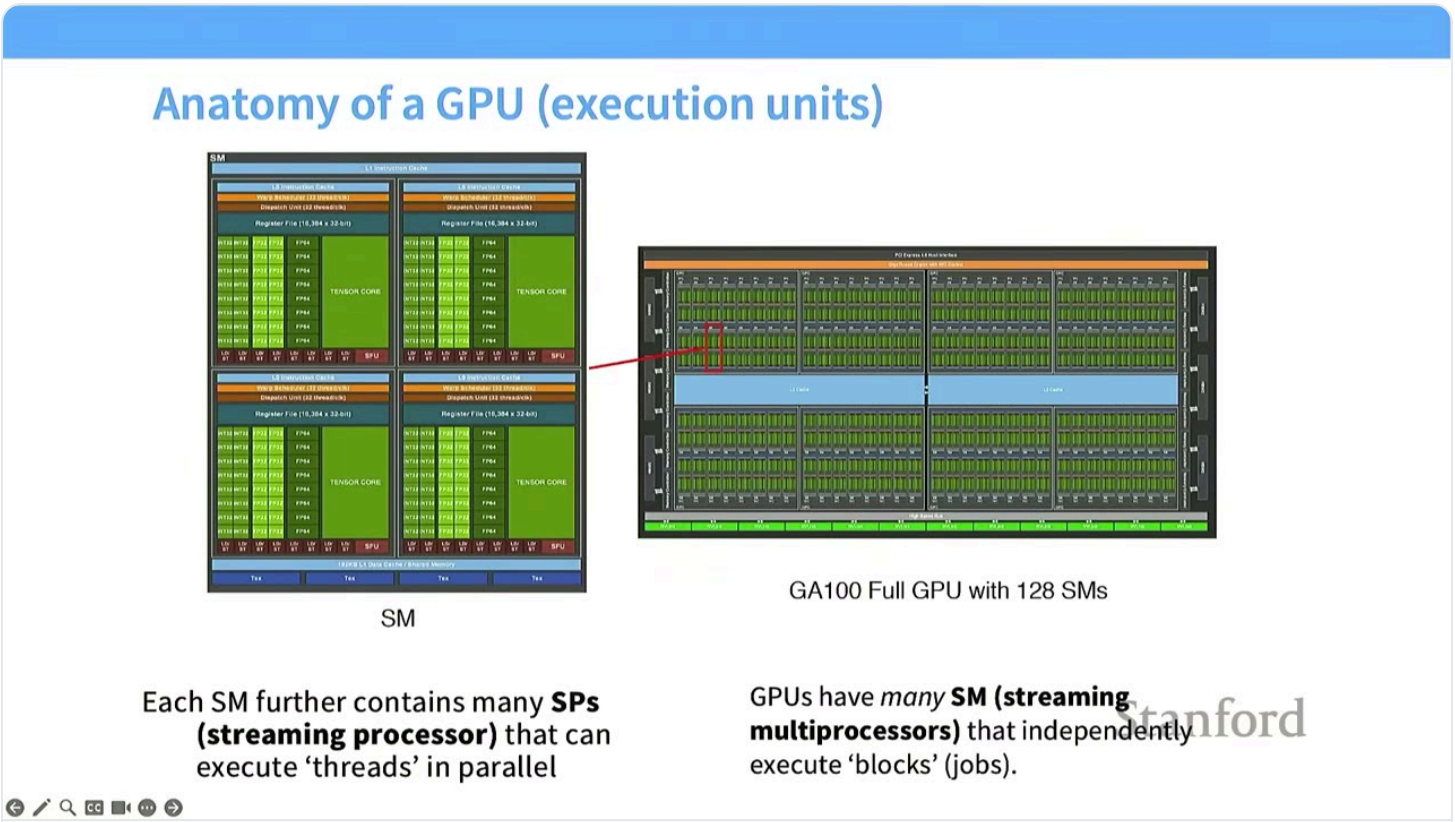
[6:22] How is a GPU Different from a CPU?



6:59 The slide compares CPU and GPU architectures, highlighting their different optimization strategies for threads.

- **CPU Design:** Optimized for **latency** (finish tasks quickly).
  - Features: Large control units, complex branch prediction, large caches.
  - Goal: Execute a few threads very fast.
- **GPU Design:** Optimized for **throughput** (process total data quickly).
  - Features: Many tiny compute units (ALUs), much less support for branching/control.
  - Goal: Execute many threads in parallel.
- **Visual Analogy:**
  - **CPU:** A few powerful workers (ALUs) with a sophisticated manager (Control) and fast local storage (Cache). Each worker processes a task to completion as fast as possible.
  - **GPU:** Many simple workers (ALUs) with minimal management (Control). Tasks are processed in parallel, with the goal of maximizing the total amount of work done over time, even if individual tasks take longer.

[8:28] Anatomy of a GPU (Execution Units)



8:50 The slide shows the anatomy of a GPU, detailing streaming multiprocessors (SMs) and their execution units.

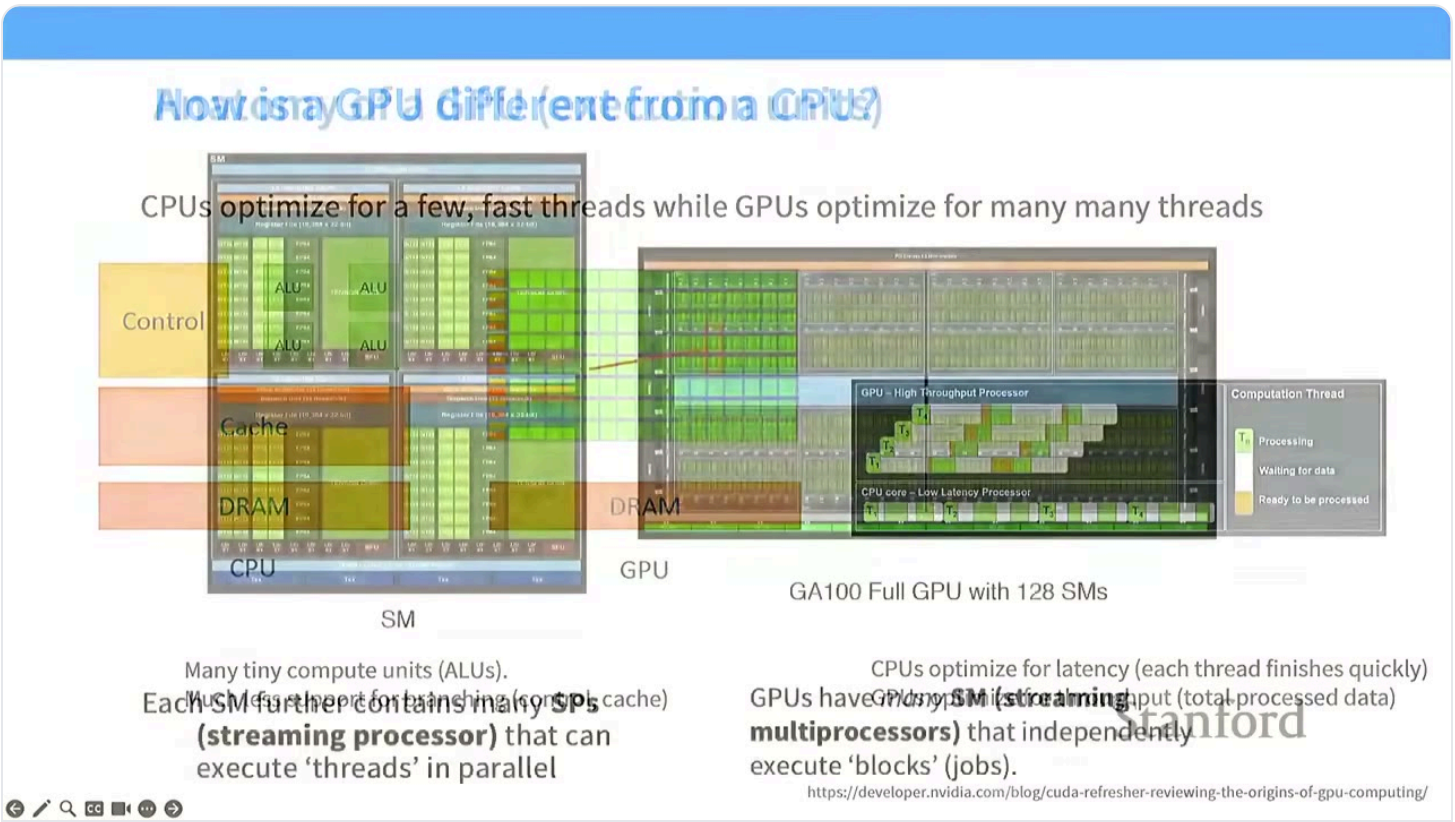
- **Streaming Multiprocessor (SM):** The atomic unit of execution on a GPU. GPUs have many SMs (e.g., GA100 Full GPU with 128 SMs). SMs can independently execute "blocks" (jobs).
- **Streaming Processor (SP):** Each SM contains many SPs. SPs execute "threads" in parallel.
- **Execution Model:**
  - An SM has control logic and can decide what to execute (e.g., branching).
  - SPs within an SM take the same instruction and apply it to many different pieces of data (Single Instruction, Multiple Threads - SIMT). This allows for massive parallel computation.

[9:57] Anatomy of a GPU (Memory)

- **Memory Hierarchy is Crucial:** The closer memory is to the SM, the faster it is. Physical proximity matters significantly for performance.
- **Memory Types and Latencies (approximate clock cycles):**
  - **Registers / Shared Memory (L1 cache):** Inside the SM, very fast (20-33 cycles).
  - **L2 Cache:** On-die (on the GPU chip), still relatively fast (200 cycles).
  - **Global Memory (HBM/DRAM):** Off-chip, slowest (290 cycles).

- **Impact of Slow Memory:** If a computation requires frequent access to global memory, the SMs might run out of work and sit idle, leading to poor utilization. This is a key theme in GPU optimization.

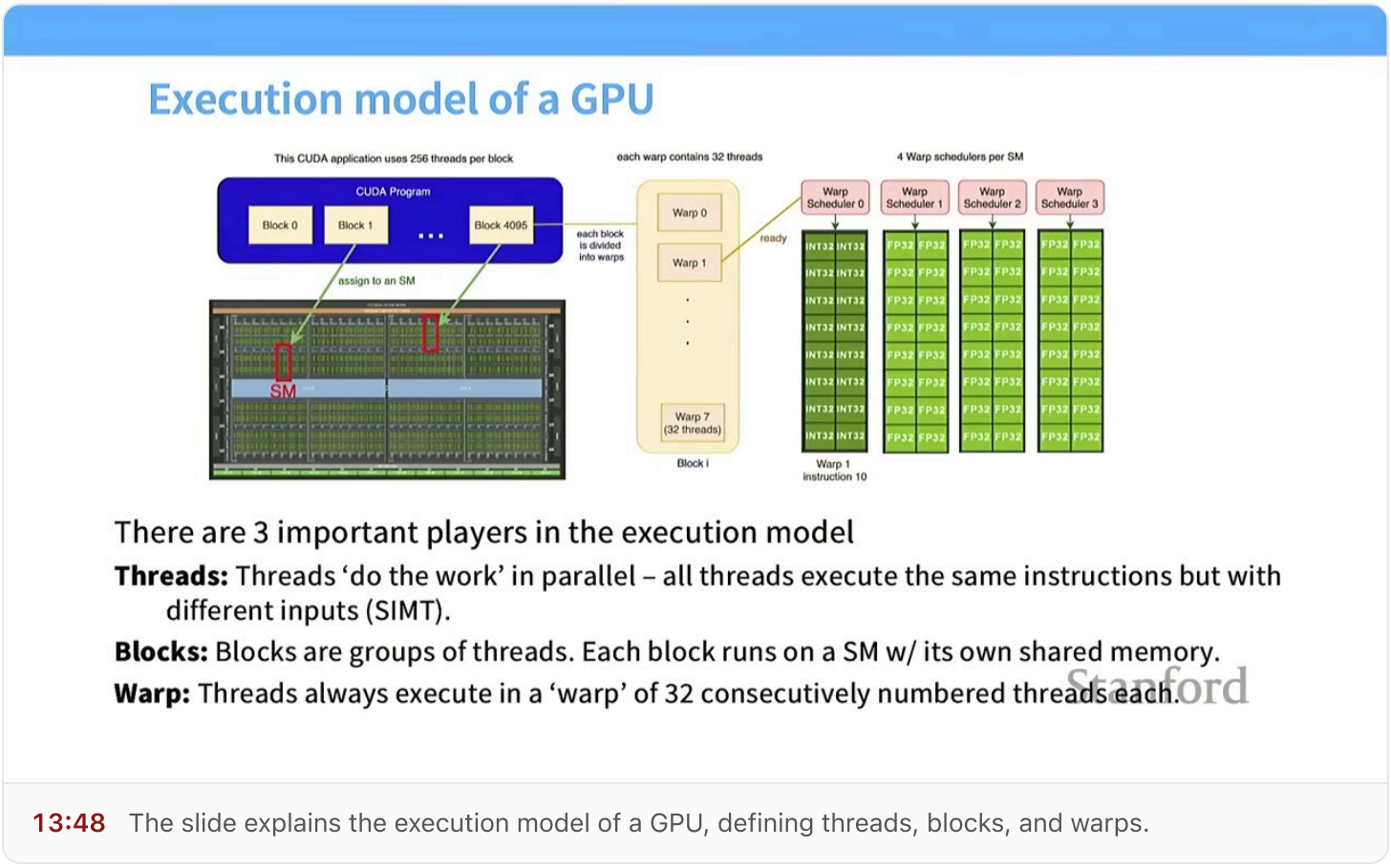
[10:05] Is this GPU the same as that GPU?



10:22 This slide is a duplicate of a previous slide, comparing CPU and GPU architectures and their execution units.

- The previous diagram was a cartoon version. The detailed diagram shows SMs (green blocks) with their own control units and specialized processing units (like FP32, INT32, Tensor Cores).
- HBM (High Bandwidth Memory) connectors (yellow) are visible on the chip diagram, connecting to off-chip DRAM.

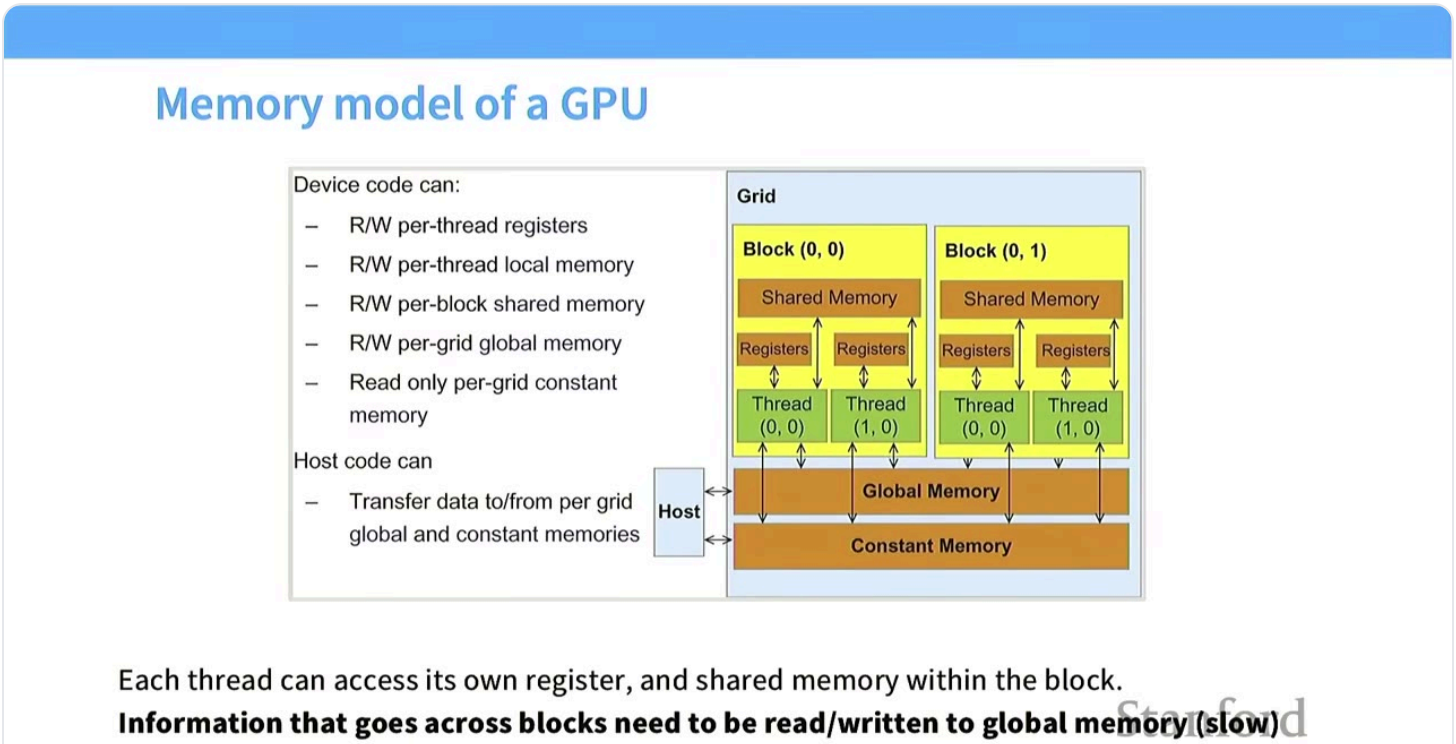
[13:15] Execution Model of a GPU



13:48 The slide explains the execution model of a GPU, defining threads, blocks, and warps.

- **Three important players:**
  1. **Threads:** "Do the work" in parallel. All threads execute the same instructions but with different inputs (SIMT).
  2. **Blocks:** Groups of threads. Each block runs on an SM with its own shared memory.
  3. **Warp:** Threads always execute in a "warp" of 32 consecutively numbered threads each.
- **Process:**
  - A CUDA program launches many blocks.
  - Each block is assigned to an SM.
  - Within an SM, a block is divided into warps.
  - Each warp (32 threads) is scheduled by a Warp Scheduler to run on the SM's SPs.
  - All threads in a warp execute the same instruction on different data.

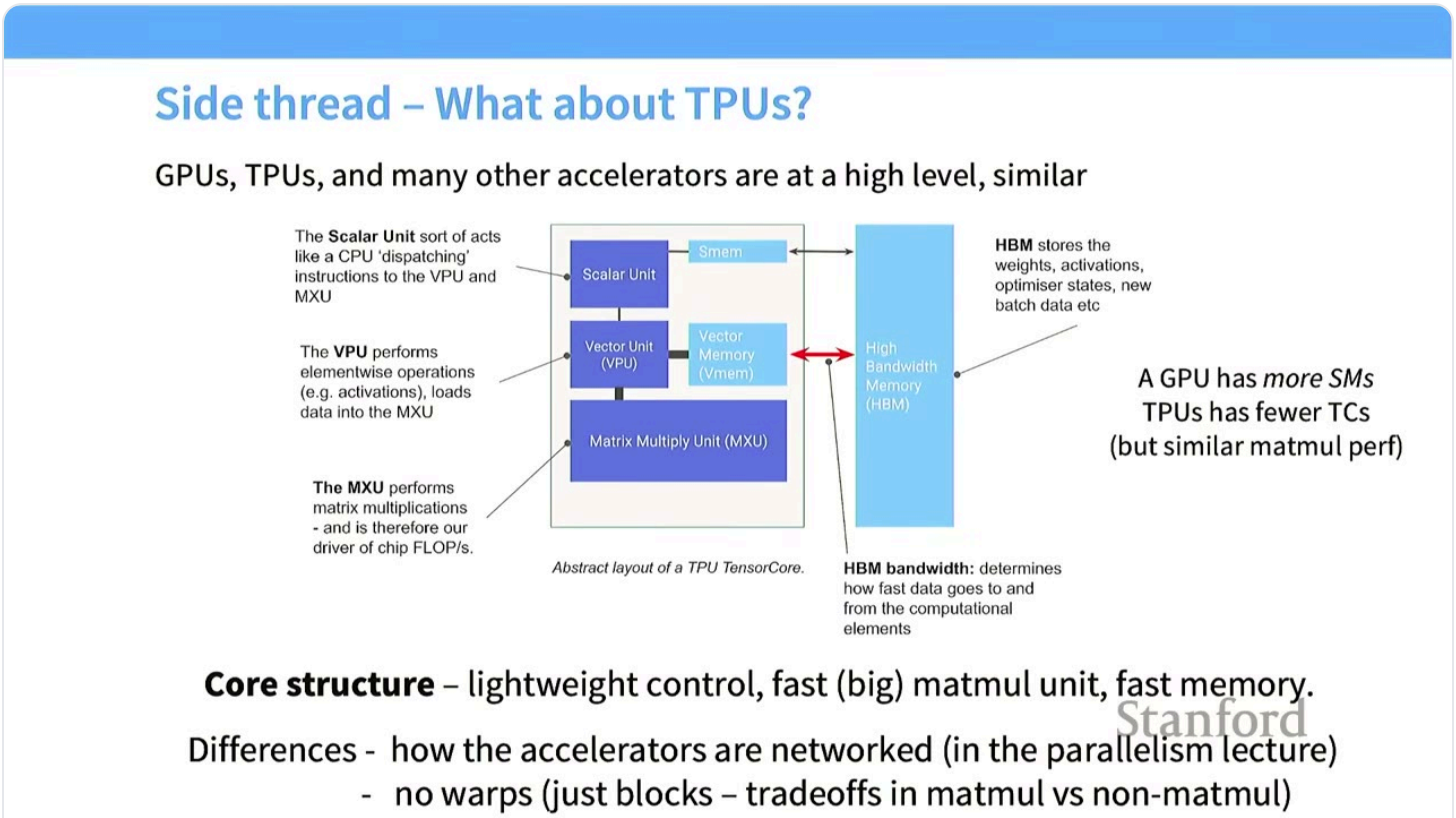
[15:04] Memory Model of a GPU



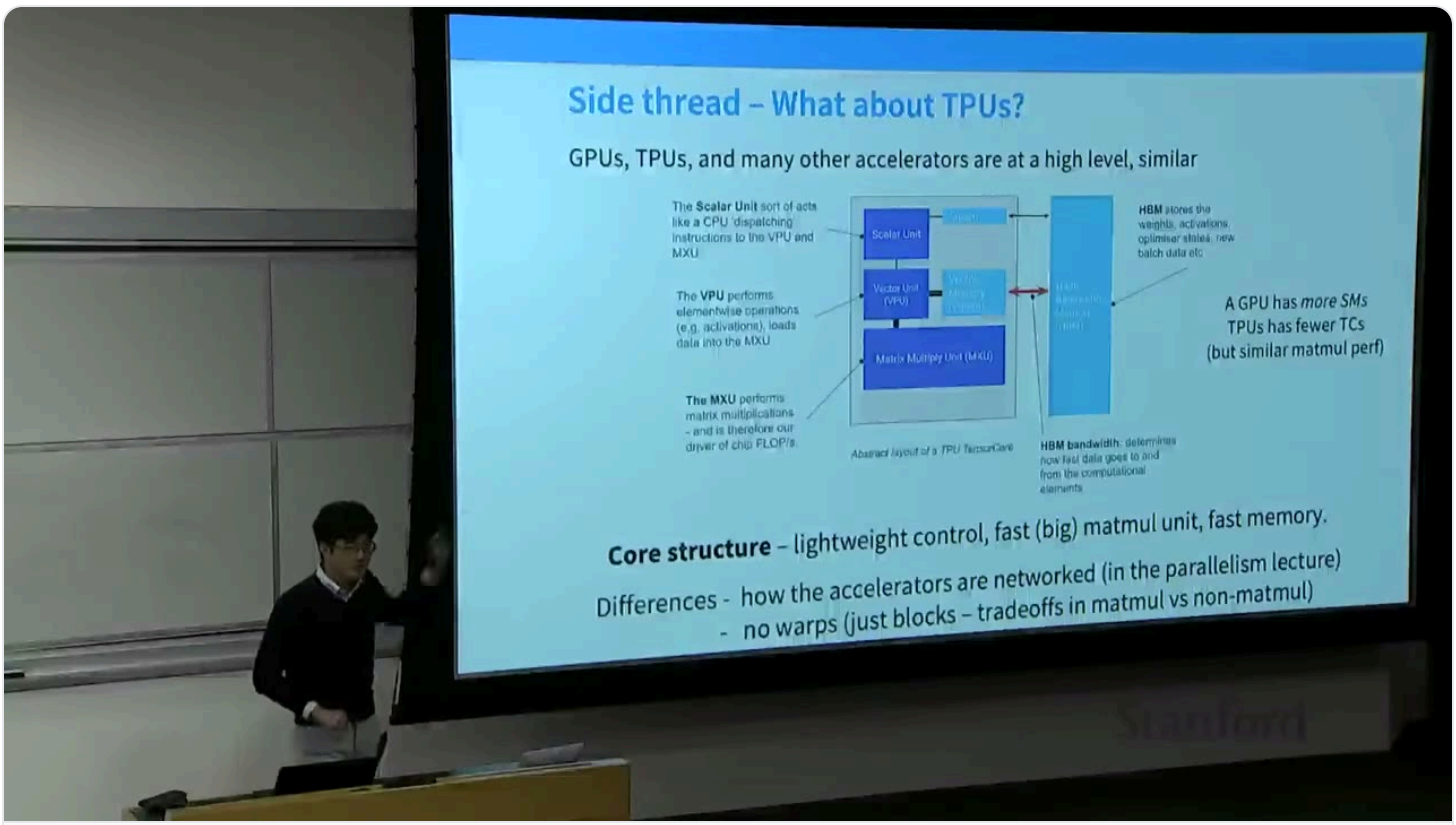
15:57 The slide illustrates the memory model of a GPU, showing different memory types and their accessibility.

- **Device Code (GPU) can access:**
  - Read/Write per-thread registers (fastest)
  - Read/Write per-thread local memory
  - Read/Write per-block shared memory
  - Read/Write per-grid global memory (slowest)
  - Read only per-grid constant memory
- **Host Code (CPU) can:**
  - Transfer data to/from per-grid global and constant memories.
- **Key Principle:** Each thread can access its own register and shared memory within the block. Information that goes across blocks needs to be written to global memory.
- **Implication:** Accessing global memory is very slow. Ideally, threads should operate on small amounts of data loaded into fast shared memory.

[16:34] Side Thread - What about TPUs?



17:19 The slide discusses TPUs as an alternative to GPUs, highlighting their core structure and differences.



18:30 The slide discusses TPUs as an alternative to GPUs, highlighting their core structure and differences.

- **TPUs (Tensor Processing Units):** Google's custom ASICs for ML.

- **High-level Similarity to GPUs:** TPUs, GPUs, and many other accelerators share a similar high-level structure.
    - **Tensor Core (analogous to SM):** An atomic unit that operates on data.
    - **Scalar Unit:** Control unit, dispatches instructions.
    - **Vector Unit (VPU):** Performs element-wise operations on vectors.
    - **Matrix Multiply Unit (MXU):** Dedicated hardware for matrix multiplications.
    - **Memory:** Fast on-core memory (Vector Memory, SMEM) and High Bandwidth Memory (HBM) off-core.
- **Core Structure:** Lightweight control, fast (big) matrix multiply, fast memory.
- **Differences:**
    - TPUs have fewer Tensor Cores (TCs) than GPUs have SMs, but individual TCs are more powerful.
    - TPUs have no concept of "warps" or "blocks" in the same way as GPUs. They are optimized purely for matrix multiplications, making them simpler.
    - The networking of accelerators is different (covered in a later lecture).

[19:15] Strengths of the GPU Model

1. **Easily scales up hard workloads:** By adding more SMs, performance can be increased without necessarily increasing clock speed or dealing with heat dissipation issues.
2. **Easy (?) to program due to the SIMT model:** The Single Instruction, Multiple Threads (SIMT) model, where all threads in a warp execute the same instruction on different data, is conceptually straightforward for parallelizing operations on matrices.
3. **Threads are "lightweight":** GPU threads have minimal state and can be stopped and started quickly. This allows the GPU to hide latency by switching between active threads, ensuring high utilization even when some threads are waiting for data.

[20:18] GPUs as Fast Matrix Multipliers

- **Early Days:** GPUs (Graphics Processing Units) were initially designed for graphics. Researchers "hacked" programmable shaders to perform matrix multiplications.
- **Modern Hardware:** NVIDIA and other manufacturers realized the importance of matrix multiplications for deep learning.
    - **Tensor Cores:** Introduced in the V (Volta) and T (Turing) series, Tensor Cores are specialized matrix multiplication circuits.

    - **Performance Gap:** Matmuls are now >10x faster than other floating-point operations on GPUs.
- **Implication:** If you're designing a neural architecture, most of your workload should be matrix multiplications to leverage this specialized hardware. Non-matmul based operations will be significantly slower.

[22:03] Compute Scaling is Faster than Memory Scaling

- **Relative Scaling:**
    - **HW FLOPs (Compute):** Scales incredibly fast (60,000x in 20 years, ~3.0x every 2 years).
    - **DRAM BW (Memory):** Scales much slower (100x in 20 years, ~1.2x every 2 years).
    - **Interconnect BW (Host-GPU):** Scales even slower (20x in 20 years, ~1.1x every 2 years).
- **Memory Bottleneck:** While early ML problems might have been compute-bound, modern GPUs are so fast that memory access (DRAM bandwidth) is now the primary bottleneck. This gap will continue to widen.
- **Design Implication:** Hardware-efficient algorithms must prioritize minimizing memory movement.

[23:55] Recap: GPUs - What are they and how do they work?

- GPUs are massively parallel processors: Same instructions applied across many workers (SMs).
- Compute (especially matmuls) has scaled faster than memory.
- We must respect the memory hierarchy to make things go fast.

[25:02] Part 2: Making ML Workloads Fast on a GPU

- **The Matrix Multiply Mystery Plot:** The initial plot showed unpredictable, wave-like performance patterns for square matrix multiplications as size increases. We aim to understand these patterns.
    - **X-axis:** Size of square matrix (N).
    - **Y-axis:** Throughput (TFlops/s), representing hardware utilization.

- **Observed Patterns:** Multiple lines, wave-like behavior, sudden drops/increases.

[26:21] What Makes ML Workloads Fast? The Roofline Model

- **Roofline Model:** A performance model that identifies whether a workload is compute-bound or memory-bound.
  - **Memory-limited regime (left side):** Performance is limited by the rate at which data can be fetched from memory.
  - **Throughput-limited / Compute-bound regime (right side):** Performance is limited by the raw processing power of the compute units.
  - **Goal:** Avoid being memory-bound and operate in the compute-bound regime (fully utilizing compute units).
- The mystery plot's general shape (diagonal then flat) is consistent with the roofline model.

[27:36] How Do We Make GPUs Go Fast? (Tricks)

- To avoid memory bottlenecks, we need a large array of tricks.
- **1. Control Divergence (not a memory bottleneck):**
  - **SIMT Model:** Every thread in a warp executes the same instruction.
  - **Conditionals (if/else statements):** If threads within a warp take different branches, the GPU must execute both branches sequentially, pausing threads that don't follow the current path. This leads to significant overhead.
  - **Implication:** Avoid conditional statements within warps in performance-critical code.
- **2. Low Precision Computation (Quantization):**
  - **Gains:** Moving from FP32 to FP16, BF16, or INT8 can yield many orders of magnitude performance gains (e.g., 16x for FP32 to INT8).
  - **Reason:** Fewer bits mean fewer bits to move. This effectively increases memory bandwidth for free.
  - **Example (Element-wise ReLU):**
    - **FP32:** 1 read, 1 write (if $x < 0$), 1 comparison, 1 FLOP. Memory access: 8 bytes. Intensity: 8 bytes/FLOP.
    - **FP16:** 1 read, 1 write (if $x < 0$), 1 comparison, 1 FLOP. Memory access: 4 bytes. Intensity: 4 bytes/FLOP.
    - **Result:** Halving memory access effectively doubles memory bandwidth.

- **Tensor Cores:** Modern GPUs (like NVIDIA's Tensor Cores) are designed for mixed precision. Inputs can be 16-bit, intermediate accumulations can be 32-bit for precision, and the final result can be downcast to 16-bit.
- **Careful Engineering:** Not all parts of a network or training algorithm can be low precision. Operations needing high precision (e.g., sums, reductions) or high dynamic range (e.g., exponential functions) might need FP32 or BF16.

- **3. Operator Fusion:**
  - **Problem:** Naive execution of sequential operations (e.g., square -> triangle -> circle) involves repeatedly writing intermediate results to global memory and reading them back. This incurs significant memory overhead.
  - **Solution (Fused Kernel):** Combine multiple sequential operations into a single CUDA kernel. The intermediate results stay in fast on-chip memory (registers, shared memory) and are not written back to global memory until the final output.
  - **Example (sin^2(x) + cos^2(x)):** Naive implementation launches 5 separate CUDA kernels, each involving global memory access. A fused kernel performs all operations on-chip, dramatically reducing memory traffic.
  - **Automation:** "Easy" fusions can be done automatically by compilers like `torch.compile`.

- **4. Recomputation:**
  - **Problem (Backpropagation):** During backpropagation, activations from the forward pass are needed to compute gradients. Storing all activations can consume significant memory and lead to slow global memory accesses.
  - **Solution:** Instead of storing all activations, recompute them on the fly during the backward pass.
  - **Example (Stacked Sigmoids):**
    - **Normal Forward Pass:** 1 memory read (x), 3 memory writes (S1, S2, out).
    - **Normal Backward Pass:** 3 memory reads (S1, S2, dout), 1 memory write (dx). Total 8 memory accesses.
    - **Recomputed Forward Pass:** 1 memory read (x), 1 memory write (out). S1 and S2 are not stored.
    - **Recomputed Backward Pass:** 2 memory reads (x, dout). S1 and S2 are recomputed on-chip. 1 memory write (dx). Total 5 memory accesses.

- **Trade-off:** Recomputation increases compute but significantly reduces memory accesses. This is beneficial when compute is abundant and memory bandwidth is scarce.
- **Relation to Checkpointing:** Similar to gradient checkpointing, but here the goal is execution speed rather than just memory savings.

- **5. Memory Coalescing and DRAM:**
  - **DRAM Burst Mode:** When reading from global memory (DRAM), the hardware doesn't return just the requested byte, but a whole "burst section" (e.g., 4 bytes, 16 bytes, 128 bytes) for free. This is because moving data to the sense amplifier is the slow step; once there, many bytes can be read quickly.
  - **Coalesced Accesses:** Memory accesses are "coalesced" if all threads in a warp access locations that fall within the same burst section. This allows the GPU to fetch multiple data points with a single, efficient DRAM read.
  - **Impact on Matrix Multiplication:**
    - **Row-major access (not coalesced):** If threads move along rows (e.g., Thread 0 reads M0,0, Thread 1 reads M0,1), their memory accesses might fall into different burst sections, leading to many inefficient DRAM reads.
    - **Column-major access (coalesced):** If threads move down columns (e.g., Thread 0 reads M0,0, Thread 1 reads M1,0), their memory accesses are more likely to fall within the same burst section, allowing for coalesced reads and higher throughput.
  - **Importance:** Correct memory traversal order is crucial for performance. Misaligned access patterns can drastically slow down memory access.

- **6. Tiling (The Big One):**
  - **Idea:** Grouping and ordering threads to minimize global memory access.
  - **Process:**
    1. Cut large matrices into smaller "tiles."
    2. Load a tile from global memory into fast shared memory.
    3. Perform all necessary computations on that tile within shared memory.
    4. Write the final result back to global memory (or accumulate if possible).
    5. Repeat for the next tile.
  - **Advantages:**

        - **Reduced Global Memory Access:** Each input is read N/T times from global memory (where N is matrix size, T is tile size), and T times within shared memory. This is a factor of T reduction in global memory access.
        - **Repeated Reads Now Shared:** Data within a tile is loaded once and reused by multiple threads, reducing redundant global memory reads.
        - **Memory Access Can Be Coalesced:** Within a tile, access patterns can be optimized for coalescing.
    - **Complexities:**
        - **Tile Size Selection:** Factors affecting optimal tile size include coalesced memory access, shared memory size, and divisibility of the matrix dimensions.
        - **Divisibility Issues:** If matrix dimensions are not perfectly divisible by tile sizes, some tiles will be very sparse, leading to underutilized SMs and reduced overall performance. This requires careful padding.
        - **Memory Alignment:** Tiles must align with burst sections in DRAM for fast loading. Misalignment can double memory accesses.

[1:10:50] Putting it all together: The Forward Pass of Flash Attention

- **Matrix Multiply Mystery Revisited:** The wave-like patterns in matrix multiply performance are due to the complex interplay of compute intensity, tiling, and wave quantization (burst mode alignment).
- **Andrej Karpathy's Tweet:** Highlights how a seemingly small change in vocabulary size (50257 to 50304, nearest multiple of 64) can lead to a 25% speedup in nanoGPT, due to better memory alignment and higher occupancy. This illustrates the importance of low-level details.

[1:11:10] Flash Attention: How it Dramatically Accelerates Attention

- **Standard Attention Computation:** Involves three matrix multiplies (Q, K, V) and a softmax operation.
- **The Challenge of Softmax:** Softmax is a global operation, requiring summing the entire row of attention scores. This is problematic for tiling, as it breaks the locality needed to keep computations within tiles.
- **Online Softmax (from Mikailov and Gimelshein 2018):**

    - **Normal Softmax:** Requires all values (x1 to xn) upfront to compute the maximum and sum for normalization.
    - **Online Softmax:** Allows incremental computation of the softmax. It maintains a running maximum and a telescoping sum.
    - **Key Idea:** You don't need all x1 to xn upfront. You can process a stream of values, incrementally updating the max and sum.
    - **Benefit:** This allows computing the softmax tile-by-tile.
- **Flash Attention's Forward Pass (from Dao 2023):**
    1. **Tile-wise computation of the inner products (S):** The QK^T matrix multiply is tiled, as discussed.
    2. **Fusion of the exponential operator:** The exponential function (part of softmax) is fused with the inner product computation.
    3. **Tile-wise computation of the softmax via the online, telescoping sum trick:** This is the crucial part. By using the online softmax, Flash Attention can compute the softmax within each tile, incrementally updating the necessary components (max and sum) without needing to materialize the full N^2 attention matrix.
    4. **Output:** Once all tiles are processed, the final output is computed by combining the partial results.
- **Backward Pass:** Flash Attention uses recomputation tile-by-tile for the backward pass to avoid storing the N^2 attention matrix, which would be too memory-intensive.

[1:12:54] Recap for the Whole Lecture

- **Hardware Power:** Hardware scales, and low-level details determine what scales or doesn't.
- **Current GPU Compute:** Strongly encourages compute based on matrix multiplication and efficient data movement.
- **Thinking Carefully about the GPU:** Coalescing, tiling, and fusion are key techniques that lead to good performance on things like Flash Attention.

Practical Takeaways - **Memory is King:** For modern ML on GPUs, memory bandwidth is often the bottleneck, not raw FLOPs. Prioritize minimizing global memory access. - **Embrace Low-Level Optimizations:** Techniques like tiling, fusion, coalescing, and recomputation are not just academic exercises; they are essential for achieving high performance. - **Understand the Memory Hierarchy:** Design algorithms to keep data in the fastest available memory (registers, shared memory) as much as possible. - **Be Mindful of Data Access Patterns:** Optimize memory

access patterns to leverage hardware features like burst mode and avoid control divergence. - **Leverage Low Precision:** Use FP16, BF16, or INT8 where possible to reduce memory traffic and increase effective bandwidth.

Open Questions / Things to Remember - How do different GPU architectures (e.g., NVIDIA vs. AMD vs. Intel) impact the optimal values for tiling sizes, burst sections, and other low-level parameters? - What are the practical implications of these low-level optimizations for researchers who primarily use high-level frameworks like PyTorch or TensorFlow? (e.g., how much can compilers like `torch.compile` automate?) - How do these principles extend to multi-GPU and distributed training scenarios? - What are the trade-offs between numerical stability and performance when using mixed precision and quantization?

← Lec 04

Lec 06 →

Lecture 06: Kernels, Triton

Stanford CS336 · Spring 2025 · 1:20:22 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



TL;DR: * **Benchmarking & Profiling are crucial:** Always measure performance to identify bottlenecks, rather than guessing. Use `torch.cuda.synchronize()` and warm-up iterations for accurate GPU timings. * **GPU Architecture:** Understand SMs, memory hierarchy (DRAM, L2, L1, registers), and the execution model (Grid, Thread Block, Thread, Warp) for efficient kernel design. * **Kernel Fusion:** Combine multiple element-wise operations into a single kernel to reduce expensive memory transfers between global memory (DRAM) and on-chip caches. * **CUDA C++:** Provides fine-grained control over GPU hardware, allowing for highly optimized kernels, but requires manual management of threads, blocks, and memory. * **Triton:** A Python-based DSL for GPU programming that offers a higher-level abstraction than CUDA C++, automatically managing memory coalescing, shared memory, and scheduling, often achieving performance comparable to hand-written CUDA. * **`torch.compile`:** PyTorch's JIT compiler can automatically fuse operations

and optimize kernels, providing significant speedups without manual kernel writing, especially for common patterns like element-wise operations and matrix multiplications.

Key concepts: * GPU architecture (SMs, memory hierarchy) * GPU execution model (Grid, Thread Block, Thread, Warp) * Arithmetic intensity (FLOPs/bytes) * Benchmarking (wall-clock time) * Profiling (time spent per function/kernel) * `torch.cuda.synchronize()` * Warm-up iterations * CPU-GPU asynchronous execution * Kernel fusion * CUDA C++ * Triton DSL * PTX (Parallel Thread Execution) * `torch.compile` (JIT compilation) * Memory coalescing * Shared memory

---

[0:00] Introduction and Lecture Plan



0:05 A black screen, likely a transition slide or the beginning/end of the lecture.



0:27 A speaker stands at a podium with a laptop, presenting in a lecture hall.

This lecture focuses on writing high-performance code for GPUs, specifically for standard components in language models.

Lecture Agenda: 1. **Review of GPUs:** A brief overview of GPU architecture and execution model. 2. **Benchmarking & Profiling:** Basic techniques for measuring and understanding performance. 3. **Kernel Fusion Motivation:** Understanding why combining operations is crucial for performance. 4. **CUDA Kernels in C++:** Writing custom GPU kernels using CUDA C++. 5. **Triton Kernels in Python:** Writing custom GPU kernels using the Triton DSL. 6. **PyTorch Compilation:** Leveraging PyTorch's built-in JIT compiler for automatic optimization. 7. **More Advanced Computations:** Applying these techniques to a softmax implementation.

The goal is to dig deep into the PTX (Parallel Thread Execution) machine code to understand what the GPU is doing under the hood.

[1:47] Announcements

- **Assignment 1:** Has concluded. The leaderboard is still active for submissions and updates.

- **Assignment 2:** Is now released. It involves GPU kernels and parallelism. This lecture will cover the GPU kernel part, and next week will cover parallelism (data parallelism, etc.).

[2:25] Review of GPUs

Hardware Overview

- **GPU Structure (e.g., A100/H100):** Consists of many **Streaming Multiprocessors (SMs)**. An A100 has 108 SMs.
- **SM Structure:** Each SM contains a large number of units for computation (e.g., FP32, INT32 cores, Tensor Cores).
- **Memory Hierarchy:**
  - **DRAM (Global Memory):** Large (e.g., 80GB on A100), but slow.
  - **L2 Cache:** Smaller (e.g., 40MB on A100), faster.
  - **L1 Cache (Shared Memory):** Even smaller (e.g., 192KB per SM), very fast.
  - **Register File:** Very fast memory directly accessible by individual threads. Heavy use of registers is key for high-performance code.

Execution Model

- **Grid:** A collection of **Thread Blocks (CTAs - Concurrent Thread Arrays)**.
- **Thread Block (CTA):** A group of threads scheduled on a single SM. This is the atomic unit for scheduling.
- **Thread:** The individual unit of execution (i.e., `f(i)`). Each thread processes a part of the data.

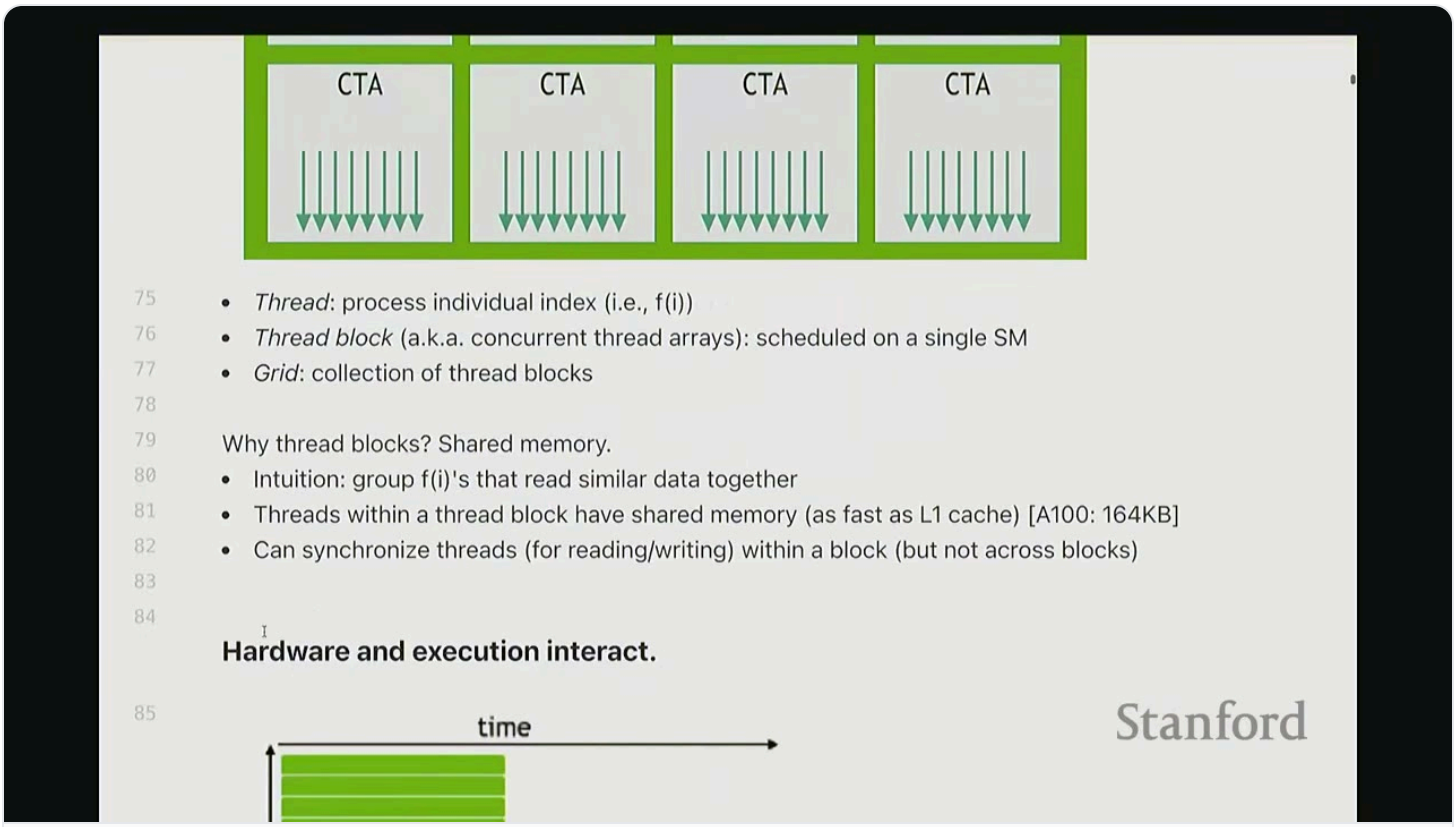
Why Thread Blocks? Shared Memory: * Threads within a thread block can communicate with each other via shared memory (L1 cache speed, e.g., 164KB on A100). This is very fast. * Threads within a block can also synchronize for read/write operations. * Communication *across* thread blocks is very expensive (requires global memory). Therefore, data needed for inter-thread communication should ideally stay within the same thread block.

Waves (Warps): * Threads are scheduled on SMs in **waves**. A wave is a group of 32 threads that execute together. * **Problem:** The last wave in a thread block might have fewer threads, leading to low SM occupancy. * **Wave Quantization:** The number of thread blocks should ideally divide the number of SMs. * **Rule of Thumb:** The number of thread blocks should be $\geq 4\times$ # SMs. *

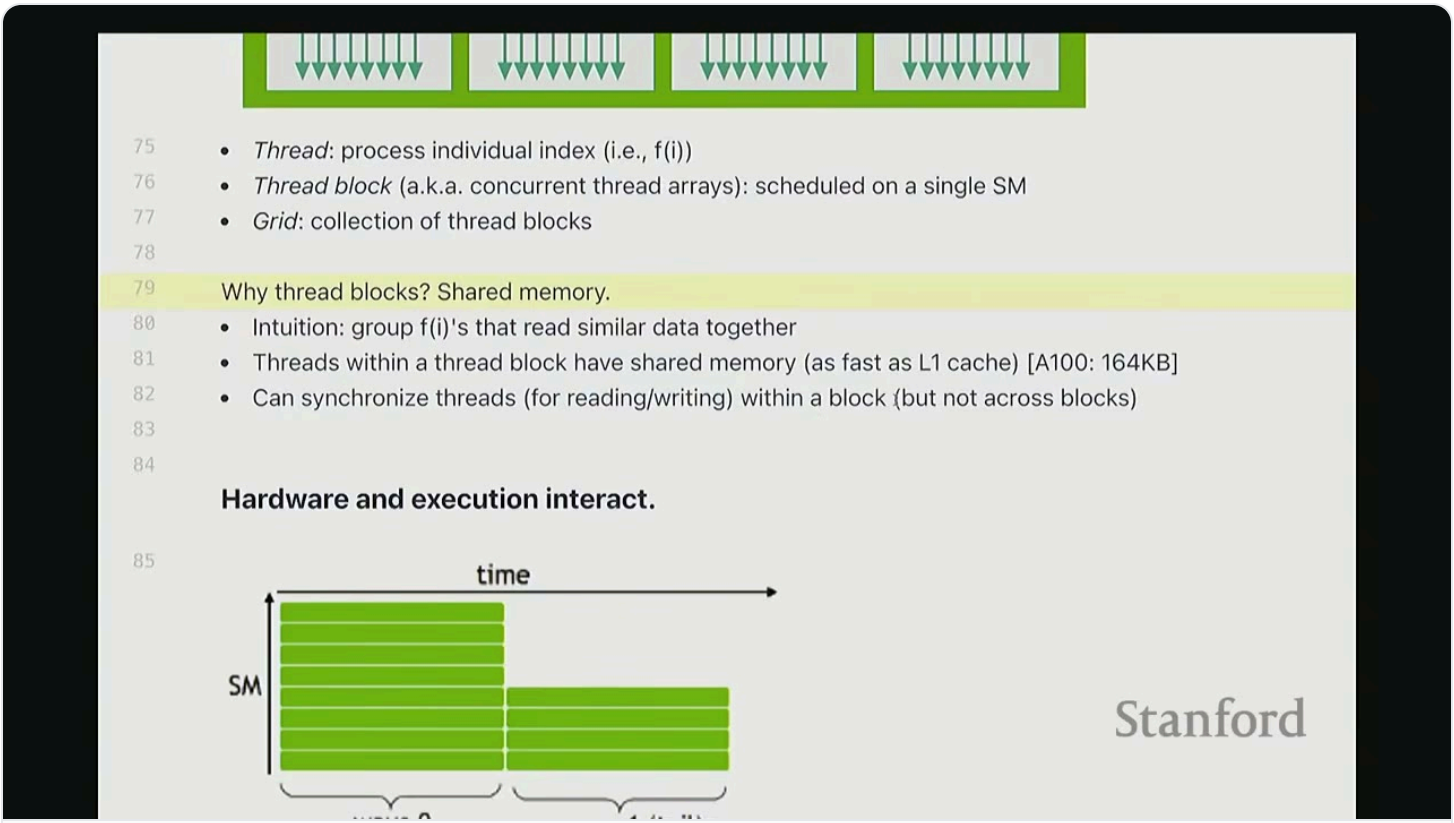
Challenge: Some aspects of hardware (e.g., scheduling, # SMs) are hidden from the execution model.

What is a Warp? A warp is a group of threads (typically 32) that are executed together. Warps exist to reduce the amount of control machinery needed, as a single control unit can manage 32 threads executing the same instruction. This allows for more compute units with simpler controls compared to CPUs.

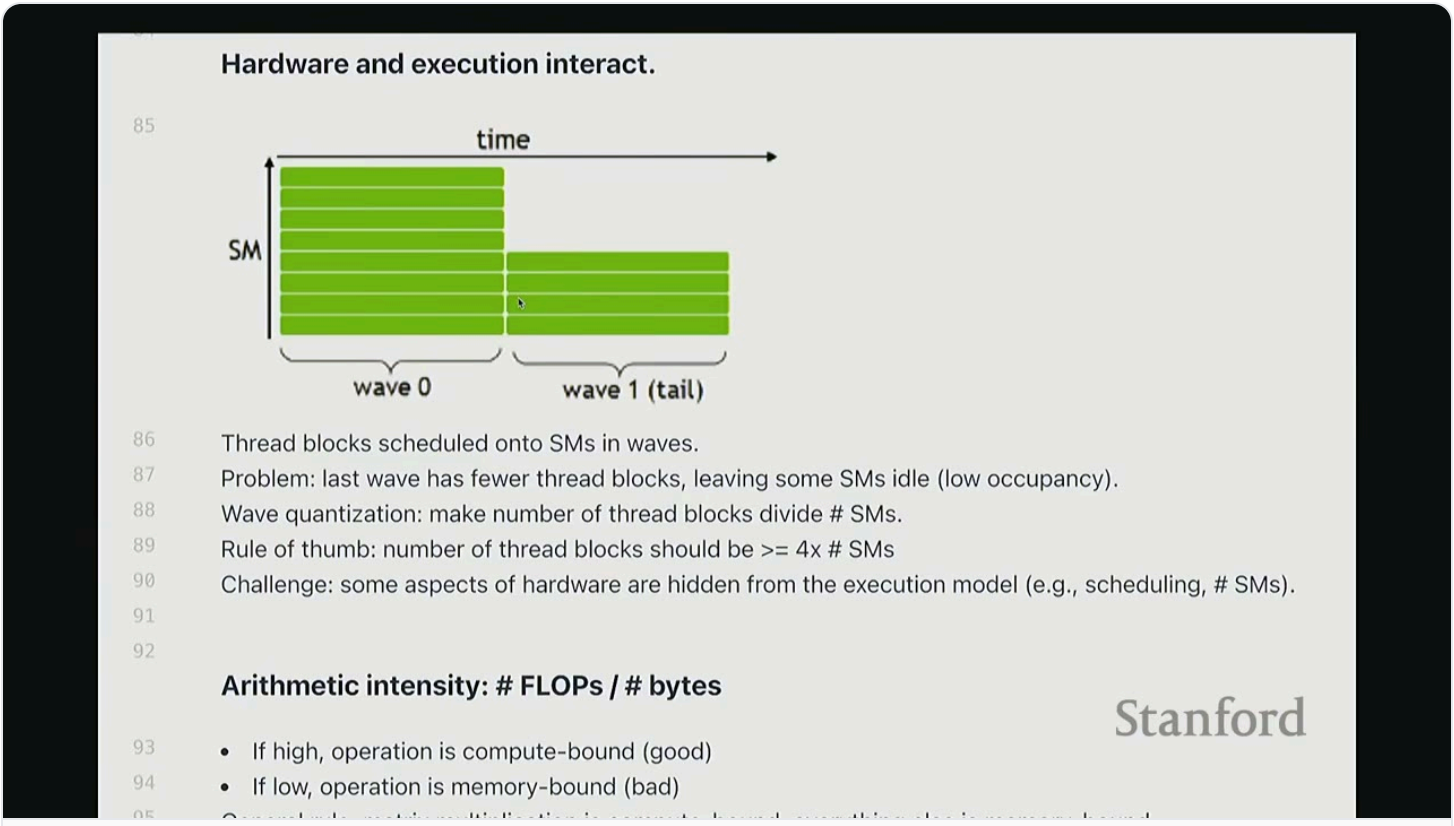
[0:00] Arithmetic Intensity: # FLOPs / # bytes



3:45 A slide showing a diagram of CTAs (Concurrent Thread Arrays) and definitions of thread, thread block, and grid.



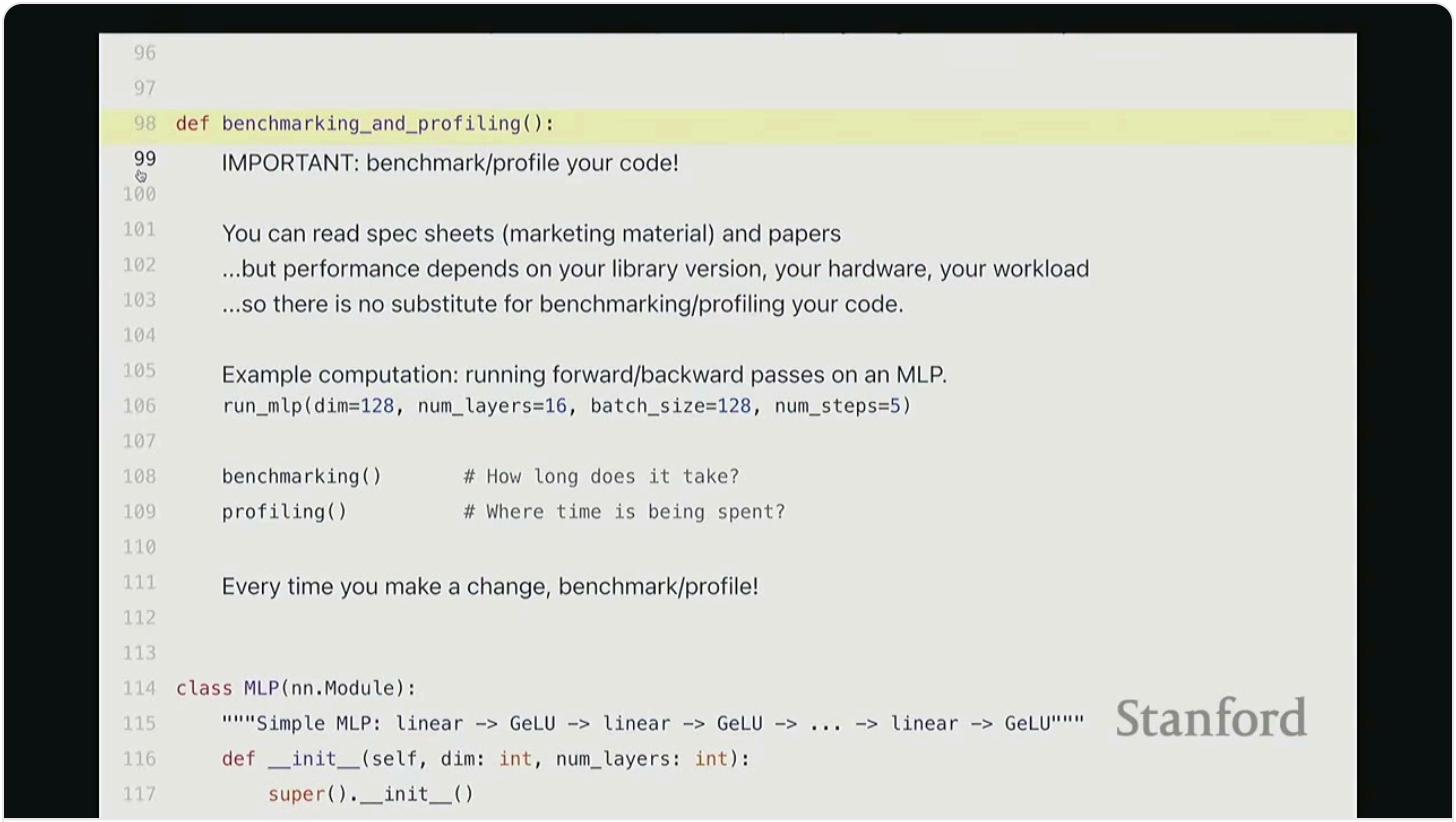
4:27 The slide highlights the section "Why thread blocks? Shared memory" explaining the intuition and benefits.



5:10 A diagram illustrating thread blocks scheduled onto SMs in waves, with text explaining wave quantization.

- **Definition:** The ratio of floating-point operations (FLOPs) to bytes transferred from/to memory.
- **Goal:** Keep arithmetic intensity high (more FLOPs per byte).
- **High Intensity:** Operation is compute-bound (good).
- **Low Intensity:** Operation is memory-bound (bad).
- **Reasoning:** Compute scaling is much faster than memory scaling. Many operations become memory-bound, meaning the GPU spends more time waiting for data than computing.
- **General Rule:** Matrix multiplication (if done cleverly) is compute-bound. Everything else is often memory-bound. We aim to reduce memory-bound operations.

[7:06] Benchmarking and Profiling



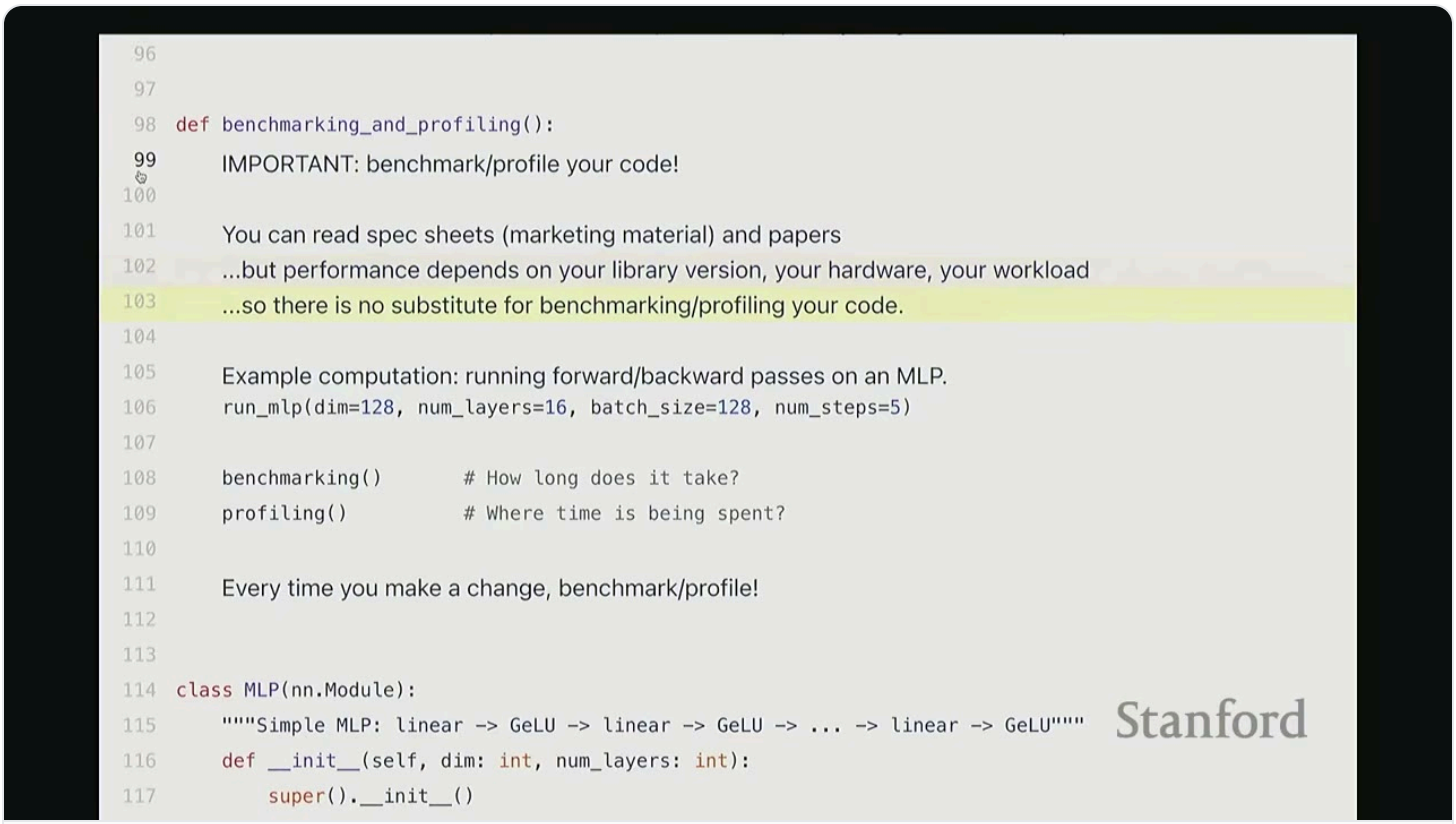
7:14 A code snippet defining `benchmarking_and_profiling()` and emphasizing the importance of benchmarking.

IMPORTANT: Benchmark/profile your code!

- You can read spec sheets (marketing material) and papers, but performance depends on your library version, your hardware, and your workload.
- There is no substitute for benchmarking/profiling your code.

- Every time you make a change, benchmark/profile!

[8:43] Example Computation: MLP



8:39 The slide highlights the text: "...so there is no substitute for benchmarking/profiling your code."

We'll use a simple Multi-Layer Perceptron (MLP) for demonstration. * `dim = 128` (dimension) * `num_layers = 16` * `batch_size = 128` * `num_steps = 5` (forward/backward passes)

```
class MLP(nn.Module):
    """Simple MLP: Linear -> GELU -> Linear -> GELU -> ... -> Linear"""
    def __init__(self, dim: int, num_layers: int):
        super().__init__()
        self.layers = nn.ModuleList(
            [nn.Linear(dim, dim) for _ in range(num_layers)]
        )

    def forward(self, x: torch.Tensor):
        for layer in self.layers:
            x = layer(x)
            x = torch.nn.functional.gelu(x)
        return x

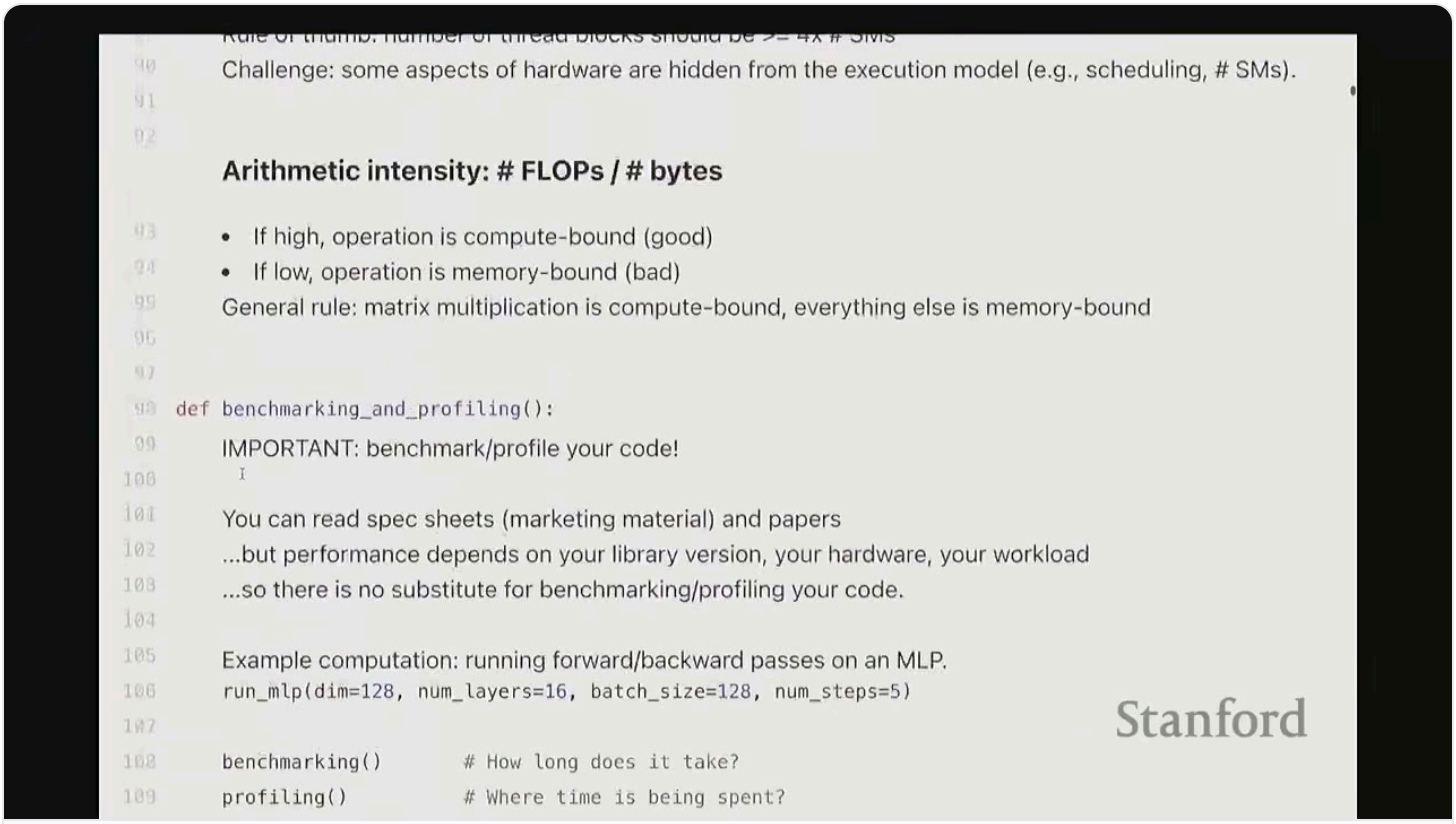
def run_mlp(dim: int, num_layers: int, batch_size: int, num_steps: int) -> Callable:
    # Define a model (with random weights)
    model = MLP(dim, num_layers).to(get_device())
```

```
    # Define an input (random)
    x = torch.randn(batch_size, dim, device=get_device())

    def run():
        # Run the model `num_steps` times (note: no optimizer updates)
        for step in range(num_steps):
            # Forward
            y = model(x)
            # Backward
            y.backward(y.mean())
        return y

    return run
```

[10:05] Benchmarking vs. Profiling

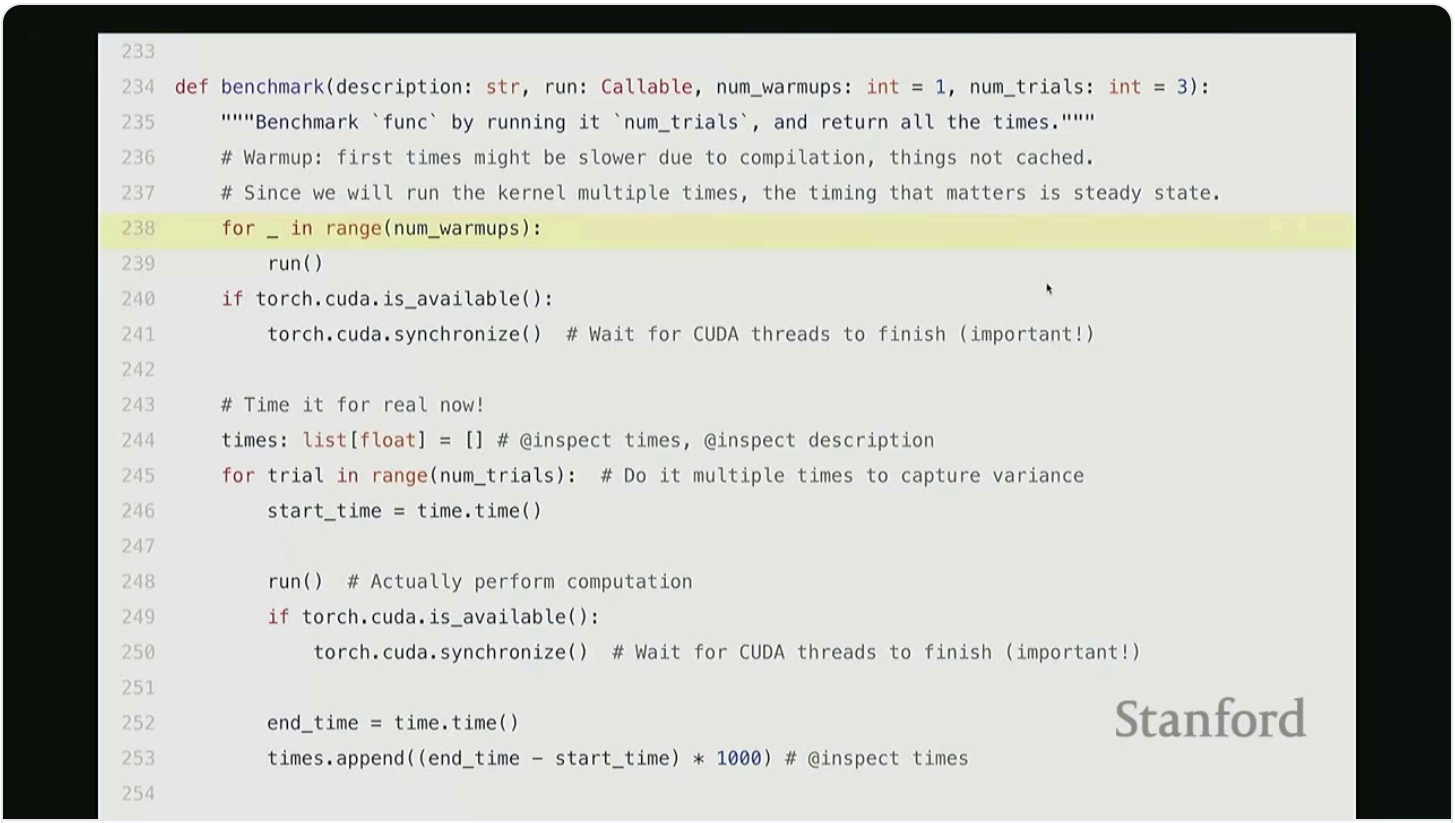


10:00 The slide shows the definition of "Arithmetic intensity: # FLOPs / # bytes" and its implications.

- **Benchmarking:** Measures the wall-clock time of performing some operation. It gives you end-to-end time, not where time is spent.
    - Useful for: comparing different implementations, understanding how performance scales (e.g., with dimension).

- **Profiling:** Helps you understand where time is being spent within a function. Deeper profiling helps you understand *what is being called* under the hood.

[11:09] Benchmarking Implementation



11:47 A Python code snippet for a `benchmark` function, including warm-up runs and timing trials.

```python
def benchmark(description: str, run: Callable, num_warmups: int = 1, num_trials: int = 3):
    """Benchmark `func` by running it `num_trials`, and compilation all the times."""
    # Warmup: first times might be slower due to compilation, things not cached.
    # Since we will run the kernel multiple times, the timing that matters is steady state.
    for _ in range(num_warmups):
        run()
    if torch.cuda.is_available():
        torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)

    # Time it for real now!
    times: list[float] = [] # @inspect times, @inspect description
    for trial in range(num_trials): # Do it multiple times to capture variance
        start_time = time.time()
        run() # Actually perform computation
        if torch.cuda.is_available():
            torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)
        end_time = time.time()
        times.append((end_time - start_time) * 1000) # @inspect times
```

```
    mean_time = mean(times) # @inspect mean_time
    return mean_time
```

Key Considerations for Benchmarking: 1. **Warm-up Iterations:** The first few runs of a GPU kernel are often slower due to JIT compilation, caching, and other initialization overheads. Warm-up runs ensure you measure the steady-state performance. 2. **`torch.cuda.synchronize()`:** The CPU and GPU operate asynchronously. When the CPU dispatches a CUDA kernel, it doesn't wait for the GPU to finish before executing subsequent CPU code. To get accurate end-to-end GPU execution time, you *must* call `torch.cuda.synchronize()` before and after the timed operation to ensure both devices are synchronized. Otherwise, your measured time will only reflect the CPU's dispatch time, not the actual GPU computation. 3. **Multiple Trials:** Run the operation multiple times and take the mean to account for variance (e.g., due to thermal throttling).

[14:34] Benchmarking Matrix Multiplication

We benchmark `torch.matmul` for various square matrix dimensions on an A100 GPU.

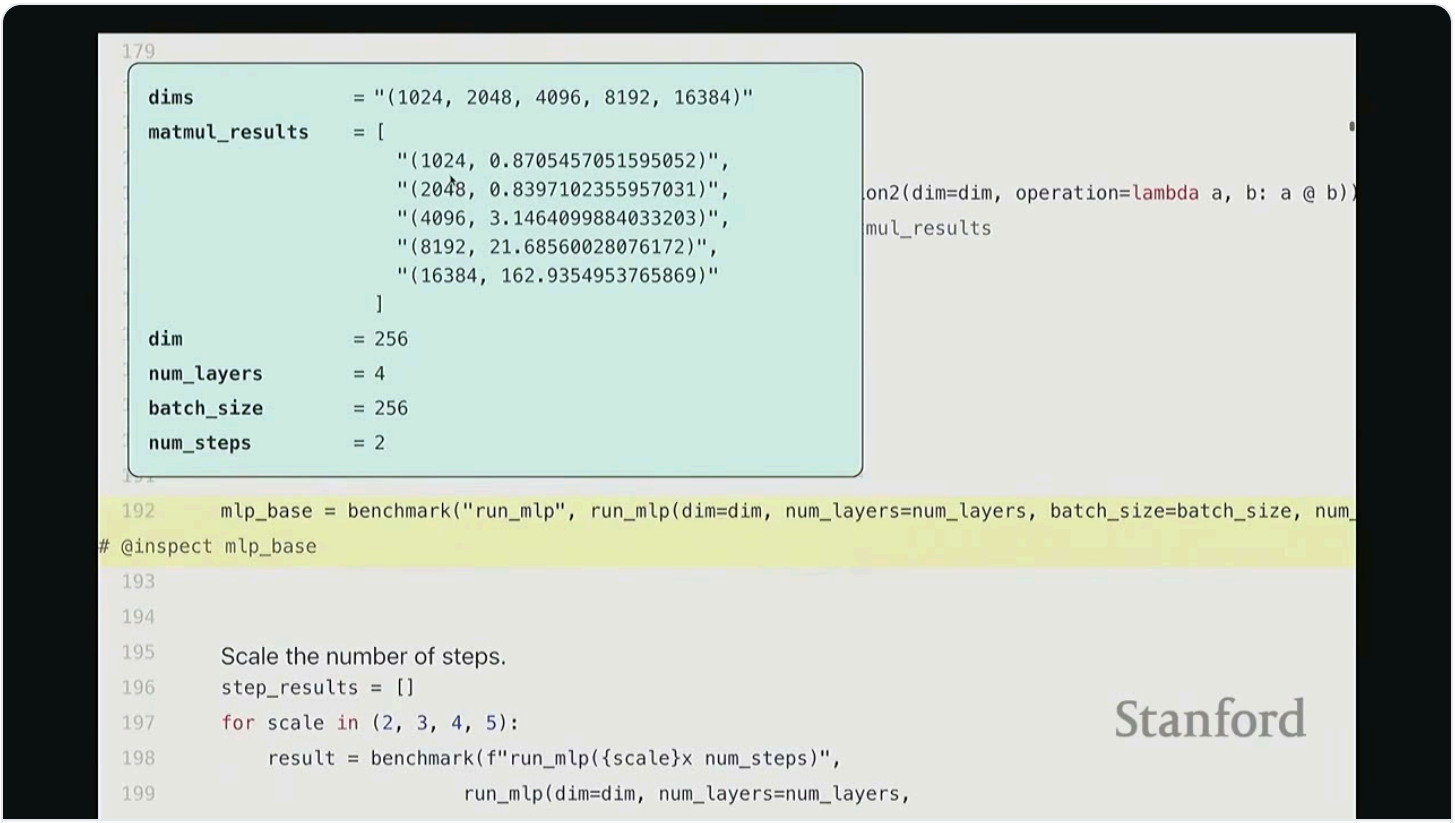
```
if torch.cuda.is_available():
    dims = ("1024", "2048", "4096", "8192", "16384")
    # dims = (1024, 2048, 4096, 8192, 16384) # @inspect dims
    # else:
    # dims = (1024, 2048) # @inspect dims

    matmul_results = []
    for dim in dims:
        # result = benchmark(f"matmul(dim={dim})", run_operation2(dim=dim, operation=lambda a, b
        # matmul_results.append(result)
        # @inspect matmul_results
        pass # Code executed in the video, results shown below
```

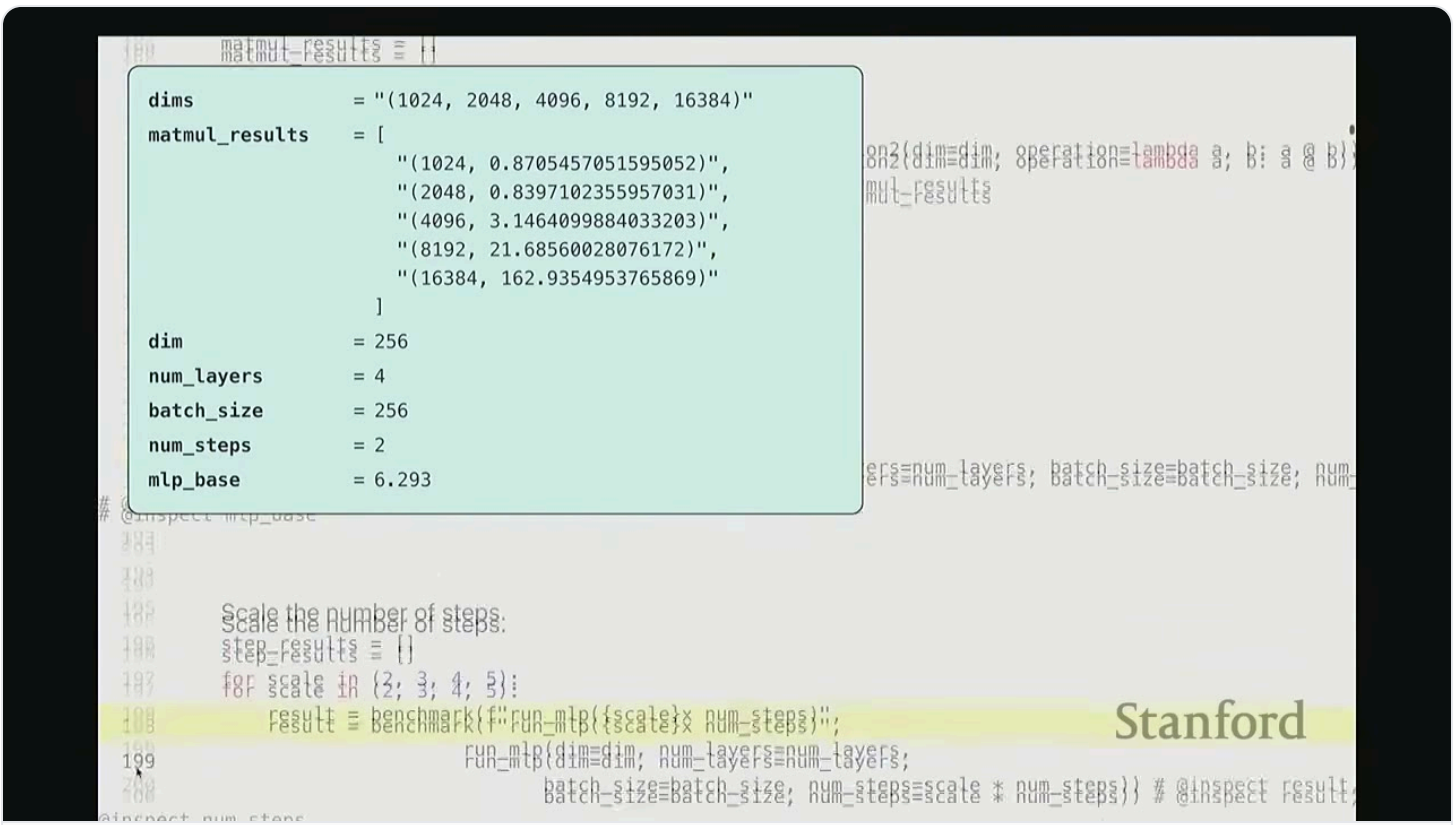
Results (approximate values from video): * (1024, 0.70ms) * (2048, 0.83ms) * (4096, 3.14ms) * (8192, 21.68ms) * (16384, 162.93ms)

Observations: * For very small matrices (e.g., 1024x1024, 2048x2048), the runtime doesn't grow much. This is due to constant factor overheads like launching the kernel and data transfer between CPU and GPU. * For larger matrices, we observe super-linear scaling, as expected for matrix multiplication ($O(N^3)$ operations).

[15:40] Benchmarking MLP Scaling



15:55 A code snippet showing `matmut_results` and parameters for an MLP, with a call to `benchmark`.



16:40 The code snippet highlights the line where `result` is assigned within a loop to scale the number of steps.

We benchmark our MLP by scaling the number of steps and layers.

```
# Let us benchmark our MLP!
dim = 256 # @inspect dim
num_layers = 4 # @inspect num_layers
batch_size = 256 # @inspect batch_size
num_steps = 2 # @inspect num_steps

mlp_base = benchmark("run_mlp", run_mlp(dim, num_layers, batch_size, num_steps))
# @inspect mlp_base # Result: 6.293 ms

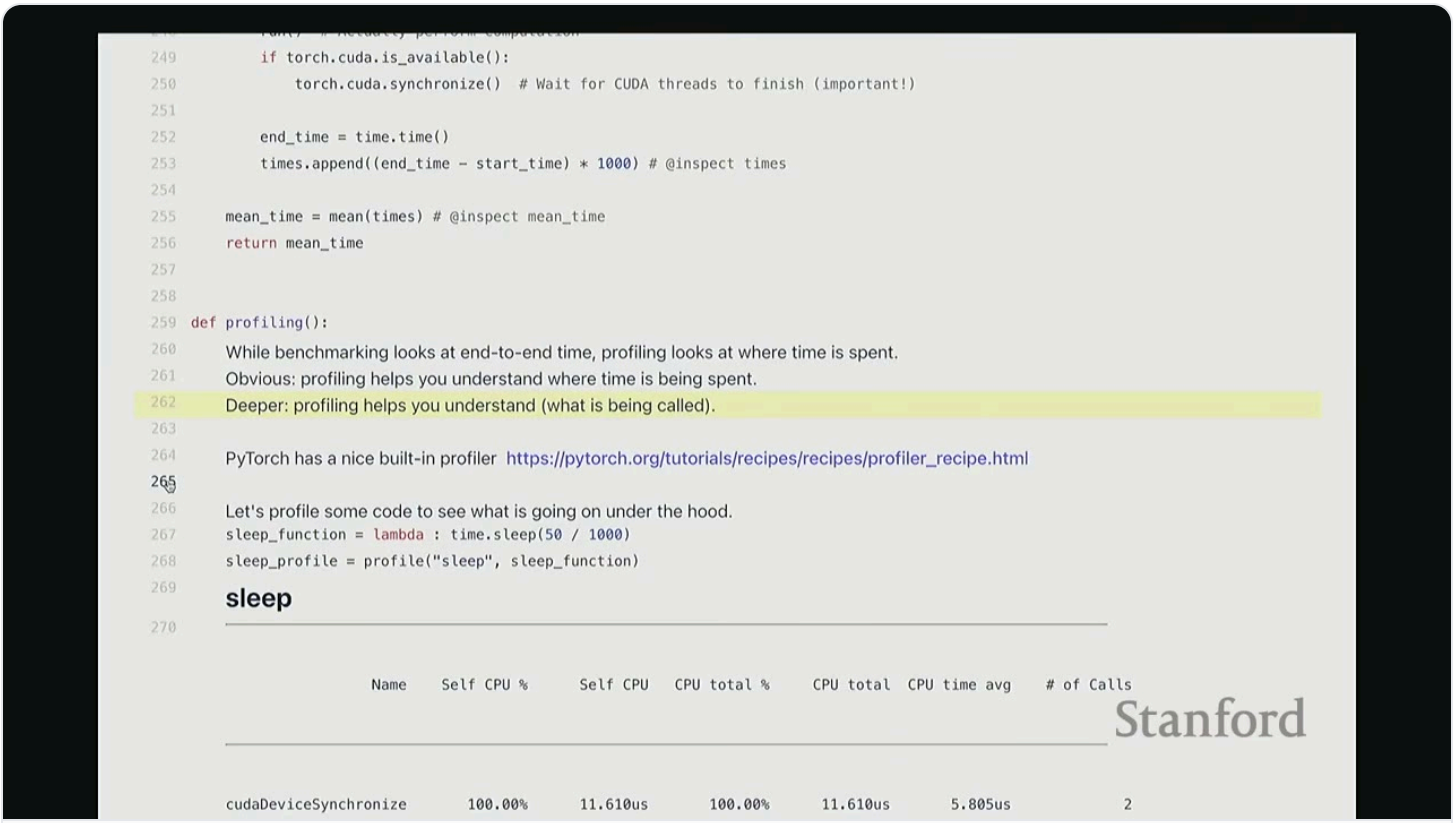
# Scale the number of steps.
step_results = []
for scale in (2, 3, 4, 5):
    # result = benchmark(f"run_mlp(scale={scale} x num_steps)", run_mlp(dim, num_layers, batch_s
    # step_results.append((scale, result)) # @inspect step_results
    pass # Code executed in the video, results shown below

# Scale the number of layers.
layer_results = []
for scale in (2, 3, 4, 5):
    # result = benchmark(f"run_mlp(scale={scale} x num_layers)", run_mlp(dim, scale * num_layers
    # layer_results.append((scale, result)) # @inspect layer_results
    pass # Code executed in the video, results shown below
```

Results (approximate values from video): * **Base MLP (2 steps, 4 layers):** 6.293 ms * **Scaling Steps:** * (2, 11.326 ms) * (3, 16.761 ms) * (4, 22.028 ms) * (5, 27.310 ms) * *Observation:* Linear scaling with `num_steps`, approximately 5ms per step. * **Scaling Layers:** * (2, 9.783 ms) * (3, 13.274 ms) * (4, 16.856 ms) * (5, 20.847 ms) * *Observation:* Linear scaling with `num_layers`, approximately 4ms per layer.

Conclusion: For both `num_steps` and `num_layers`, the runtime scales linearly, as expected.

[17:46] Profiling Implementation



18:35 A slide defining profiling and highlighting the text: "profiling helps you understand (what is being called)."

```python
def profiling(description: str, run: Callable, num_warmups: int = 1, with_stack: bool = False):
    """Profile `run` by running it `num_warmups` times, and print a table."""
    # Warmup
    for _ in range(num_warmups):
        run()
    if torch.cuda.is_available():
        torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)

    # Run the code with the profiler
    activities = [ProfilerActivity.CPU, ProfilerActivity.CUDA]
    if with_stack:
        # Output stack trace for visualization
        # Needed to export stack trace for visualization
        # experimental_config=torch.profiler.ExperimentalConfig(verbose=True) as prof:
        pass # Simplified for brevity

    with torch.profiler.profile(activities=activities, with_stack=with_stack) as prof:
        run()
    if torch.cuda.is_available():
        torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)

    # Print out table
    print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=10, max_name_width=80))
    # Write stack trace visualization
```

```
    # prof.export_stack_trace(f"{description}.txt")
    return prof
```

Key Benefits of Profiling: * **Deeper Insight:** Helps understand *what is being called* under the hood, not just end-to-end time. * **PyTorch's Built-in Profiler:** `torch.profiler` is a convenient tool that allows you to see low-level CUDA calls without leaving the Python environment.

[19:02] Profiling `sleep` function

```
sleep_function = lambda : time.sleep(50 / 1000)
sleep_profile = profiling("sleep", sleep_function)
```

Profiling Output for `sleep`:

```
Name                 Self CPU %      Self CPU      CPU total %     CPU total      CUDA time %
--------------------  ----------  -------------  -------------  -------------  -------------  ----
cudadeviceSynchronize    100.00%      11.610us        100.00%      11.610us        100.00%
--------------------  ----------  -------------  -------------  -------------  -------------  ----
Self CPU time total: 11.610us
```

Observation: 100% of the time is spent in `cudadeviceSynchronize` because `time.sleep` is a CPU-only operation, and the GPU is idle.

[19:53] Profiling `add` (Element-wise addition)

```
add_function = lambda a, b: a + b
add_profile = profiling("add", run_operation2(dim=2048, operation=add_function))
```

Profiling Output for `add` (2048x2048 matrices):

```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------
aten::add                                                               98.02%       1.39ms
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_add_f...    0.00%        0.00
cudaLaunchKernel                                                        0.00%       0.00us
cudadeviceSynchronize                                                   0.00%       0.00us
```

```
-------------------------------------------------------------------  ----------  ------------
Self CPU time total: 1.420ms, Self CUDA time total: 17.02us
```

Observations: * The Python `a + b` call (`aten::add`) takes 1.39ms on the CPU. * This dispatches to a specific CUDA kernel (`vectorized_elementwise_kernel`) which takes 16.90us on the GPU. * `cudaLaunchKernel` (CPU sending command to GPU) takes 0.12us. * `cudadeviceSynchronize` takes negligible time here. * The CPU time is dominated by the `aten::add` wrapper, indicating overhead in dispatching to the GPU.

[22:21] Profiling `matmul` (Matrix Multiplication)

```
matmul_function = lambda a, b: a @ b
matmul_profile = profiling("matmul", run_operation2(dim=2048, operation=matmul_function))
```

Profiling Output for `matmul` (2048x2048 matrices):

```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------
aten::matmul                                                            99.20%       7.23ms
void at::native::cutlass::kernel_2048_2048_2048_stage2_warp_size_64...     0.00%       0.00us
cudaLaunchKernel                                                         0.00%       0.00us
cudadeviceSynchronize                                                    0.00%       0.00us
-------------------------------------------------------------------  ----------  ------------
Self CPU time total: 7.280ms, Self CUDA time total: 343.14us
```

Observations: * Similar pattern to `add` , but the GPU kernel (`cutlass::kernel...`) takes significantly longer (342.62us) than for `add` , as expected for matrix multiplication. * The `cutlass` library is NVIDIA's high-performance matrix multiply library.

[25:46] Profiling `matmul` (different dimension)

```
matmul_profile_dim128 = profiling("matmul(dim=128)", run_operation2(dim=128, operation=matmul_f
```

Profiling Output for `matmul` (128x128 matrices):

```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------
aten::matmul                                                            99.20%       7.23ms
void at::native::xmma_gemm_f32f32_f32_t_t_t_stage2_warp_size_64...       0.00%       0.00us
cudaLaunchKernel                                                         0.00%       0.00us
cudadeviceSynchronize                                                    0.00%       0.00us
-------------------------------------------------------------------  ----------  ------------
Self CPU time total: 7.280ms, Self CUDA time total: 343.14us
```

Observations: * For a smaller 128x128 matrix, PyTorch dispatches to a *different* CUDA kernel (`xmma_gemm...`). This kernel is optimized for smaller dimensions. * This illustrates that the specific CUDA kernels called depend on the tensor dimensions and underlying hardware.

[27:30] Profiling `cdist` (Pairwise Euclidean Distance)

```
cdist_profile = profiling("cdist", run_operation2(dim=2048, operation=torch.cdist))
```

Profiling Output for `cdist` (2048x2048 matrices):

```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------
aten::cdist                                                             27.40%       1.39ms
aten::euclidean_dist                                                     0.00%       0.00us
aten::max                                                                0.00%       0.00us
aten::sum                                                                0.00%       0.00us
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_add_f...   0.00%       0.00u
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_mul_f...   0.00%       0.00u
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_pow_f...   0.00%       0.00u
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_sqrt_f...  0.00%       0.00u
-------------------------------------------------------------------  ----------  ------------
Self CPU time total: 1.39ms, Self CUDA time total: 343.14us
```

Observations: * `cdist` is a more complex operation, requiring dot products, square roots, etc. * The PyTorch `torch.cdist` call maps to `aten::euclidean_dist` in the C++ interface. * This then decomposes into multiple CUDA kernels: `aten::matmul` (for dot products), `aten::pow` (for squaring), `aten::sqrt` (for square root), `aten::sum` (for reduction), and `aten::max` (for stability). * The `aten::matmul` operation consumes the most GPU time (78%), followed by `aten::pow` (6%), and `aten::sum` (3%). * This detailed breakdown helps identify where optimization efforts would be most impactful (e.g., optimizing matrix multiplication).

[30:27] Profiling `gelu` and `softmax`

```
gelu_function = lambda a, b: torch.nn.functional.gelu(a + b)
gelu_profile = profiling("gelu", run_operation2(dim=2048, operation=gelu_function))

softmax_function = lambda x: torch.nn.functional.softmax(x, dim=-1)
softmax_profile = profiling("softmax", run_operation2(dim=2048, operation=softmax_function))
```

Profiling Output for `gelu`:

```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------
aten::add                                                               86.27%       1.42ms
aten::gelu                                                               0.00%       0.00us
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_add_f...    0.00%          0.00u
cudaLaunchKernel                                                         0.00%       0.00us
cudadeviceSynchronize                                                    0.00%       0.00us
-------------------------------------------------------------------  ----------  ------------
Self CPU time total: 1.42ms, Self CUDA time total: 55.14us
```

Observations: * `gelu` involves `aten::add` and `aten::gelu`. * The `aten::gelu` operation itself is a fused CUDA kernel, consuming 33% of GPU time. The `aten::add` operation consumes 66% of GPU time. * Many core primitives like `gelu` and `softmax` have dedicated fused CUDA kernels, meaning there's no back-and-forth between CPU and GPU for their internal components.

[32:50] Understanding CPU-GPU Interaction with Nsight Systems

To understand the CPU's role and why it appears to "run ahead" of the GPU, we use NVIDIA Nsight Systems.

- Nsight Systems provides a detailed timeline view of CPU and GPU activities.
- **CPU (Threads):** The lower part of the timeline shows CPU threads.
- **CUDA HW:** The upper part shows GPU hardware activity.

[36:30] Asynchronous Execution: * When the CPU calls a PyTorch operation, it dispatches CUDA kernels to the GPU. * The CPU does *not* wait for the GPU to finish; it continues executing subsequent Python code. * This creates a queue of CUDA commands on the GPU. * The CPU can "run ahead" of the GPU, dispatching many kernels before the GPU even starts executing the first

one. * **Example:** In our MLP, the CPU might be processing `layer_9` while the GPU is still working on `layer_1`.

[39:10] Impact of `print` Statements: * A `print` statement on the CPU that requires a GPU value (e.g., printing the loss after a backward pass) forces a `torch.cuda.synchronize()`. * This synchronization makes the CPU wait for the GPU to complete all pending operations up to that point. * **Observation:** In the Nsight Systems timeline, a `print` statement causes the CPU's timeline to pause until the corresponding GPU operations are finished. This effectively "synchronizes" the CPU and GPU at that point, preventing the CPU from running far ahead. * While this might not always lead to a CPU bottleneck if the GPU is still fully utilized, it can significantly alter the execution flow and potentially introduce stalls if the CPU has to wait frequently for GPU results. This is why `torch.cuda.synchronize()` is crucial for accurate benchmarking.

[44:24] Kernel Fusion Motivation

- **Analogy:** Memory (DRAM) is a warehouse, Compute (SMs + SRAM) is a factory. Data transfer between them incurs "bandwidth cost".
- **Naive Approach:** Performing operations sequentially means frequently moving data from DRAM to SM, computing, and then writing back to DRAM for each small operation. This incurs high bandwidth cost.
- **Kernel Fusion:** Combine multiple operations into a single kernel. This allows data to be loaded from DRAM to SM once, all computations to be performed on-chip, and the final result written back to DRAM only once. This drastically reduces bandwidth cost.

[45:57] Writing CUDA Kernels in C++

Goal: Write a custom GLU kernel in CUDA C++ to achieve better performance than the naive PyTorch implementation.

PyTorch GLU (Naive Implementation):

```
y1 = torch.nn.functional.gelu(x)
# This is an approximation of the true GELU for easier comparison
y2 = 0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * x**3)))
```

- Benchmarking shows PyTorch's native `gelu` is much faster (1.173 ms) than our manual Python implementation (8.162 ms). This is because PyTorch's `gelu` is a fused CUDA kernel.

CUDA C++ `gelu_kernel` Structure: * CUDA C++ is an extension of C++ with APIs for managing GPUs. * **Simplified Picture:** Write a `__global__` function (the kernel) that executes on the GPU. * **Grid:** Collection of thread blocks (`num_blocks`). * **Thread Block:** Collection of threads (`block_size`). * **Thread:** Single unit of operation.

[48:45] CUDA C++ `gelu_kernel` Code:

```
// CUDA code has the full logic
cuda_gelu_src = open("gelu.cu").read()
// #include <math.h> #include <torch/extension.h> #include <c10/cuda/CUDAException.h>
// __global__ void gelu_kernel(float* in, float* out, int num_elements) {
//     // Get the index into the tensor
//     int i = blockIdx.x * blockDim.x + threadIdx.x;
//     // To handle the case when n < num_blocks * blockDim
//     if (i < num_elements) {
//         // Do the actual computation
//         out[i] = 0.5 * in[i] * (1.0 + tanh(0.79788456 * (in[i] + 0.044715 * in[i] * in[i] * 
//     }
// }

// inline unsigned int cdiv(unsigned int a, unsigned int b) {
//     return (a + b - 1) / b;
// }

// torch::Tensor gelu(torch::Tensor x) {
//     TORCH_CHECK(x.device().is_cuda(), "x.device().is_cuda() is false");
//     TORCH_CHECK(x.is_contiguous(), "x.is_contiguous() is false");

//     // Allocate empty tensor
//     torch::Tensor y = torch::empty_like(x);

//     // Determine grid (elements divided into blocks)
//     int num_elements = x.numel();
//     int block_size = 1024; // Number of threads
//     int num_blocks = cdiv(num_elements, block_size);

//     // Launch the kernel
//     gelu_kernel<<<num_blocks, block_size>>>(x.data_ptr<float>(), y.data_ptr<float>(), num_el
//     C10_CUDA_KERNEL_LAUNCH_CHECK(); // Catch errors immediately

//     return y;
// }

// C++ code defines the gelu function
// cpp_gelu_src = "torch::Tensor gelu(torch::Tensor x);"
```

Explanation of CUDA C++ Code:

1. `__global__ void gelu_kernel(...)` : This is the actual GPU kernel function.
    - `float* in, float* out` : Pointers to input and output data in global memory.
    - `int num_elements` : Total number of elements to process.
    - **Index Calculation:** `int i = blockIdx.x * blockDim.x + threadIdx.x;` This formula calculates the global index `i` that the current thread is responsible for.
        - `blockIdx.x` : The ID of the current thread block (0 to `num_blocks` -1).
        - `blockDim.x` : The number of threads per block (`block_size`).
        - `threadIdx.x` : The ID of the current thread within its block (0 to `block_size` -1).
    - **Boundary Check:** `if (i < num_elements)` : Essential for CUDA. Threads might be launched beyond the actual data size (due to rounding up `num_blocks`). This check prevents out-of-bounds memory access.
    - **Actual Computation:** `out[i] = 0.5 * in[i] * (1.0 + tanh(0.79788456 * (in[i] + 0.044715 * in[i] * in[i] * in[i])));` This is the GLU formula, applied element-wise.

2. `torch::Tensor gelu(torch::Tensor x)` **(Wrapper Function):** This C++ function acts as a wrapper, callable from Python, to launch the CUDA kernel.
    - **Input Checks:**
        - `TORCH_CHECK(x.device().is_cuda(), ...)` : Ensures the input tensor `x` is on the GPU.
        - `TORCH_CHECK(x.is_contiguous(), ...)` : Ensures `x` is stored contiguously in memory. This is important because the kernel assumes linear indexing. If not contiguous, the indexing arithmetic would be incorrect.
    - **Allocate Output:** `torch::Tensor y = torch::empty_like(x);` Allocates GPU memory for the output tensor `y` . Using `empty_like` avoids initializing memory, saving operations.
    - **Grid Determination:**
        - `num_elements = x.numel();`
        - `block_size = 1024;` (A common choice for number of threads per block)
        - `num_blocks = cdiv(num_elements, block_size);` Calculates the number of thread blocks needed, using a ceiling division helper `cdiv` .
    - **Kernel Launch:** `gelu_kernel<<<num_blocks, block_size>>>(x.data_ptr<float>(), y.data_ptr<float>(), num_elements);` This is the CUDA kernel launch syntax.

- `<<<num_blocks, block_size>>>` : Specifies the grid and block dimensions.
- `x.data_ptr<float>()` , `y.data_ptr<float>()` : Pass raw pointers to the tensor data.
- `C10_CUDA_KERNEL_LAUNCH_CHECK();` : A PyTorch utility to check for CUDA errors immediately after launch.

[56:45] Compiling and Binding CUDA Code to Python: PyTorch's `torch.utils.cpp_extension.load_inline` function allows compiling C++ (including CUDA C++) code and binding it to a Python module dynamically.

```
# Compile the CUDA code and bind it to a Python module.
# ensure_directory_exists("var/cuda_gelu")
# if not torch.cuda.is_available():
#     return None
# module = load_inline(
#     cuda_sources=[cuda_gelu_src],
#     cpp_sources=[cpp_gelu_src],
#     functions=["gelu"],
#     extra_cflags=["-O2"],
#     verbose=True,
#     name="inline_gelu",
#     build_directory="var/cuda_gelu",
# )
# cuda_gelu = getattr(module, "gelu")
```

This process is very convenient, as it handles compilation and linking within Python.

[57:16] Benchmarking Custom CUDA `gelu` : We benchmark our custom CUDA `gelu` against the manual Python and native PyTorch `gelu` .

```
# Check correctness of our implementation.
# check_equal(cuda_gelu(x), manual_gelu(x))

# Benchmark our CUDA version.
# manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu)) # @i
# pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu)) #
# cuda_time = benchmark("cuda_gelu", run_operation1(dim=16384, operation=cuda_gelu)) # @inspect
```

Results (approximate values from video): * `manual_time` : 8.162 ms * `pytorch_time` : 1.173 ms * `cuda_time` : 1.845 ms

Observations: * Our custom CUDA `gelu` (1.845 ms) is significantly faster than the naive Python implementation (8.162 ms). * It's not quite as fast as PyTorch's native `gelu` (1.173 ms), but it's much closer. This shows the power of kernel fusion.

[57:55] Profiling Custom CUDA `gelu`:

```
cuda_gelu_profile = profiling("cuda_gelu", run_operation1(dim=16384, operation=cuda_gelu))
```

Profiling Output for Custom CUDA `gelu`:

```
Name                  Self CPU %      Self CPU      CPU total %     CPU total      CUDA time %
--------------------  ----------  ------------  -------------  ------------  -------------  ----
gelu_kernel(float*,...     0.00%        0.00us          0.00%        0.00us        100.00%      1
aten::empty_like           0.20%        6.20us          0.20%        6.20us          0.00%
aten::empty_strided       46.31%        1.42ms         46.31%       1.42ms           0.00%
cudaLaunchKernel           8.93%      274.07us          8.93%      274.07us          0.00%
cudadeviceSynchronize     44.56%        1.36ms         44.56%       1.36ms           0.00%
--------------------  ----------  ------------  -------------  ------------  -------------  ----
Self CPU time total: 3.070ms, Self CUDA time total: 1.664ms
```

Observations: * The `gelu_kernel` itself consumes 100% of the GPU time (1.664ms). * The CPU time is dominated by `aten::empty_strided` (allocating output) and `cudadeviceSynchronize` (waiting for GPU). * This confirms that our custom CUDA kernel is a single, fused operation.

[58:37] Element-wise Operations vs. Reductions

- Element-wise operations are relatively easy to write in CUDA (like our `gelu`).
- More interesting operations (e.g., `softmax`, `norm`, `matmul`) require reading multiple values and performing reductions, which are more complex to optimize.

[59:03] Triton Kernels in Python

Motivation: CUDA C++ is powerful but can be complex. Triton offers a Python-based DSL for GPU programming that is more accessible.

Triton Advantages: * Write GPU programming in Python. * Think about thread blocks rather than individual threads. * **Automatic Management:** * Memory coalescing (transfer from DRAM): Triton automatically groups memory accesses for efficiency. * Shared memory management: Triton handles allocating and managing shared memory within SMs. * Scheduling within SMs: Triton automatically schedules threads within an SM. * **Manual Management:** * Scheduling across SMs: This remains a manual decision for the programmer (e.g., how many blocks to launch). * **Compiler Power:** Triton's compiler can often outperform PyTorch's native implementations by generating highly optimized kernels. * **Debuggability:** Since it's Python, you can step through the code and debug it more easily.

[1:04:10] Triton `gelu` Kernel Code:

```
# Triton gelu_kernel
@triton.jit
def triton_gelu_kernel(x_ptr, y_ptr, num_elements, BLOCK_SIZE: tl.constexpr):
    # Input is at `x_ptr` and output is at `y_ptr`
    # BLOCK_0 | BLOCK_1 | ... | num_elements
    # Indices where this thread block should operate
    pid = tl.program_id(axis=0) # block_id
    block_start = pid * BLOCK_SIZE

    # Offsets = block_start + tl.arange(0, BLOCK_SIZE)
    offsets = block_start + tl.arange(0, BLOCK_SIZE)

    # Handle boundary
    mask = offsets < num_elements

    # Read
    x = tl.load(x_ptr + offsets, mask=mask)

    # Approx gelu is 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 * x^3)))
    # Compute (tl.tanh doesn't exist, use tanh(a) = (exp(2a) - 1) / (exp(2a) + 1))
    a = 0.79788456 * (x + 0.044715 * x * x * x)
    exp_2a = tl.exp(2 * a)
    tanh_a = (exp_2a - 1) / (exp_2a + 1)
    y = 0.5 * x * (1 + tanh_a)

    # Store
    tl.store(y_ptr + offsets, y, mask=mask)
```

Explanation of Triton `gelu` Code:

1. **`@triton.jit`** : Decorator indicating this is a Triton JIT-compiled kernel.
2. **`triton_gelu_kernel(...)`** : The kernel function.

    - `x_ptr, y_ptr` : Pointers to input/output data.
    - `num_elements` : Total elements.
    - `BLOCK_SIZE: tl.constexpr` : Block size, a compile-time constant.
3. **Index Calculation:** `pid = tl.program_id(axis=0)` gets the block ID. `block_start = pid * BLOCK_SIZE` calculates the starting index for the block.
4. **Offsets (Vectorized):** `offsets = block_start + tl.arange(0, BLOCK_SIZE)` creates a *vector* of indices for all threads within the block. Triton operates on blocks, not individual threads.
5. **Boundary Mask:** `mask = offsets < num_elements` creates a boolean mask to handle out-of-bounds access.
6. **Read:** `x = tl.load(x_ptr + offsets, mask=mask)` loads a *vector* of values from global memory into registers, applying the mask.
7. **Computation:** The GLU formula is applied to the *vector* `x` . Triton provides `tl.exp` for exponentiation. `tl.tanh` is implemented manually using `exp` because it's not directly available in Triton.
8. **Store:** `tl.store(y_ptr + offsets, y, mask=mask)` stores the *vector* of results back to global memory.

[1:07:53] PTX Code Generated by Triton: Triton compiles to PTX (Parallel Thread Execution) code, which is like assembly for GPUs.

```
// Generated by LLVM NVPX Back-End v8.4.target sm_90a.address_size 64
.version 8.4
.target sm_90a
.address_size 64

.visible .entry triton_gelu_kernel (
    .param .u64 triton_gelu_kernel_param_0,
    .param .u64 triton_gelu_kernel_param_1,
    .param .u32 triton_gelu_kernel_param_2
)
{
    .reg .pred %p<5>;
    .reg .b32 %r<49>;
    .reg .f32 %f<113>;
    .reg .b64 %rd<8>;

    .loc 1 552 0
    $L_func_begin0:
    .loc 1 552 0
```

```
    // %bb.0:
    ld.param.u64 %rd5, [triton_gelu_kernel_param_0];
    ld.param.u64 %rd6, [triton_gelu_kernel_param_1];
    .loc 1 557 24
    mov.u32 %r2, 0;
    mov.u32 %r3, 0;
    mov.u32 %r4, 0;
    mov.u32 %r5, 0;
    mov.u32 %r6, 0;
    mov.u32 %r7, 0;
    mov.u32 %r8, 0;
    mov.u32 %r9, 0;
    mov.u32 %r10, 0;
    mov.u32 %r11, 0;
    mov.u32 %r12, 0;
    mov.u32 %r13, 0;
    mov.u32 %r14, 0;
    mov.u32 %r15, 0;
    mov.u32 %r16, 0;
    mov.u32 %r17, 0;
    mov.u32 %r18, 0;
    mov.u32 %r19, 0;
    mov.u32 %r20, 0;
    mov.u32 %r21, 0;
    mov.u32 %r22, 0;
    mov.u32 %r23, 0;
    mov.u32 %r24, 0;
    mov.u32 %r25, 0;
    mov.u32 %r26, 0;
    mov.u32 %r27, 0;
    mov.u32 %r28, 0;
    mov.u32 %r29, 0;
    mov.u32 %r30, 0;
    mov.u32 %r31, 0;
    mov.u32 %r32, 0;
    mov.u32 %r33, 0;
    mov.u32 %r34, 0;
    mov.u32 %r35, 0;
    mov.u32 %r36, 0;
    mov.u32 %r37, 0;
    mov.u32 %r38, 0;
    mov.u32 %r39, 0;
    mov.u32 %r40, 0;
    mov.u32 %r41, 0;
    mov.u32 %r42, 0;
    mov.u32 %r43, 0;
    mov.u32 %r44, 0;
```

```
    mov.u32 %r45, 0;
    mov.u32 %r46, 0;
    mov.u32 %r47, 0;
    mov.u32 %r48, 0;
    .loc 1 558 24
    // begin inline asm
    mov.u32 %r2, %ctaid.x;
    // end inline asm
    .loc 1 561 41
    shl.b32 %r2, %r2, 10;
    mov.u32 %r3, %tid.x;
    ld.param.u32 %r4, [triton_gelu_kernel_param_2];
    mov.u32 %r5, %r2;
    add.u32 %r5, %r5, %r3;
    setp.lt.s32 %p1, %r5, %r4;
    // begin inline asm
    mov.u32 %r6, 0;
    mov.u32 %r7, 0;
    mov.u32 %r8, 0;
    mov.u32 %r9, 0;
    mov.u32 %r10, 0;
    mov.u32 %r11, 0;
    mov.u32 %r12, 0;
    mov.u32 %r13, 0;
    mov.u32 %r14, 0;
    mov.u32 %r15, 0;
    mov.u32 %r16, 0;
    mov.u32 %r17, 0;
    mov.u32 %r18, 0;
    mov.u32 %r19, 0;
    mov.u32 %r20, 0;
    mov.u32 %r21, 0;
    mov.u32 %r22, 0;
    mov.u32 %r23, 0;
    mov.u32 %r24, 0;
    mov.u32 %r25, 0;
    mov.u32 %r26, 0;
    mov.u32 %r27, 0;
    mov.u32 %r28, 0;
    mov.u32 %r29, 0;
    mov.u32 %r30, 0;
    mov.u32 %r31, 0;
    mov.u32 %r32, 0;
    mov.u32 %r33, 0;
    mov.u32 %r34, 0;
    mov.u32 %r35, 0;
    mov.u32 %r36, 0;
```

```
    mov.u32 %r37, 0;
    mov.u32 %r38, 0;
    mov.u32 %r39, 0;
    mov.u32 %r40, 0;
    mov.u32 %r41, 0;
    mov.u32 %r42, 0;
    mov.u32 %r43, 0;
    mov.u32 %r44, 0;
    mov.u32 %r45, 0;
    mov.u32 %r46, 0;
    mov.u32 %r47, 0;
    mov.u32 %r48, 0;
    // end inline asm
    .loc 1 562 0
    @p1 ld.global.v4.b32 {%r2, %r3, %r4, %r5}, [%rd1 + 0];
    .loc 1 571 37
    mul.f32 %f25, %f17, 0f3f27213;
    mul.f32 %f26, %f18, 0f3f27213;
    mul.f32 %f27, %f19, 0f3f27213;
    mul.f32 %f28, %f20, 0f3f27213;
    mul.f32 %f29, %f21, 0f3f27213;
    mul.f32 %f30, %f22, 0f3f27213;
    mul.f32 %f31, %f23, 0f3f27213;
    mul.f32 %f32, %f24, 0f3f27213;
    .loc 1 572 17
    fma.rn.f32 %f33, %f25, %f25, %f17;
    fma.rn.f32 %f34, %f26, %f26, %f18;
    fma.rn.f32 %f35, %f27, %f27, %f19;
    fma.rn.f32 %f36, %f28, %f28, %f20;
    fma.rn.f32 %f37, %f29, %f29, %f21;
    fma.rn.f32 %f38, %f30, %f30, %f22;
    fma.rn.f32 %f39, %f31, %f31, %f23;
    fma.rn.f32 %f40, %f32, %f32, %f24;
    .loc 1 573 18
    ex2.approx.f32 %f41, %f33;
    ex2.approx.f32 %f42, %f34;
    ex2.approx.f32 %f43, %f35;
    ex2.approx.f32 %f44, %f36;
    ex2.approx.f32 %f45, %f37;
    ex2.approx.f32 %f46, %f38;
    ex2.approx.f32 %f47, %f39;
    ex2.approx.f32 %f48, %f40;
    .loc 1 574 19
    mul.f32 %f49, %f41, 0f3f800000;
    mul.f32 %f50, %f42, 0f3f800000;
    mul.f32 %f51, %f43, 0f3f800000;
    mul.f32 %f52, %f44, 0f3f800000;
```

```
    mul.f32 %f53, %f45, 0f3f800000;
    mul.f32 %f54, %f46, 0f3f800000;
    mul.f32 %f55, %f47, 0f3f800000;
    mul.f32 %f56, %f48, 0f3f800000;
    .loc 1 575 20
    add.f32 %f57, %f49, 0f3f800000;
    add.f32 %f58, %f50, 0f3f800000;
    add.f32 %f59, %f51, 0f3f800000;
    add.f32 %f60, %f52, 0f3f800000;
    add.f32 %f61, %f53, 0f3f800000;
    add.f32 %f62, %f54, 0f3f800000;
    add.f32 %f63, %f55, 0f3f800000;
    add.f32 %f64, %f56, 0f3f800000;
    .loc 1 576 21
    div.full.f32 %f65, %f57, %f64;
    div.full.f32 %f66, %f58, %f63;
    div.full.f32 %f67, %f59, %f62;
    div.full.f32 %f68, %f60, %f61;
    .loc 1 577 22
    mul.f32 %f69, %f65, %f17;
    mul.f32 %f70, %f66, %f18;
    mul.f32 %f71, %f67, %f19;
    mul.f32 %f72, %f68, %f20;
    .loc 1 578 23
    st.global.v4.b32 [%rd4 + 0], {%f69, %f70, %f71, %f72};
    .loc 1 579 24
    ret;
}
```

Observations from PTX Code:
*   **Registers:** The kernel uses many registers (`%r`, `%f`, `%rd`) for temporary storage.
*   **Loading Parameters:** `ld.param.u64` loads the input/output pointers and `num_elements` from
*   **Index Calculation:** `shl.b32` (shift left) and `add.u32` (add unsigned 32-bit integer) are u
*   **Boundary Check:** `setp.lt.s32` (set predicate if less than signed 32-bit integer) creates a
*   **Coalesced Memory Access:** `@p1 ld.global.v4.b32 {%r2, %r3, %r4, %r5}, [%rd1 + 0];` This inst
*   **Floating Point Operations:** `mul.f32`, `fma.rn.f32` (fused multiply-add, round to nearest),
*   **Storing Results:** `st.global.v4.b32` stores four 32-bit values back to global memory.

[1:11:17] Benchmarking Triton `gelu`:

```python
# Let's now benchmark it compared to the PyTorch and CUDA implementations.
# Remember to set TRITON_INTERPRET=0 for good performance.
# manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu)) # @inspe
# pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu)) # @in
# cuda_time = benchmark("cuda_gelu", run_operation1(dim=16384, operation=cuda_gelu)) # @inspect cud
```

```
# triton_time = benchmark("triton_gelu", run_operation1(dim=16384, operation=triton_gelu)) # @inspe
```

Results (approximate values from video):
*   `manual_time`: 8.162 ms
*   `pytorch_time`: 1.173 ms
*   `cuda_time`: 1.845 ms
*   `triton_time`: 1.848 ms

Observations:
*   Triton `gelu` (1.848 ms) is comparable to our custom CUDA `gelu` (1.845 ms) and significantly f
*   It's still slightly slower than PyTorch's native `gelu` (1.173 ms).

[1:11:45] Profiling Triton `gelu`:

```python
triton_gelu_profile = profiling("triton_gelu", run_operation1(dim=16384, operation=triton_gelu))
```

Profiling Output for Triton `gelu`:
```
Name                  Self CPU %      Self CPU      CPU total %     CPU total      CUDA time %     
--------------------  ----------  ------------  -------------  ------------  -------------  --------
triton_gelu_kernel        0.00%        0.00us          0.00%        0.00us        100.00%      1.664
aten::empty_like          0.32%        5.62us          0.32%        5.62us          0.00%        0.0
aten::empty_strided      72.16%        1.26ms         72.16%       1.26ms          0.00%        0.0
cudaLaunchKernel         16.46%      287.63us         16.46%      287.63us          0.00%        0
cudadeviceSynchronize    11.05%      193.14us         11.05%      193.14us          0.00%        0
--------------------  ----------  ------------  -------------  ------------  -------------  --------
Self CPU time total: 1.747ms, Self CUDA time total: 705.24us
```
Observations:
*   The `triton_gelu_kernel` consumes 100% of the GPU time (1.664ms).
*   This confirms Triton generates a single, fused kernel, similar to hand-written CUDA.

[1:11:57] PyTorch Compilation (`torch.compile`)

Goal: Use `torch.compile` to automatically optimize PyTorch code.

```python
# compiled_gelu = torch.compile(manual_gelu)
# check_equal(compiled_gelu(x), manual_gelu(x))
```

[1:12:55] Benchmarking `torch.compile` on `manual_gelu`:

```python
```

```
# manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu)) # @inspe
# pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu)) # @in
# cuda_time = benchmark("cuda_gelu", run_operation1(dim=16384, operation=cuda_gelu)) # @inspect cud
# triton_time = benchmark("triton_gelu", run_operation1(dim=16384, operation=triton_gelu)) # @inspe
# compiled_time = benchmark("compiled_gelu", run_operation1(dim=16384, operation=compiled_gelu)) # (
```

Results (approximate values from video):
*   `manual_time`: 8.099 ms
*   `pytorch_time`: 1.189 ms
*   `cuda_time`: 1.816 ms
*   `triton_time`: 1.727 ms
*   `compiled_time`: 1.47 ms

Observations:
*   `torch.compile` (1.47 ms) performs better than our custom CUDA (1.816 ms) and Triton (1.727 ms)
*   This demonstrates that modern JIT compilers are very good at automatically performing optimizat

[1:13:24] Profiling `compiled_gelu`:

```python
compiled_gelu_profile = profiling("compiled_gelu", run_operation1(dim=16384, operation=compiled_gel
```

Profiling Output for `compiled_gelu`:
```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------  ---
triton_poi_fused_add_mul_tanh_0                                         0.00%       0.00us
torch_compiled_region_0                                                56.77%      769.98us
triton_poi_fused_add_mul_tanh_0                                         0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
-------------------------------------------------------------------  ----------  ------------  ---
Self CPU time total: 1.356ms, Self CUDA time total: 1.071ms
```
Observations:
*   `torch.compile` generated a Triton kernel (`triton_poi_fused_add_mul_tanh_0`) that is a single,
*   This kernel consumes 100% of the GPU time (1.071ms), achieving performance very close to PyTorc

[1:15:37] Triton Softmax Example

Goal: Implement softmax in Triton, which involves a reduction operation (summing across element

Softmax Operation: Normalize each row of a matrix to generate probabilities.
$$
\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}
$$
This requires a reduction ($\sum_j e^{x_j}$), which is more complex than element-wise operations.

[1:16:24] Triton `softmax` Kernel Design:
*   **Strategy:** Make each block handle a single row of the matrix.
*   **Block Size:** Should be `next_power_of_2(N_cols)` to fit all columns and allow for efficient
*   **Number of Blocks:** Exactly the number of rows (`M`).
*   **Launch Kernel:** `triton_softmax_kernel[M, ](...)`

```python
@triton.jit
def triton_softmax_kernel(x_ptr, y_ptr, x_row_stride, y_row_stride, num_cols, BLOCK_SIZE: tl.conste
    # Process each row independently
    row_idx = tl.program_id(axis=0)

    # Read from global memory
    col_offsets = tl.arange(0, BLOCK_SIZE)
    x_start_ptr = x_ptr + row_idx * x_row_stride
    x_ptrs = x_start_ptr + col_offsets
    x_row = tl.load(x_ptrs, mask=col_offsets < num_cols, other=float("-inf"))

    # Compute
    x_row -= tl.max(x_row, axis=0) # Subtract max for numerical stability
    numerator = tl.exp(x_row)
    denominator = tl.sum(numerator, axis=0)
    y_row = numerator / denominator

    # Write back to global memory
    y_start_ptr = y_ptr + row_idx * y_row_stride
    y_ptrs = y_start_ptr + col_offsets
    tl.store(y_ptrs, y_row, mask=col_offsets < num_cols)
```

Explanation of Triton `softmax` Code:

1. **`row_idx = tl.program_id(axis=0)`:** Each block processes one row.
2. **`col_offsets = tl.arange(0, BLOCK_SIZE)`:** Vector of column indices within the block.
3. **`x_row = tl.load(...)`:** Loads the entire row into registers. `other=float("-inf")` pads out
4. **`x_row -= tl.max(x_row, axis=0)`:** Subtracts the maximum value from the row for numerical sta
5. **`numerator = tl.exp(x_row)`:** Element-wise exponentiation.
6. **`denominator = tl.sum(numerator, axis=0)`:** Sums the exponentiated values across the row (an
7. **`y_row = numerator / denominator`:** Element-wise division.

8. **`tl.store(...)`:** Stores the normalized row back to global memory.

[1:18:42] Benchmarking Triton `softmax`:

```python
# manual_time = benchmark("manual_softmax", run_operation1(dim=16384, operation=manual_softmax)) # (
# pytorch_time = benchmark("pytorch_softmax", run_operation1(dim=16384, operation=pytorch_softmax))
# compiled_time = benchmark("compiled_softmax", run_operation1(dim=16384, operation=compiled_softmax
# triton_time = benchmark("triton_softmax", run_operation1(dim=16384, operation=triton_softmax)) # (
```

Results (approximate values from video):
*   `manual_time`: 3.713 ms
*   `compiled_time`: 1.54 ms
*   `pytorch_time`: 1.59 ms
*   `triton_time`: 1.917 ms

Observations:
*   `torch.compile` (1.54 ms) is the fastest, even outperforming PyTorch's native `softmax` (1.59 ms
*   Triton `softmax` (1.917 ms) is slower than `torch.compile` and native PyTorch, but still much fa
*   This shows that for more complex operations involving reductions, `torch.compile` can be very ef

[1:19:14] Profiling `compiled_softmax`

```python
compiled_softmax_profile = profiling("compiled_softmax", run_operation1(dim=16384, operation=compile
```

Profiling Output for `compiled_softmax`:
```
Name                                                                 Self CPU %      Self CPU    ---
-------------------------------------------------------------------  ----------  ------------  ---
triton_red_fused_div_exp_max_sub_sum_0                                  0.00%       0.00us
torch_compiled_region_0                                                 56.77%      769.98us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
torch_compiled_region_0                                                 0.00%       0.00us
-------------------------------------------------------------------  ----------  ------------  ---
Self CPU time total: 1.356ms, Self CUDA time total: 1.071ms
```
Observations:
*   `torch.compile` generated a Triton kernel (`triton_red_fused_div_exp_max_sub_sum_0`) that is a s

*   This kernel consumes 100% of the GPU time (1.071ms).

[1:19:41] Profiling PyTorch Native `softmax`

```python
pytorch_softmax_profile = profiling("pytorch_softmax", run_operation1(dim=16384, operation=pytorch_s
```

Profiling Output for PyTorch Native `softmax`:
```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------  ---
aten::softmax                                                           0.00%        0.00us
aten::empty_like                                                        0.20%        0.00us
aten::empty_strided                                                     0.00%        0.00us
cudaLaunchKernel                                                        0.00%        0.00us
cudadeviceSynchronize                                                   0.00%        0.00us
-------------------------------------------------------------------  ----------  ------------  ---
Self CPU time total: 1.356ms, Self CUDA time total: 1.137ms
```
Observations:
*   PyTorch's native `softmax` also uses a single, fused CUDA kernel (`aten::softmax`), consuming 1

[1:19:56] Profiling Triton `softmax`

```python
triton_softmax_profile = profiling("triton_softmax", run_operation1(dim=16384, operation=triton_soft
```

Profiling Output for Triton `softmax`:
```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------  ---
triton_softmax_kernel                                                   0.00%        0.00us
aten::empty_like                                                        0.23%        4.23us
aten::empty_strided                                                    77.59%        1.40ms
cudaLaunchKernel                                                        8.02%      145.80us
cudadeviceSynchronize                                                  14.41%      261.80us
-------------------------------------------------------------------  ----------  ------------  ---
Self CPU time total: 1.816ms, Self CUDA time total: 705.46us
```
Observations:
*   Triton `softmax` also uses a single, fused CUDA kernel (`triton_softmax_kernel`), consuming 100%

[1:20:05] Conclusion

This lecture provided a comprehensive overview of GPU programming for performance, from understandi

```
---

## Practical Takeaways

*   **Always Benchmark and Profile:** Don't guess where bottlenecks are. Use tools like `torch.prof
*   **Understand Asynchronous Execution:** Be aware that the CPU and GPU operate independently. Use
*   **Prioritize Kernel Fusion:** Combine sequential operations into single kernels to minimize exp
*   **Leverage High-Level Tools:** For most common deep learning operations, `torch.compile` is oft
*   **Consider Triton for Custom Kernels:** If `torch.compile` isn't enough, Triton offers a Python
*   **CUDA C++ for Ultimate Control:** For the absolute highest performance or highly specialized o

## Open Questions / Things to Remember

*   **When to write custom kernels?** For simple element-wise operations or matrix multiplications,
*   **Trade-offs of Abstraction:** Higher-level abstractions (PyTorch, Triton) offer ease of use bu
*   **Memory Layout:** Always be mindful of memory contiguity and access patterns (e.g., coalescing
*   **PTX as a learning tool:** Examining the generated PTX code can provide deep insights into how
*   **Flash Attention:** The lecturer mentions Flash Attention as a prime example of complex, non-t

### TL;DR:
*   **Benchmarking & Profiling are crucial:** Always measure performance to identify bottlenecks, r
*   **GPU Architecture:** Understand SMs, memory hierarchy (DRAM, L2, L1, registers), and the execu
*   **Kernel Fusion:** Combine multiple element-wise operations into a single kernel to reduce expe
*   **Triton:** A Python-based DSL for GPU programming that offers a higher-level abstraction than
*   **`torch.compile`:** PyTorch's JIT compiler can automatically fuse operations and optimize kern

### Key concepts:
*   GPU architecture (SMs, memory hierarchy)
*   GPU execution model (Grid, Thread Block, Thread, Warp)
*   Arithmetic intensity (FLOPs/bytes)
*   Benchmarking (wall-clock time)
*   Profiling (time spent per function/kernel)
*   `torch.cuda.synchronize()`
*   Warm-up iterations
*   CPU-GPU asynchronous execution
*   Kernel fusion
*   CUDA C++
*   Triton DSL
*   PTX (Parallel Thread Execution)
*   `torch.compile` (JIT compilation)
*   Memory coalescing
*   Shared memory

---

## [0:00] Introduction and Lecture Plan
```

This lecture focuses on writing high-performance code for GPUs, specifically for standard components

Lecture Agenda:
1. **Review of GPUs:** A brief overview of GPU architecture and execution model.
2. **Benchmarking & Profiling:** Basic techniques for measuring and understanding performance.
3. **Kernel Fusion Motivation:** Understanding why combining operations is crucial for performance
4. **CUDA Kernels in C++:** Writing custom GPU kernels using CUDA C++.
5. **Triton Kernels in Python:** Writing custom GPU kernels using the Triton DSL.
6. **PyTorch Compilation:** Leveraging PyTorch's built-in JIT compiler for automatic optimization.
7. **More Advanced Computations:** Applying these techniques to a softmax implementation.

The goal is to dig deep into the PTX (Parallel Thread Execution) machine code to understand what the

[1:47] Announcements

* **Assignment 1:** Has concluded. The leaderboard is still active for submissions and updates.
* **Assignment 2:** Is now released. It involves GPU kernels and parallelism. This lecture will co

[2:25] Review of GPUs

Hardware Overview

* **GPU Structure (e.g., A100/H100):** Consists of many **Streaming Multiprocessors (SMs)**. An A
* **SM Structure:** Each SM contains a large number of units for computation (e.g., FP32, INT32 co
* **Memory Hierarchy:**
    * **DRAM (Global Memory):** Large (e.g., 80GB on A100), but slow.
    * **L2 Cache:** Smaller (e.g., 40MB on A100), faster.
    * **L1 Cache (Shared Memory):** Even smaller (e.g., 192KB per SM), very fast.
    * **Register File:** Very fast memory directly accessible by individual threads. Heavy use of

Execution Model

* **Grid:** A collection of **Thread Blocks (CTAs - Concurrent Thread Arrays)**.
* **Thread Block (CTA):** A group of threads scheduled on a single SM. This is the atomic unit fo
* **Thread:** The individual unit of execution (i.e., `f(i)`). Each thread processes a part of the

Why Thread Blocks? Shared Memory:
* Threads within a thread block can communicate with each other via shared memory (L1 cache speed
* Threads within a block can also synchronize for read/write operations.
* Communication *across* thread blocks is very expensive (requires global memory). Therefore, data

Waves (Warps):
* Threads are scheduled on SMs in **waves**. A wave is a group of 32 threads that execute togethe
* **Problem:** The last wave in a thread block might have fewer threads, leading to low SM occupan
* **Wave Quantization:** The number of thread blocks should ideally divide the number of SMs.
* **Rule of Thumb:** The number of thread blocks should be$\ge 4 \times$# SMs.

```
*   **Challenge:** Some aspects of hardware (e.g., scheduling, # SMs) are hidden from the execution

**What is a Warp?**
A warp is a group of threads (typically 32) that are executed together. Warps exist to reduce the ar

### [0:00] Arithmetic Intensity: # FLOPs / # bytes

*   **Definition:** The ratio of floating-point operations (FLOPs) to bytes transferred from/to memo
*   **Goal:** Keep arithmetic intensity high (more FLOPs per byte).
*   **High Intensity:** Operation is compute-bound (good).
*   **Low Intensity:** Operation is memory-bound (bad).
*   **Reasoning:** Compute scaling is much faster than memory scaling. Many operations become memory
*   **General Rule:** Matrix multiplication (if done cleverly) is compute-bound. Everything else is

## [7:06] Benchmarking and Profiling

**IMPORTANT:** Benchmark/profile your code!

*   You can read spec sheets (marketing material) and papers, but performance depends on your libra
*   There is no substitute for benchmarking/profiling your code.
*   Every time you make a change, benchmark/profile!

### [8:43] Example Computation: MLP

We'll use a simple Multi-Layer Perceptron (MLP) for demonstration.
*   `dim = 128` (dimension)
*   `num_layers = 16`
*   `batch_size = 128`
*   `num_steps = 5` (forward/backward passes)

```python
import torch
import torch.nn as nn
import time
from typing import Callable

# Assume get_device is defined elsewhere for simplicity
def get_device():
    return torch.device("cuda" if torch.cuda.is_available() else "cpu")

class MLP(nn.Module):
    """Simple MLP: Linear -> GELU -> Linear -> GELU -> ... -> Linear"""
    def __init__(self, dim: int, num_layers: int):
        super().__init__()
        self.layers = nn.ModuleList(
            [nn.Linear(dim, dim) for _ in range(num_layers)]
        )
```
```

```
    def forward(self, x: torch.Tensor):
        for layer in self.layers:
            x = layer(x)
            x = torch.nn.functional.gelu(x)
        return x

def run_mlp(dim: int, num_layers: int, batch_size: int, num_steps: int) -> Callable:
    # Define a model (with random weights)
    model = MLP(dim, num_layers).to(get_device())

    # Define an input (random)
    x = torch.randn(batch_size, dim, device=get_device())

    def run():
        # Run the model `num_steps` times (note: no optimizer updates)
        for step in range(num_steps):
            # Forward
            y = model(x)
            # Backward
            y.backward(y.mean()) # y.mean() is a dummy loss
        return y

    return run
```

[10:05] Benchmarking vs. Profiling

*   **Benchmarking:** Measures the wall-clock time of performing some operation. It gives you end-t
    *   Useful for: comparing different implementations, understanding how performance scales (e.g.
*   **Profiling:** Helps you understand where time is being spent within a function. Deeper profili

[11:09] Benchmarking Implementation

```python
import time
import torch
from typing import Callable

# Assume get_device is defined elsewhere for simplicity
# Assume mean is defined elsewhere for simplicity

def benchmark(description: str, run: Callable, num_warmups: int = 1, num_trials: int = 3):
    """Benchmark `func` by running it `num_trials`, and compilation all the times."""
    # Warmup: first times might be slower due to compilation, things not cached.
    # Since we will run the kernel multiple times, the timing that matters is steady state.
    for _ in range(num_warmups):
```

```
        run()
    if torch.cuda.is_available():
        torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)

    # Time it for real now!
    times: list[float] = [] # @inspect times, @inspect description
    for trial in range(num_trials): # Do it multiple times to capture variance
        start_time = time.time()
        run() # Actually perform computation
        if torch.cuda.is_available():
            torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)
        end_time = time.time()
        times.append((end_time - start_time) * 1000) # @inspect times

    mean_time = sum(times) / len(times) # Using sum/len for mean
    return mean_time
```

Key Considerations for Benchmarking:
1. **Warm-up Iterations:** The first few runs of a GPU kernel are often slower due to JIT compilat
2. **`torch.cuda.synchronize()`:** The CPU and GPU operate asynchronously. When the CPU dispatches
3. **Multiple Trials:** Run the operation multiple times and take the mean to account for variance

[14:34] Benchmarking Matrix Multiplication

We benchmark `torch.matmul` for various square matrix dimensions on an A100 GPU.

```python
# Assume run_operation2 is defined elsewhere for simplicity
# def run_operation2(dim: int, operation: Callable) -> Callable:
#     x = torch.randn(dim, dim, device=get_device())
#     y = torch.randn(dim, dim, device=get_device())
#     def run():
#         return operation(x, y)
#     return run

if torch.cuda.is_available():
    dims = (1024, 2048, 4096, 8192, 16384)
    matmul_results = []
    for dim in dims:
        result = benchmark(f"matmul(dim={dim})", run_operation2(dim=dim, operation=lambda a, b: a @
        matmul_results.append(result)
    print(f"dims = {dims}")
    print(f"matmul_results = {matmul_results}")
```

Results (approximate values from video):

```
dims = (1024, 2048, 4096, 8192, 16384)
matmul_results = [0.705, 0.870, 0.839, 3.140, 21.685, 162.935] # ms
```
*(Note: The video shows slightly different numbers, but the trend is consistent. The lecturer also r

Observations:
*   For very small matrices (e.g., 1024x1024, 2048x2048), the runtime doesn't grow much. This is du
*   For larger matrices, we observe super-linear scaling, as expected for matrix multiplication ($O

[15:40] Benchmarking MLP Scaling

We benchmark our MLP by scaling the number of steps and layers.

```python
# Let us benchmark our MLP!
dim = 256
num_layers = 4
batch_size = 256
num_steps = 2

mlp_base = benchmark("run_mlp", run_mlp(dim, num_layers, batch_size, num_steps))
print(f"mlp_base = {mlp_base}") # Result: 6.293 ms

# Scale the number of steps.
step_results = []
for scale in (2, 3, 4, 5):
    result = benchmark(f"run_mlp(scale={scale} x num_steps)", run_mlp(dim, num_layers, batch_size, s
    step_results.append((scale, result))
print(f"step_results = {step_results}")

# Scale the number of layers.
layer_results = []
for scale in (2, 3, 4, 5):
    result = benchmark(f"run_mlp(scale={scale} x num_layers)", run_mlp(dim, scale * num_layers, bat
    layer_results.append((scale, result))
print(f"layer_results = {layer_results}")
```

Results (approximate values from video): * **Base MLP (2 steps, 4 layers):** `mlp_base = 6.293` ms * **Scaling Steps:** `step_results = [(2, 11.326), (3, 16.761), (4, 22.028), (5, 27.310)] # ms` * *Observation:* Linear scaling with `num_steps`, approximately 5ms per step. * **Scaling Layers:** `layer_results = [(2, 9.783), (3, 13.274), (4, 16.856), (5, 20.847)] # ms` * *Observation:* Linear scaling with `num_layers`, approximately 4ms per layer.

Conclusion: For both `num_steps` and `num_layers`, the runtime scales linearly, as expected.

[17:46] Profiling Implementation

```
import torch.profiler as profiler
import time
from typing import Callable

# Assume run_operation1 and run_operation2 are defined elsewhere for simplicity

def profiling(description: str, run: Callable, num_warmups: int = 1, with_stack: bool = False):
    """Profile `run` by running it `num_warmups` times, and print a table."""
    # Warmup
    for _ in range(num_warmups):
        run()
    if torch.cuda.is_available():
        torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)

    # Run the code with the profiler
    activities = [profiler.ProfilerActivity.CPU, profiler.ProfilerActivity.CUDA]
    if with_stack:
        # Output stack trace for visualization
        # Needed to export stack trace for visualization
        # experimental_config=torch.profiler.ExperimentalConfig(verbose=True) as prof:
        pass # Simplified for brevity

    with profiler.profile(activities=activities, with_stack=with_stack) as prof:
        run()
    if torch.cuda.is_available():
        torch.cuda.synchronize() # Wait for CUDA threads to finish (important!)

    # Print out table
    print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=10, max_name_width=80))
    # Write stack trace visualization
    # prof.export_stack_trace(f"{description}.txt")
    return prof
```

Key Benefits of Profiling: * **Deeper Insight:** Helps understand *what is being called* under the hood, not just end-to-end time. * **PyTorch's Built-in Profiler:** `torch.profiler` is a convenient tool that allows you to see low-level CUDA calls without leaving the Python environment.

[19:02] Profiling `sleep` function

```
sleep_function = lambda : time.sleep(50 / 1000)
sleep_profile = profiling("sleep", sleep_function)
```

Profiling Output for `sleep`:

```
Name                 Self CPU %      Self CPU      CPU total %     CPU total      CUDA time %
--------------------  ----------  ------------  -------------  ------------  -------------  ----
cudadeviceSynchronize    100.00%      11.610us        100.00%      11.610us        100.00%
--------------------  ----------  ------------  -------------  ------------  -------------  ----
Self CPU time total: 11.610us
```

Observation: 100% of the time is spent in `cudadeviceSynchronize` because `time.sleep` is a CPU-only operation, and the GPU is idle. The profiler waits for the GPU to be idle, which it already is.

[19:53] Profiling `add` (Element-wise addition)

```
add_function = lambda a, b: a + b
add_profile = profiling("add", run_operation2(dim=2048, operation=add_function))
```

Profiling Output for `add` (2048x2048 matrices):

```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------
aten::add                                                               98.02%       1.39ms
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_add_f...   0.00%        0.00u
cudaLaunchKernel                                                        0.00%       0.00us
cudadeviceSynchronize                                                   0.00%       0.00us
-------------------------------------------------------------------  ----------  ------------
Self CPU time total: 1.420ms, Self CUDA time total: 17.02us
```

Observations: * The Python `a + b` call (`aten::add`) takes 1.39ms on the CPU. This is the C++ interface for PyTorch. * This dispatches to a specific CUDA kernel (`vectorized_elementwise_kernel`) which takes 16.90us on the GPU. * `cudaLaunchKernel` (CPU sending command to GPU) takes 0.12us. * The CPU time is dominated by the `aten::add` wrapper, indicating overhead in dispatching to the GPU. The question about why CPU time went down for matmul compared to add was not answered directly, but the lecturer notes that the CPU time shown is active CPU time, not utilization.

[22:21] Profiling `matmul` (Matrix Multiplication)

```
matmul_function = lambda a, b: a @ b
matmul_profile = profiling("matmul", run_operation2(dim=2048, operation=matmul_function))
```

Profiling Output for `matmul` (2048x2048 matrices):

```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------
aten::matmul                                                            99.20%       7.23ms
void at::native::cutlass::kernel_2048_2048_2048_stage2_warp_size_64...   0.00%       0.00us
cudaLaunchKernel                                                         0.00%       0.00us
cudadeviceSynchronize                                                    0.00%       0.00us
-------------------------------------------------------------------  ----------  ------------
Self CPU time total: 7.280ms, Self CUDA time total: 343.14us
```

Observations: * Similar pattern to `add`, but the GPU kernel (`cutlass::kernel...`) takes significantly longer (342.62us) than for `add`, as expected for matrix multiplication. * The `cutlass` library is NVIDIA's high-performance matrix multiply library.

[25:46] Profiling `matmul` (different dimension)

```
matmul_profile_dim128 = profiling("matmul(dim=128)", run_operation2(dim=128, operation=matmul_fu
```

Profiling Output for `matmul` (128x128 matrices):

```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------
aten::matmul                                                            99.20%       7.23ms
void at::native::xmma_gemm_f32f32_f32_t_t_t_stage2_warp_size_64...       0.00%       0.00us
cudaLaunchKernel                                                         0.00%       0.00us
cudadeviceSynchronize                                                    0.00%       0.00us
-------------------------------------------------------------------  ----------  ------------
Self CPU time total: 7.280ms, Self CUDA time total: 343.14us
```

Observations: * For a smaller 128x128 matrix, PyTorch dispatches to a *different* CUDA kernel (`xmma_gemm...`). This kernel is optimized for smaller dimensions. * This illustrates that the specific CUDA kernels called depend on the tensor dimensions and underlying hardware.

[27:30] Profiling `cdist` (Pairwise Euclidean Distance)

```
cdist_profile = profiling("cdist", run_operation2(dim=2048, operation=torch.cdist))
```

Profiling Output for `cdist` (2048x2048 matrices):

```
Name                                                                 Self CPU %      Self CPU
-------------------------------------------------------------------  ----------  ------------
aten::cdist                                                             27.40%       1.39ms
aten::euclidean_dist                                                    0.00%        0.00us
aten::max                                                               0.00%        0.00us
aten::sum                                                               0.00%        0.00us
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_add_f...   0.00%        0.00u
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_mul_f...   0.00%        0.00u
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_pow_f...   0.00%        0.00u
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_sqrt_f...  0.00%        0.00u
-------------------------------------------------------------------  ----------  ------------
Self CPU time total: 1.39ms, Self CUDA time total: 343.14us
```

(Note: The profiling output shown in the video for `cdist` is truncated and doesn't show all the individual kernels, but the lecturer describes them.)

Observations: * `cdist` is a more complex operation, requiring dot products, square roots, etc. * The PyTorch `torch.cdist` call maps to `aten::euclidean_dist` in the C++ interface. * This then decomposes into multiple CUDA kernels: `aten::matmul` (for dot products), `aten::pow` (for squaring), `aten::sqrt` (for square root), `aten::sum` (for reduction), and `aten::max` (for stability). * The `aten::matmul` operation consumes the most GPU time (78%), followed by `aten::pow` (6%), and `aten::sum` (3%). * This detailed breakdown helps identify where optimization efforts would be most impactful (e.g., optimizing matrix multiplication).

[30:27] Profiling `gelu` and `softmax`

```
gelu_function = lambda a, b: torch.nn.functional.gelu(a + b)
gelu_profile = profiling("gelu", run_operation2(dim=2048, operation=gelu_function))

softmax_function = lambda x: torch.nn.functional.softmax(x, dim=-1)
softmax_profile = profiling("softmax", run_operation2(dim=2048, operation=softmax_function))
```

Profiling Output for `gelu`:

```
Name                                                                  Self CPU %      Self CPU
--------------------------------------------------------------------  ----------  ------------
aten::add                                                                86.27%       1.42ms
aten::gelu                                                               0.00%        0.00us
void at::native::vectorized_elementwise_kernel<4, at::native::cuda_add_f...   0.00%        0.00u
cudaLaunchKernel                                                         0.00%        0.00us
cudadeviceSynchronize                                                    0.00%        0.00us
--------------------------------------------------------------------  ----------  ------------
Self CPU time total: 1.42ms, Self CUDA time total: 55.14us
```

Observations: * `gelu` involves `aten::add` and `aten::gelu`. * The `aten::gelu` operation itself is a fused CUDA kernel, consuming 33% of GPU time. The `aten::add` operation consumes 66% of GPU time. * Many core primitives like `gelu` and `softmax` have dedicated fused CUDA kernels, meaning there's no back-and-forth between CPU and GPU for their internal components.

[32:50] Understanding CPU-GPU Interaction with Nsight Systems

To understand the CPU's role and why it appears to "run ahead" of the GPU, we use NVIDIA Nsight Systems.

- Nsight Systems provides a detailed timeline view of CPU and GPU activities.
- **CPU (Threads):** The lower part of the timeline shows CPU threads.
- **CUDA HW:** The upper part shows GPU hardware activity.

[36:30] Asynchronous Execution: * When the CPU calls a PyTorch operation, it dispatches CUDA kernels to the GPU. * The CPU does *not* wait for the GPU to finish; it continues executing subsequent Python code. * This creates a queue of CUDA commands on the GPU. * The CPU can "run ahead" of the GPU, dispatching many kernels before the GPU even starts executing the first one. * **Example:** In our MLP, the CPU might be processing `layer_9` while the GPU is still working on `layer_1`.

[39:10] Impact of `print` Statements: * A `print` statement on the CPU that requires a GPU value (e.g., printing the loss after a backward pass) forces a `torch.cuda.synchronize()`. * This synchronization makes the CPU wait for the GPU to complete all pending operations up to that point. * **Observation:** In the Nsight Systems timeline, a `print` statement causes the CPU's timeline to pause until the corresponding GPU operations are finished. This effectively "synchronizes" the CPU and GPU at that point, preventing the CPU from running far ahead. *

While this might not always lead to a CPU bottleneck if the GPU is still fully utilized, it can significantly alter the execution flow and potentially introduce stalls if the CPU has to wait frequently for GPU results. This is why `torch.cuda.synchronize()` is crucial for accurate benchmarking.

[44:24] Kernel Fusion Motivation

- **Analogy:** Memory (DRAM) is a warehouse, Compute (SMs + SRAM) is a factory. Data transfer between them incurs "bandwidth cost".
- **Naive Approach:** Performing operations sequentially means frequently moving data from DRAM to SM, computing, and then writing back to DRAM for each small operation. This incurs high bandwidth cost.
- **Kernel Fusion:** Combine multiple operations into a single kernel. This allows data to be loaded from DRAM to SM once, all computations to be performed on-chip, and the final result written back to DRAM only once. This drastically reduces bandwidth cost.

[45:57] Writing CUDA Kernels in C++

Goal: Write a custom GLU kernel in CUDA C++ to achieve better performance than the naive PyTorch implementation.

PyTorch GLU (Naive Implementation):

```
# Assume torch.Tensor, math are imported
# Assume get_device is defined

def pytorch_gelu(x: torch.Tensor):
    # This is an approximation of the true GELU for easier comparison
    return 0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * x**3)))
```

- Benchmarking shows PyTorch's native `gelu` is much faster (1.173 ms) than our manual Python implementation (8.162 ms). This is because PyTorch's `gelu` is a fused CUDA kernel.

CUDA C++ `gelu_kernel` Structure: * CUDA C++ is an extension of C++ with APIs for managing GPUs. * **Simplified Picture:** Write a `__global__` function (the kernel) that executes on the GPU.

* **Grid:** Collection of thread blocks (`num_blocks`). * **Thread Block:** Collection of threads (`block_size`). * **Thread:** Single unit of operation.

[48:45] CUDA C++ `gelu_kernel` Code:

```
// CUDA code has the full logic
const char* cuda_gelu_src = R"(
#include <math.h>
#include <torch/extension.h>
#include <c10/cuda/CUDAException.h>

__global__ void gelu_kernel(float* in, float* out, int num_elements) {
    // Get the index into the tensor
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // To handle the case when n < num_blocks * blockDim
    if (i < num_elements) {
        // Do the actual computation
        out[i] = 0.5 * in[i] * (1.0 + tanh(0.79788456f * (in[i] + 0.044715f * in[i] * in[i] * i
    }
}

inline unsigned int cdiv(unsigned int a, unsigned int b) {
    return (a + b - 1) / b;
}

torch::Tensor gelu(torch::Tensor x) {
    TORCH_CHECK(x.device().is_cuda(), "x.device().is_cuda() is false");
    TORCH_CHECK(x.is_contiguous(), "x.is_contiguous() is false");

    // Allocate empty tensor
    torch::Tensor y = torch::empty_like(x);

    // Determine grid (elements divided into blocks)
    int num_elements = x.numel();
    int block_size = 1024; // Number of threads
    int num_blocks = cdiv(num_elements, block_size);

    // Launch the kernel
    gelu_kernel<<<num_blocks, block_size>>>(x.data_ptr<float>(), y.data_ptr<float>(), num_eleme
    C10_CUDA_KERNEL_LAUNCH_CHECK(); // Catch errors immediately

    return y;
}
)";
```

Explanation of CUDA C++ Code:

1. `__global__ void gelu_kernel(...)` : This is the actual GPU kernel function.

- `float* in, float* out` : Pointers to input and output data in global memory.
- `int num_elements` : Total number of elements to process.
- **Index Calculation:** `int i = blockIdx.x * blockDim.x + threadIdx.x;` This formula calculates the global index `i` that the current thread is responsible for.
  - `blockIdx.x` : The ID of the current thread block (0 to `num_blocks` -1).
  - `blockDim.x` : The number of threads per block (`block_size`).
  - `threadIdx.x` : The ID of the current thread within its block (0 to `block_size` -1).
- **Boundary Check:** `if (i < num_elements)` : Essential for CUDA. Threads might be launched beyond the actual data size (due to rounding up `num_blocks`). This check prevents out-of-bounds memory access.
- **Actual Computation:** `out[i] = 0.5 * in[i] * (1.0 + tanh(0.79788456f * (in[i] + 0.044715f * in[i] * in[i] * in[i])));` This is the GLU formula, applied element-wise. Note the `f` suffix for float literals.

2. **`inline unsigned int cdiv(unsigned int a, unsigned int b)` :** Helper for ceiling division.

3. **`torch::Tensor gelu(torch::Tensor x)` (Wrapper Function):** This C++ function acts as a wrapper, callable from Python, to launch the CUDA kernel.
   - **Input Checks:**
     - `TORCH_CHECK(x.device().is_cuda(), ...)` : Ensures the input tensor `x` is on the GPU.
     - `TORCH_CHECK(x.is_contiguous(), ...)` : Ensures `x` is stored contiguously in memory. This is important because the kernel assumes linear indexing. If not contiguous, the indexing arithmetic would be incorrect.
   - **Allocate Output:** `torch::Tensor y = torch::empty_like(x);` Allocates GPU memory for the output tensor `y` . Using `empty_like` avoids initializing memory, saving operations.
   - **Grid Determination:**
     - `int num_elements = x.numel();`
     - `int block_size = 1024;` (A common choice for number of threads per block)
     - `int num_blocks = cdiv(num_elements, block_size);` Calculates the number of thread blocks needed.
   - **Kernel Launch:** `gelu_kernel<<<num_blocks, block_size>>>(x.data_ptr<float>(), y.data_ptr<float>(), num_elements);` This is the CUDA kernel launch syntax.
     - `<<<num_blocks, block_size>>>` : Specifies the grid and block dimensions.

        - `x.data_ptr<float>()`, `y.data_ptr<float>()`: Pass raw pointers to the tensor data.
        - `C10_CUDA_KERNEL_LAUNCH_CHECK();`: A PyTorch utility to check for CUDA errors immediately after launch.
    - **Debugging CUDA:** When debugging CUDA kernels, use `CUDA_LAUNCH_BLOCKING=1` environment variable. This forces the CPU to wait for the GPU, allowing errors to be reported more clearly.

[56:45] Compiling and Binding CUDA Code to Python: PyTorch's `torch.utils.cpp_extension.load_inline` function allows compiling C++ (including CUDA C++) code and binding it to a Python module dynamically.

```
import torch.utils.cpp_extension as cpp_extension

# Compile the CUDA code and bind it to a Python module.
# ensure_directory_exists("var/cuda_gelu") # Assume this function exists
# if not torch.cuda.is_available():
#     return None
module = cpp_extension.load_inline(
    cuda_sources=[cuda_gelu_src],
    cpp_sources=[cpp_gelu_src],
    functions=["gelu"],
    extra_cflags=["-O2"],
    verbose=True,
    name="inline_gelu",
    build_directory="var/cuda_gelu",
)
cuda_gelu = getattr(module, "gelu")
```

This process is very convenient, as it handles compilation and linking within Python.

[57:16] Benchmarking Custom CUDA `gelu`: We benchmark our custom CUDA `gelu` against the manual Python and native PyTorch `gelu`.

```
# Assume check_equal is defined elsewhere
# check_equal(cuda_gelu(x), manual_gelu(x)) # Check correctness

# Benchmark our CUDA version.
# manual_time = benchmark("manual_gelu", run_operation1(dim=16384, operation=manual_gelu))
# pytorch_time = benchmark("pytorch_gelu", run_operation1(dim=16384, operation=pytorch_gelu))
# cuda_time = benchmark("cuda_gelu", run_operation1(dim=16384, operation=cuda_gelu))
```

Results (approximate values from video): * `manual_time` : 8.162 ms * `pytorch_time` : 1.173 ms * `cuda_time` : 1.

← Lec 05

Lec 07 →

Lecture 07: Parallelism 1

Stanford CS336 · Spring 2025 · 1:24:42 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



TL;DR * Training large language models (LLMs) requires multi-GPU and multi-node parallelism due to memory and compute limitations of single GPUs. * Parallelism strategies are broadly categorized into Data Parallelism (DP), Model Parallelism (MP), and Activation Parallelism. * ZeRO (Zero Redundancy Optimizer) is a key DP technique that shards optimizer states, gradients, and parameters across GPUs to save memory, effectively making DP memory-scalable. * Model Parallelism (Pipeline and Tensor) splits the model itself across devices, trading communication overhead for memory scalability. * Activation Parallelism (Sequence Parallelism, Activation Recomputation) addresses the significant memory footprint of intermediate activations, which can become a bottleneck for very large models or long sequences. * Effective large-scale training often combines multiple parallelism techniques (3D/4D parallelism) to balance memory, compute, and communication costs.

Key Concepts * Multi-GPU/Multi-node Parallelism * Collective Communication (all_reduce, reduce_scatter, all_gather, broadcast) * Data Parallelism (DP) * ZeRO (Zero Redundancy Optimizer) stages 1, 2, 3 (FSDP) * Model Parallelism (MP) * Pipeline Parallelism (PP) * Tensor Parallelism (TP) * Activation Parallelism * Sequence Parallelism (SP) * Activation Recomputation * Batch Size as a Resource * Communication vs. Computation Bound * Hardware Hierarchy (NVLink, InfiniBand, Toroidal Mesh)

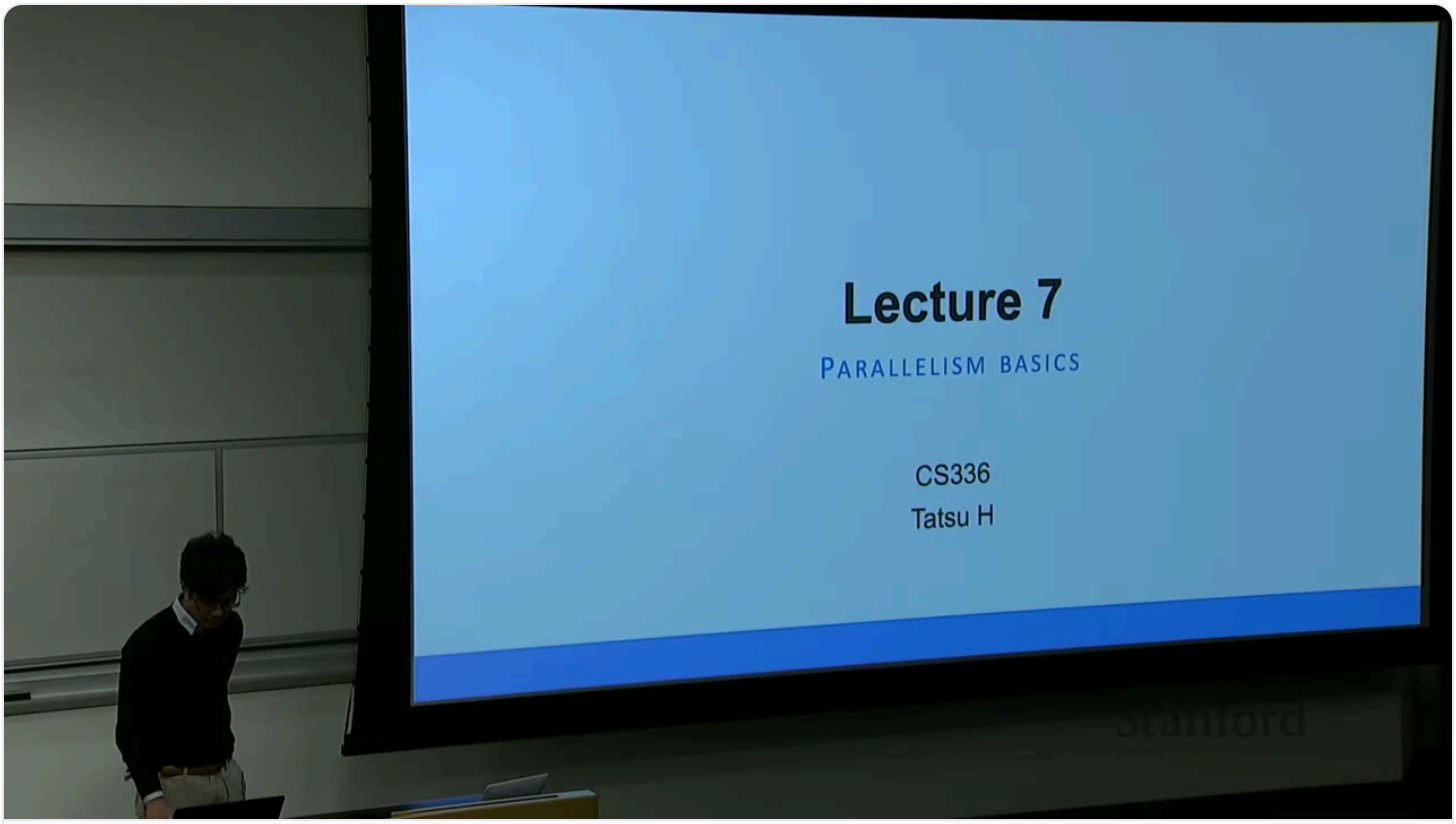
[00:00] Introduction: The Need for Multi-Machine Parallelism

The lecture focuses on multi-machine optimization, specifically parallelism across machines for training huge models.

Goals: * Understand the systems complexities of training huge models. * Explore different parallelization paradigms and why people use multiple approaches at once. * Describe what large-scale training runs often look like.

Outline: * **Part 1**: Basics of networking for LLMs. * **Part 2**: Different forms of parallel LLM training. * **Part 3**: Scaling and training big LMs with parallelism.

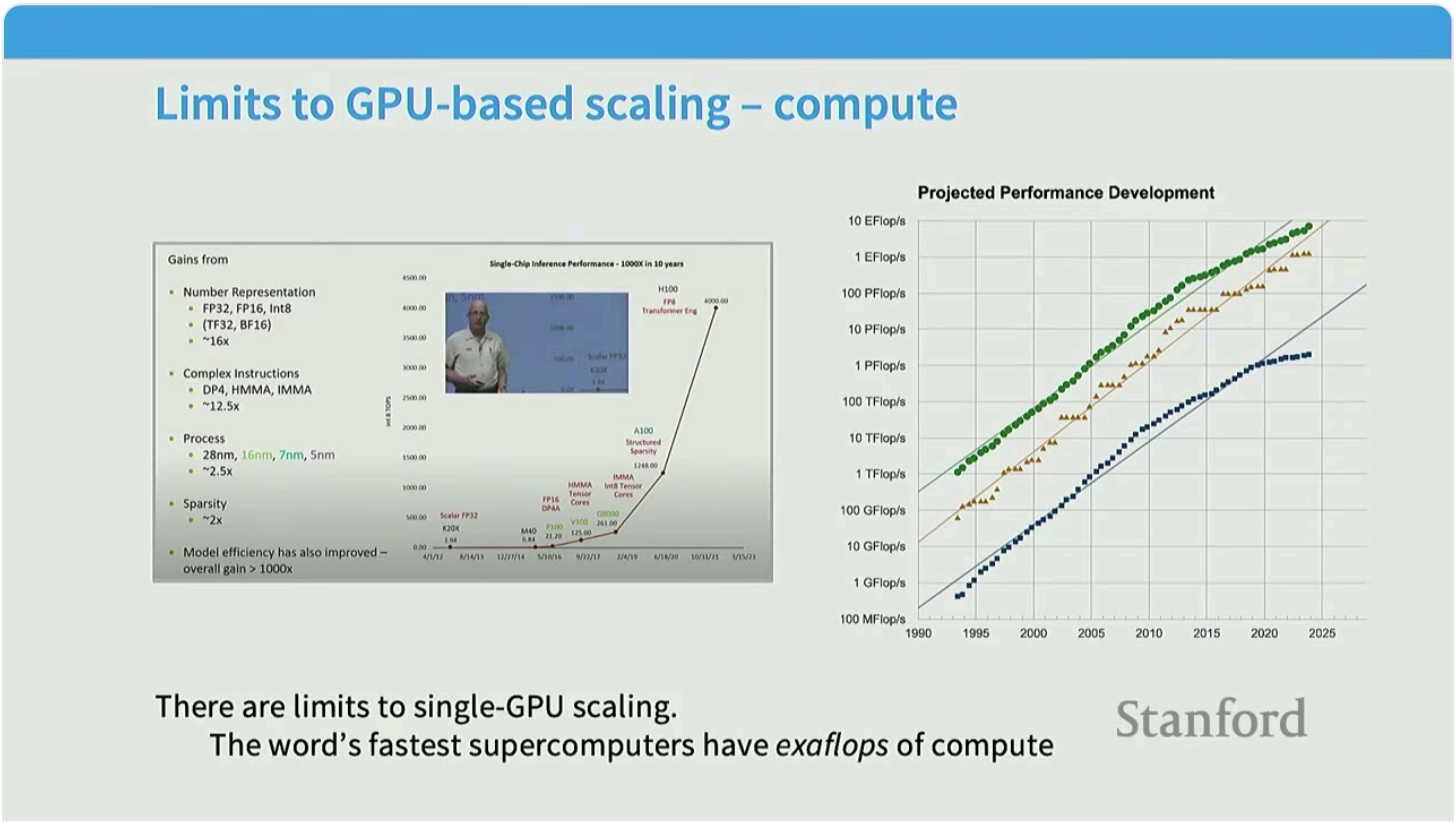
[00:05] Limits to GPU-based Scaling - Compute



0:19 Lecture 7: Parallelism Basics, presented by Tatsu H for Stanford CS336.

While single-GPU compute (FLOPs/GPU) has seen impressive exponential growth, training the largest LLMs today requires more than a single GPU can offer. The world's fastest supercomputers already operate at exaFLOPs of compute, indicating the need for distributed systems.

[02:43] Limits to GPU-based Scaling - Memory



2:30 Slide on limits to GPU-based scaling, showing performance development graphs and gains.

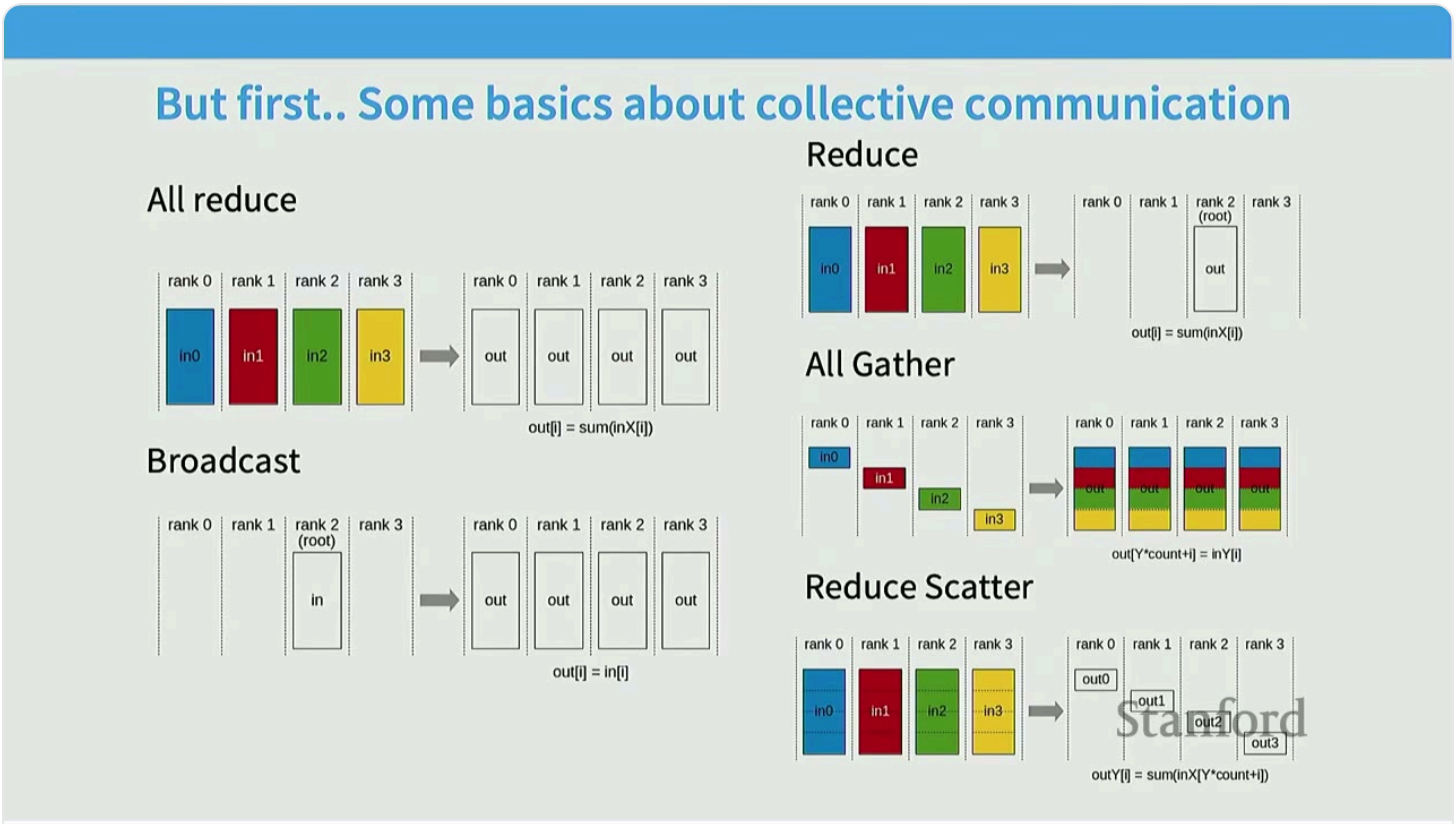
Models are getting extremely large, with billions and even trillions of parameters (e.g., GPT-3 175B, Megatron-Turing NLG 530B). A single GPU often cannot fit these models into its memory. This presents both compute and memory constraints that necessitate multi-machine parallelism.

[03:24] Multi-GPU, Multi-Machine Parallelism Hardware Overview

Modern GPU servers (like NVIDIA DGX systems) are designed with a hierarchy of communication speeds: * **Intra-GPU parallelism (within a single machine)**: GPUs within the same server are connected via very high-speed interconnects (e.g., NVLink, xGMI). This allows for very fast communication between GPUs on the same node. * **Inter-node parallelism (across machines)**: Communication between GPUs on different machines (nodes) is slower, typically going through network switches (e.g., HDR InfiniBand, PCIe Express). This creates a tiered communication latency. * **TPU Networking**: Google's TPUs use a toroidal mesh topology, where each chip primarily communicates with its immediate neighbors very quickly. This design is highly scalable for collective communication operations. In contrast, GPUs often use an all-to-all connection up to a certain number of GPUs (e.g., 256), beyond which communication becomes slower due to hierarchical switching.

This hardware hierarchy dictates how models are effectively parallelized in practice.

[04:57] Basics of Collective Communication



5:24 Slide illustrating basics of collective communication: All Reduce, Broadcast, All Gather, Reduce Scatter.

Collective communication operations are fundamental for distributed training. * **All Reduce**: Each rank (GPU/machine) has an input (`inX`). All inputs are combined (e.g., summed), and the final result (`out`) is copied to all ranks. Cost is roughly `2 * size_of_data` . * **Broadcast**: A root rank's input (`in`) is copied to all other ranks' outputs (`out`). Cost is roughly `1 * size_of_data` . * **Reduce**: Similar to all_reduce, but the combined result (`out`) is only sent to one root rank. * **All Gather**: Each rank's input (`inX`) is appended to form a larger output, and this full combined output is copied to all ranks. * **Reduce Scatter**: Each rank's input (`inX`) is combined (e.g., summed) with parts of other ranks' inputs, and then a *portion* of the combined result is sent to each rank.

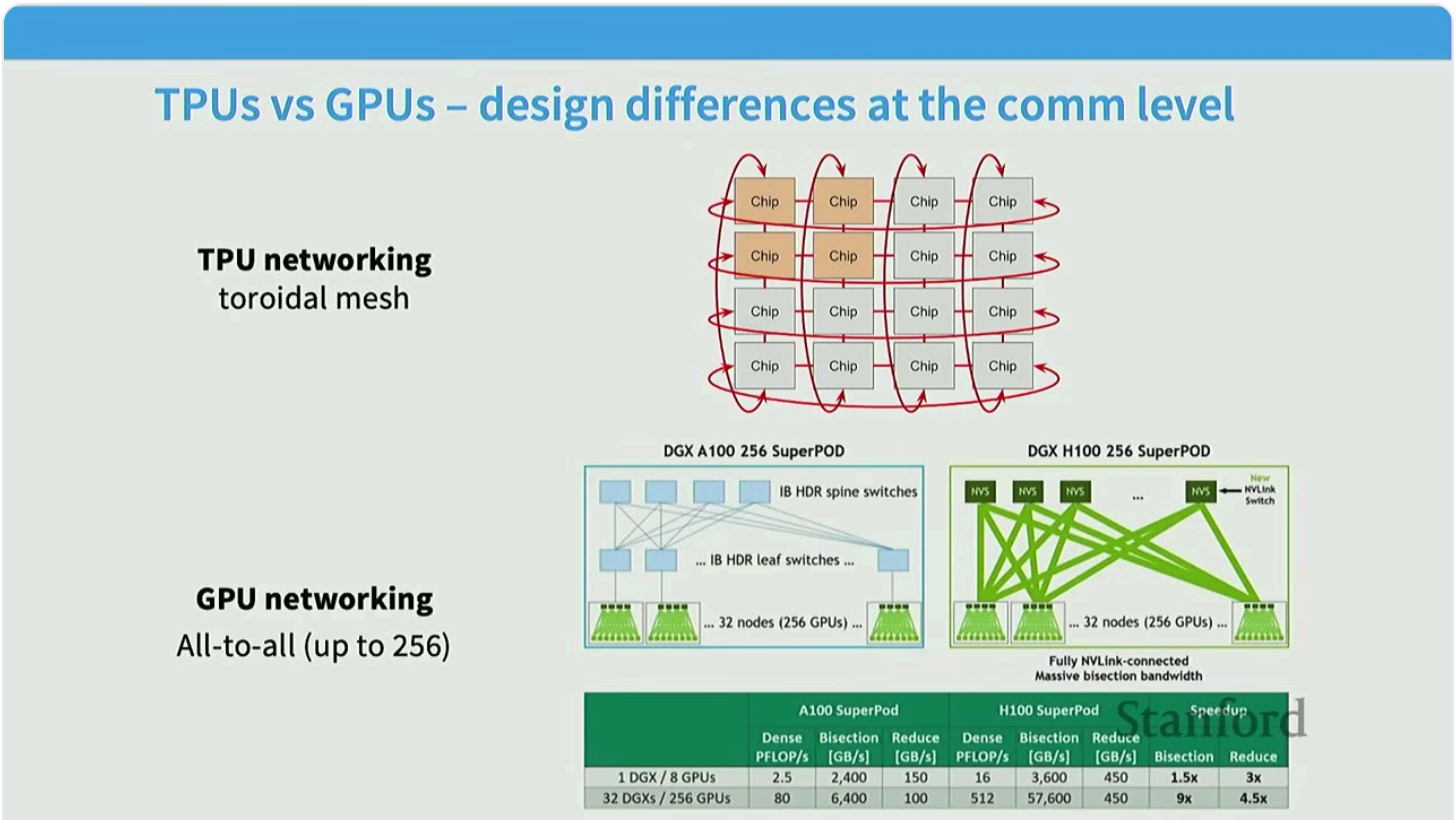
[07:03] Important Detail: all_reduce vs reduce_scatter-gather

An important equivalence: an `all_reduce` operation can be implemented as two steps: a `reduce_scatter` followed by an `all_gather` . * **All Reduce**: All GPUs contribute their data (A, B, C, D), and all GPUs receive the sum (A+B+C+D). * **Reduce-Scatter + All-Gather**: 1. **Reduce-Scatter**: Each GPU sums a *portion* of the data from all other GPUs. For example, GPU0 receives

(A0+B0+C0+D0), GPU1 receives (A1+B1+C1+D1), etc. (where A0 is the first part of A, B0 is the first part of B, etc.). 2. **All-Gather**: Each GPU then broadcasts its combined portion to all other GPUs, so every GPU eventually reconstructs the full sum (A+B+C+D).

Key Insight: In the bandwidth-limited regime, this two-step approach (reduce_scatter + all_gather) is often the best you can do, and it has the same communication cost as a direct all_reduce. This equivalence is crucial for understanding the performance characteristics of parallelization algorithms.

[08:32] TPUs vs GPUs - Design Differences at the Communication Level



	A100 SuperPod			H100 SuperPod			Speedup	
	Dense PFLOP/s	Bisection [GB/s]	Reduce [GB/s]	Dense PFLOP/s	Bisection [GB/s]	Reduce [GB/s]	Bisection	Reduce
1 DGX / 8 GPUs	2.5	2,400	150	16	3,600	450	1.5x	3x
32 DGXs / 256 GPUs	80	6,400	100	512	57,600	450	9x	4.5x

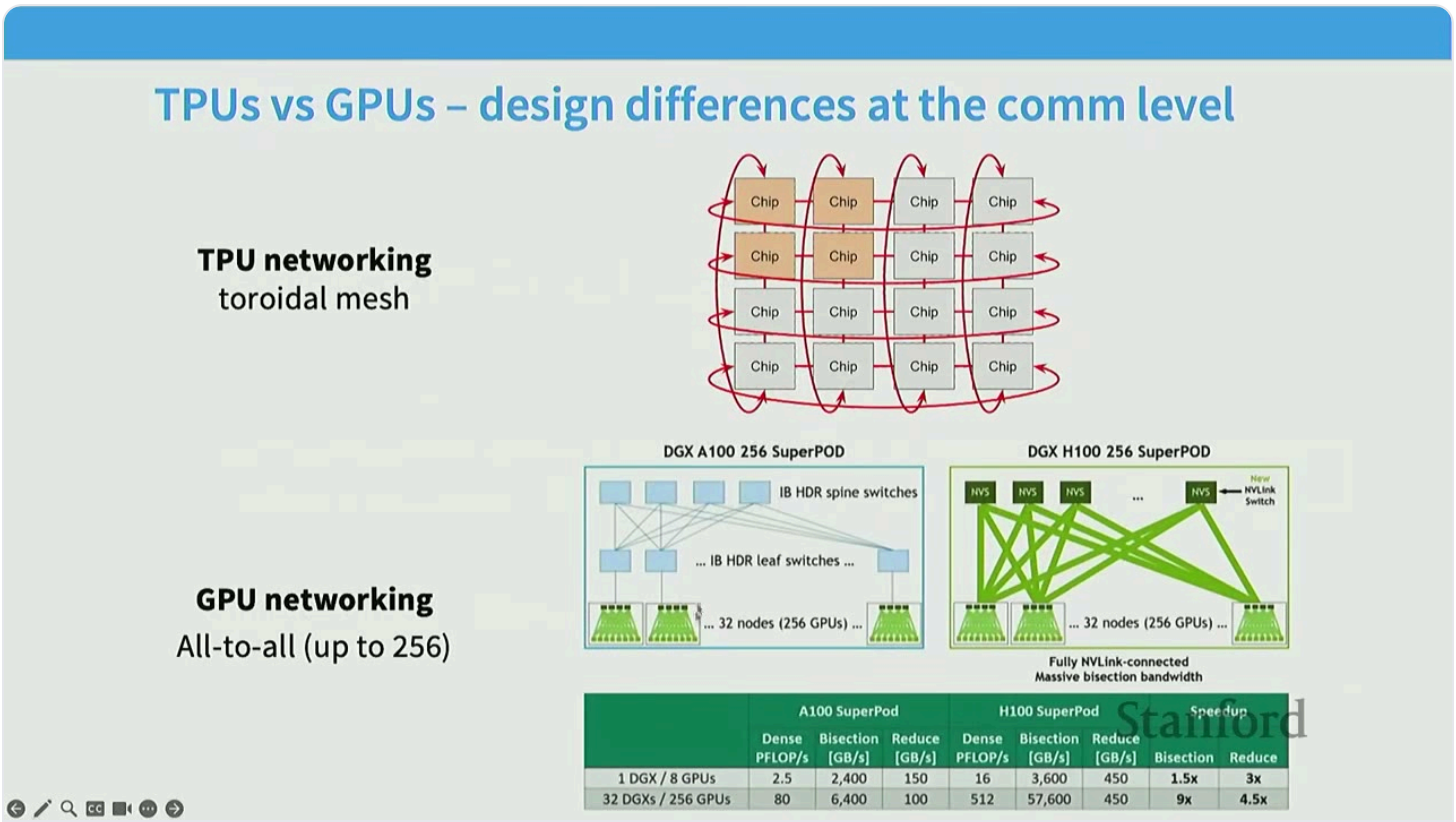
9:59 Slide comparing TPU and GPU networking design differences at the communication level.

- **TPU Networking**: Uses a toroidal mesh, where chips primarily communicate with neighbors. This is highly scalable but limits direct communication to non-neighbors.
- **GPU Networking**: Often uses an all-to-all connection up to ~256 GPUs (within a superpod/rack) via high-bandwidth NVLink/InfiniBand. Beyond this, communication typically goes through slower leaf/spine switches.

This difference implies that for collective communications, TPUs might be more efficient for certain patterns due to their direct neighbor connections, while GPUs excel at all-to-all

communication within their high-bandwidth clusters.

[10:27] Part 1 Recap



11:43 Slide comparing TPU and GPU networking design differences at the communication level.

- **New unit of compute**: The datacenter (not just a single GPU).
- **What we want from multi-machine scaling**:
  - **Linear memory scaling**: Max model parameters scales linearly with the number of GPUs.
  - **Linear compute scaling**: Model FLOPs scale linearly with the number of GPUs.
- **Simple collective comms primitives**: Algorithms are built by combining these primitives, so performance analysis often boils down to counting communication operations.

[12:33] Part 2 - Standard LLM Parallelization Primitives

How do we parallelize LLMs? There are three important ideas:

1. **Data Parallelism (DP)**:
   - Naive data parallel

    - ZeRO levels 1-3
2. **Model Parallelism (MP)**:
    - Pipeline parallel
    - Tensor parallel
3. **Activation Parallelism**:
    - Sequence parallel

These techniques, when combined, provide the tools to scale both compute and memory gracefully across many machines.

[14:17] Naive Data Parallelism

Starting Point: Imagine we are doing naive Stochastic Gradient Descent (SGD).

$$\theta_{t+1} = \theta_t - \eta \nabla_B f(x_i)$$

Naive Parallelism: Split the B-sized batch across M machines. Exchange gradients to synchronize.

How does this do? * **Compute scaling**: Each GPU processes `B/M` examples. This is good if `B` is large enough to saturate GPU compute. * **Communication overhead**: Transmits `2 * #params` every batch (for all_reduce). This is okay if batches are big, as computation can mask communication. * **Memory scaling**: **None**. Every GPU needs `#params` at least (for model weights, gradients, and optimizer states).

[16:10] What's Wrong with Naive Data Parallel? - Memory

Memory is a significant problem. A single GPU often runs out of memory. In naive DP, we copy the entire model parameters to each GPU.

Memory Situation is Terrible: Depending on precision (e.g., BF16/FP32), a single parameter requires multiple bytes for different components: * 2 bytes for FP/BF16 model parameters * 2 bytes for FP/BF16 gradients * 4 bytes for FP32 master weights (the thing you accumulate into SGD) * 4 (or 2) bytes for FP32/BF16 Adam first moment estimates * 4 (or 2) bytes for FP32/BF16 Adam second moment estimates

This means we need ~5 copies of weights, totaling ~16 bytes per parameter. This quickly exhausts GPU memory, especially for large models.

[17:57] ZeRO - Solving the Memory Overhead Issue of DP

Core idea: Split up the expensive parts (state) and use the reduce-scatter equivalence.

Let's visualize memory usage for a 7.5B parameter model distributed over 64 accelerators (GPUs) with naive DP: * **Baseline**: Total memory consumed is ~120GB. This includes parameters (P), gradients (G), and optimizer states (OS). The optimizer states (OS) are the largest component.

ZeRO stages: * **ZeRO Stage 1 (P_os)**: Optimizer state sharding. * **High-level idea**: Split up the optimizer state (first + second moments) across GPUs. Everyone still has the parameters and gradients. Each worker is responsible for updating a subset of parameters (corresponding to its slice). * **Memory consumed**: $(2 + 2 + K)\frac{\Psi}{N_d}$ where K is the optimizer state size, Ψ is total parameters, N_d is number of devices. This reduces memory from 120GB to ~31.4GB. * **How it works**: 1. Everyone computes a full gradient on their subset of the batch. 2. `ReduceScatter` the gradients: Each GPU (rank) receives the sum of gradients for the parameters it *owns*. Communication cost: `2 * #params`. 3. Each machine updates their parameter slice using their gradient + state. 4. `AllGather` the updated parameters: Each GPU broadcasts its updated parameter slice to all other GPUs, so everyone has the full updated model. Communication cost: `2 * #params`. * **Comparison with Naive DDP**: * **Communication primitive**: Naive DDP uses `all_reduce` (one gradient). ZeRO stage 1 uses `reduce_scatter` (gradients) + `all_gather` (parameters). * **Communication cost**: Both are `2 * #params`. * **Memory**: Naive DDP: `(4 + K) * #params`. ZeRO stage 1: `(4 + K/Ngpu) * #params`. * **Key point**: ZeRO stage 1 is **free** in the bandwidth-limited regime (same communication cost as all_reduce) but provides significant memory savings.

- **ZeRO Stage 2 (P_os+g)**: Optimizer state + gradient sharding.
    - **High-level idea**: Also keep the gradients (pink slices) sharded across the machines.
    - **Memory consumed**: $(2 + K)\frac{\Psi}{N_d}$. This reduces memory further to ~16.6GB.
    - **Complexity**: We can never instantiate a full gradient vector. Each worker must compute a full gradient (since we're doing a data parallel run). This means during the backward pass, as soon as a layer's gradients are computed, they must be immediately reduced/scattered to the owning GPU and then freed. This requires incremental computation/communication.
    - **How it works**:
        1. Everyone incrementally goes backward on the computation graph.

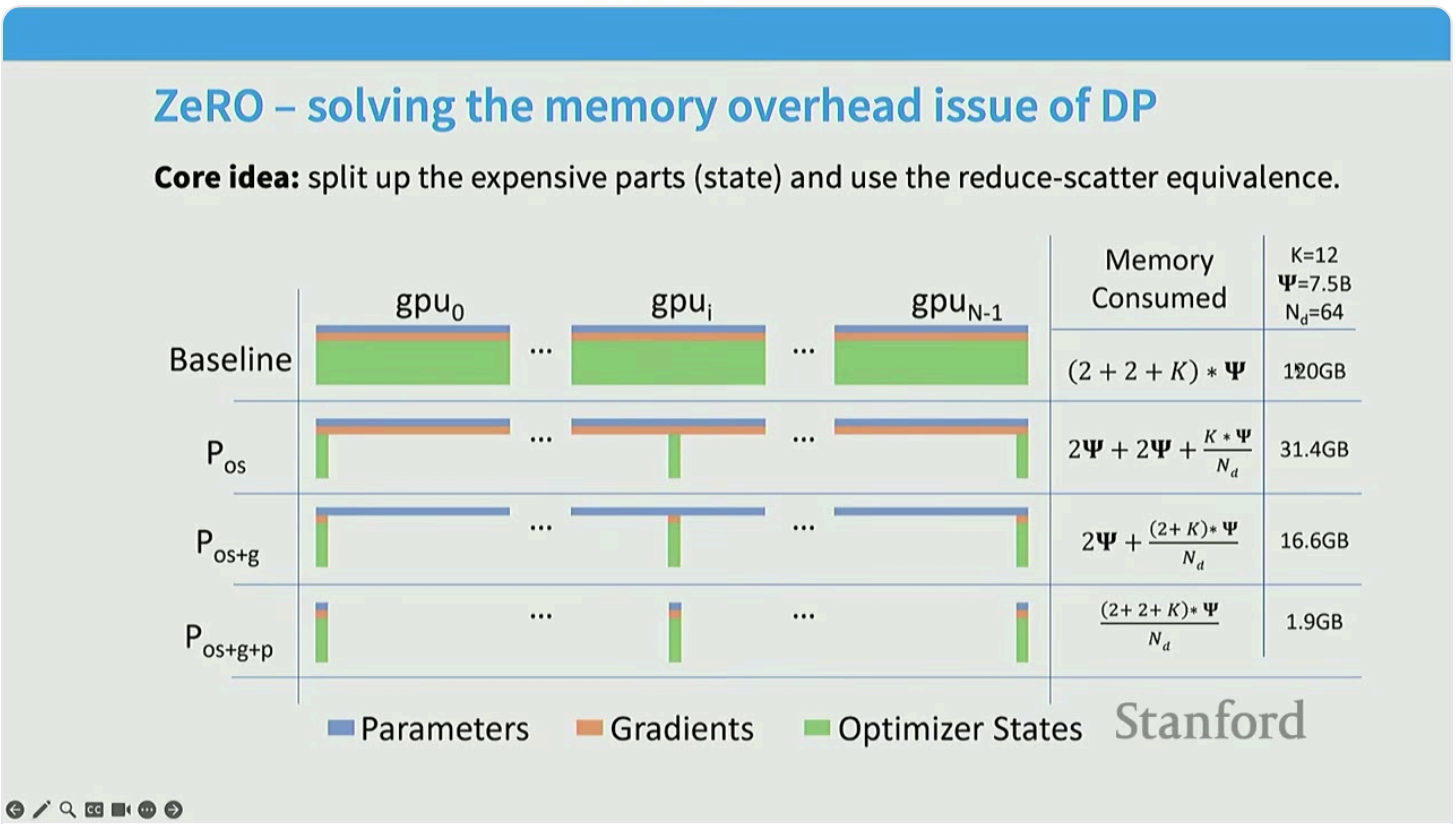
2. After computing a layer's gradients, immediately `reduce_scatter` this to the right worker (the GPU owning that parameter slice).
3. Once gradients are not needed in the backward graph, immediately free them.
4. Each machine updates their parameter using their gradient + state.
5. `AllGather` the parameters.

- **Communication cost**: Still `2 * #params` (total, not per layer).

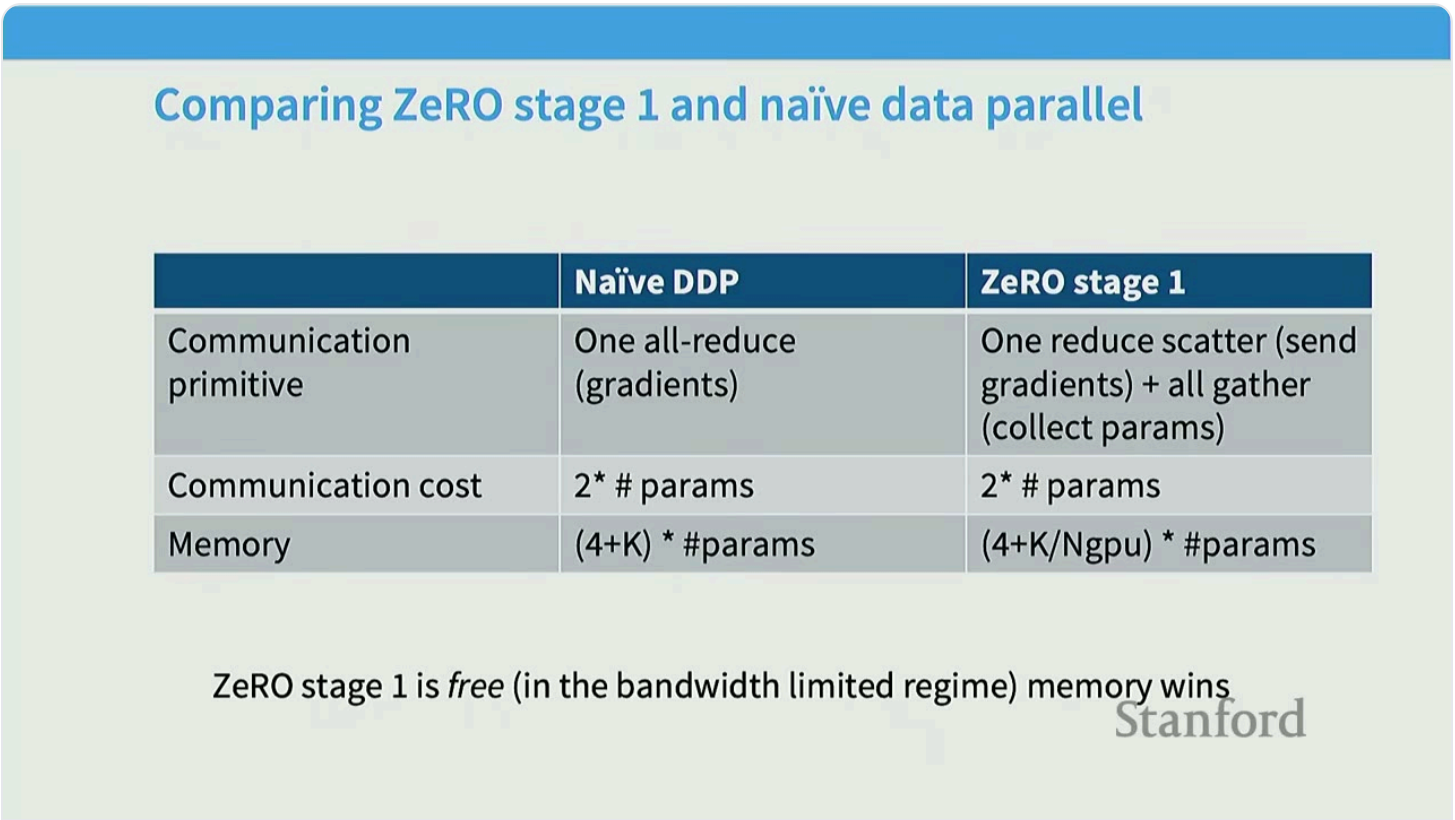
- **ZeRO Stage 3 (P_os+g+p)**: Optimizer state + gradient + parameter sharding (aka FSDP - Fully Sharded Data Parallel).

  - **High-level idea**: Shard everything, including parameters. Use the same incremental communication/computation ideas. Send and request parameters on demand while stepping through the compute graph.
  - **Memory consumed**: $(2 + K)\frac{\Psi}{N_d}$. This reduces memory further to ~1.9GB.
  - **How it works (baby version)**:
    1. **Forward Pass**: For each layer, `all_gather` the weights needed for that layer. Perform the forward computation. Immediately `free` the weights. This happens incrementally layer by layer.
    2. **Backward Pass**: For each layer, `all_gather` the weights. Perform the backward computation. `Reduce_scatter` the gradients to the owning GPU. Immediately `free` the weights and gradients.
    3. **Update Weights**: Each GPU updates its owned parameter slice.
    4. **All-Gather Parameters**: `All_gather` the updated parameters so all GPUs have the full updated model.
  - **Communication cost**: `2 * all_gather` (for all parameters) + `1 * reduce_scatter` (for all parameters). Total `3 * #params`.
  - **Key point**: The surprising part is that this is possible with **low overhead** due to overlapping communication and computation. All-gathers can happen at once while forward computation proceeds, masking the communication cost. This is achieved by queuing requests for weights/gradients before they are actually needed.

[18:28] What's the Point? (ZeRO Summary)



18:40 Slide on ZeRO, solving memory overhead of Data Parallelism, with memory consumption table.



26:49 Table comparing ZeRO stage 1 and naive data parallel communication primitives, costs, and memory.

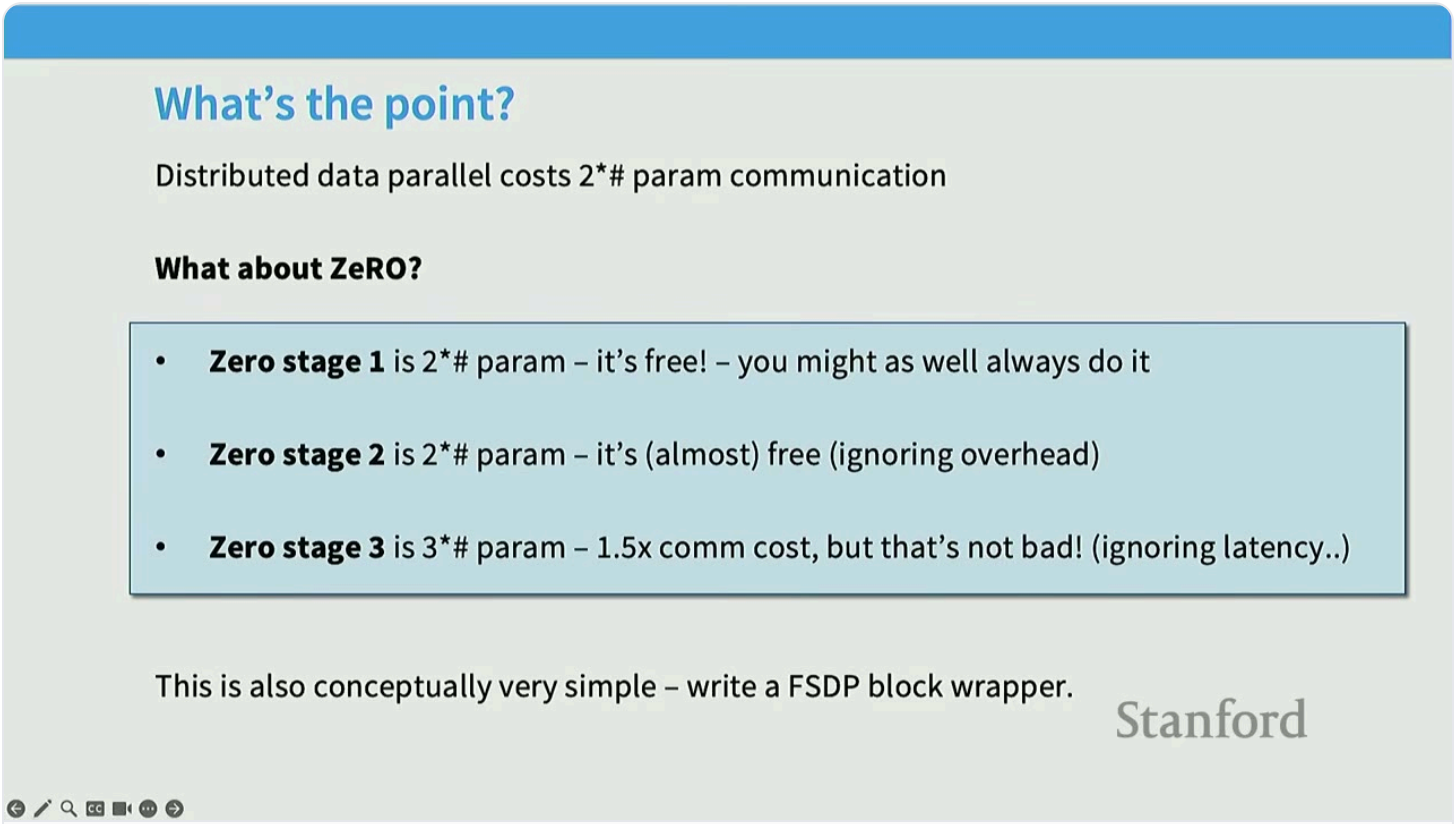
- **Distributed data parallel costs**: `2 * #param` communication (for naive DP).

- **ZeRO Stage 1**: `2 * #param` - it's free! (same comm cost, but memory savings). You might as well always do it.
- **ZeRO Stage 2**: `2 * #param` - almost free (ignoring overhead from incremental processing).
- **ZeRO Stage 3**: `3 * #param` - 1.5x comm cost, but that's not bad! (ignoring latency, which is masked by overlapping).

ZeRO in Practice - Will it fit? On an 8x A100 80GB node: * **Baseline (Naive DP)**: Max model size ~6.66B parameters. Formula for B/param: 12. * **ZeRO Stage 1**: Max model size ~16B parameters. Formula for B/param: 5. * **ZeRO Stage 2**: Max model size ~24.62B parameters. Formula for B/param: 2 (param) + 10 (grad + state) / 8. * **ZeRO Stage 3**: Max model size ~53.33B parameters. Formula for B/param: 12/8.

ZeRO allows fitting significantly larger models into memory by sharding the model state.

[30:31] Issues Remain with Data Parallel - Compute Scaling



38:52 Slide explaining the communication costs of ZeRO stages 1, 2, and 3.

- With data parallel, if `#machines > batch size` (and near this, communication overhead is high), efficiency drops.

- There are diminishing returns to batch sizes for optimization. Beyond a "critical batch size", increasing the batch size yields little benefit for optimization.
- This means data parallel alone cannot achieve arbitrarily large parallelism. Batch size becomes a critical resource that needs to be managed.

[44:00] Issues Remain with Data Parallel - Models Don't Fit

- ZeRO stages 1 and 2 don't scale memory for activations.
- ZeRO stage 3 is good in principle but can be slow and does not reduce activation memory.
- This highlights the need for other parallelism strategies when models become too large to fit even with ZeRO.

[45:14] Beyond Data Parallel - Model Parallelism

What is Model Parallelism? * It splits the parameters across GPUs (like ZeRO3). * But it communicates **activations** (while ZeRO3 sends parameters).

We cover two different types of model parallelism: 1. **Pipeline Parallelism (PP)** 2. **Tensor Parallelism (TP)**

[46:09] Layer-wise Parallelism (Pipeline Parallel)

Concept: Cuts up layers, assigns some subset to GPUs. Activations and partial gradients are passed back and forth. * **Example**: Layer 0 on GPU0, Layer 1 on GPU1, etc. For a forward pass, GPU0 computes Layer 0, sends activations to GPU1. GPU1 computes Layer 1, sends activations to GPU2, and so on. The backward pass reverses this.

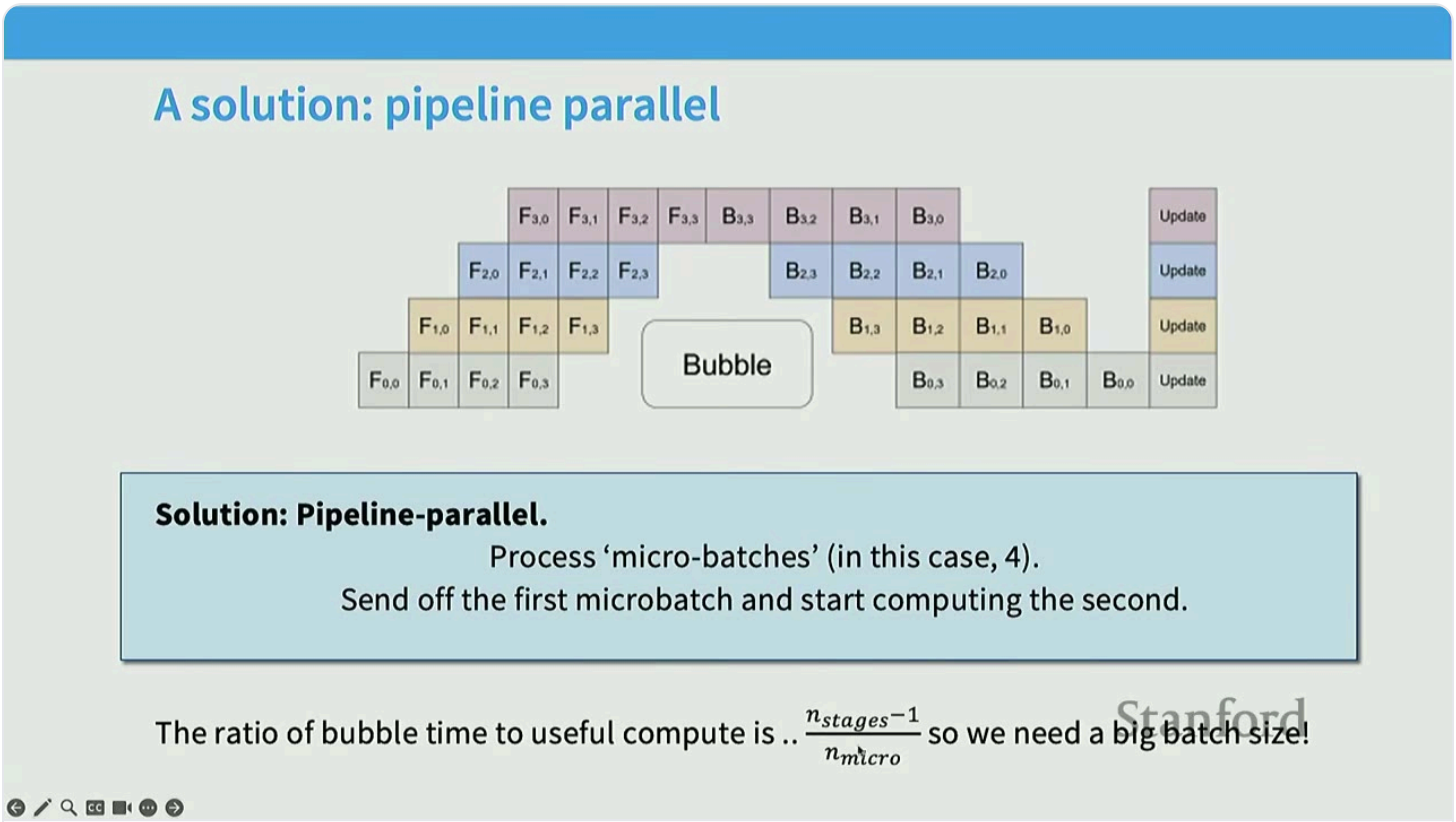
What's wrong with layer-wise parallelism? * **Terrible utilization**: With `N` GPUs, each GPU is active `1/N` of the time. * **The "Bubble"**: GPUs are idle most of the time, waiting for the forward pass to propagate through the pipeline and the backward pass to propagate back. This creates a large "bubble" of idle time.

[47:54] A Solution: Pipeline Parallel (Micro-batching)

Solution: Process 'micro-batches'. * Instead of waiting for the entire batch to clear the pipeline, split the batch into smaller micro-batches. * As soon as the first micro-batch completes a stage,

send its activations to the next GPU and start computing the second micro-batch on the first GPU. * This allows overlapping computation and communication, reducing the bubble size. * **Ratio of bubble time to useful compute**: $\frac{N_{stages}-1}{N_{microbatches}}$. * **Key point**: If you have a large number of micro-batches (which means a large batch size), the bubble can be hidden, leading to better utilization. Batch size becomes a resource that can be spent to improve pipeline parallel efficiency.

[49:27] Why Pipeline Parallel?



49:14 Diagram illustrating pipeline parallel solution with micro-batches and bubble time equation.

Pipelines seem terrible due to the bubble, so why do we use them? 1. **Pipelines save memory (compared to DDP)**: By distributing layers, each GPU only needs to store the parameters and activations for its assigned layers, not the entire model. 2. **Pipelines can have good communication properties (compared to FSDP)**: It depends only on activations (batch_size * sequence_length * hidden_dim) and communication is point-to-point. This can be favorable on slower network links (e.g., inter-node communication across racks or data centers).

Rule of thumb: Generally, pipeline parallel is used on slower network links (i.e., inter-node) as a way to get better memory-wise scaling.

[50:48] Pipeline Performance is Highly Dependent on Batch Size

- **Observation**: With a small batch size (e.g., 8), increasing pipeline parallel size (number of GPUs involved in the pipeline) rapidly degrades throughput per GPU.
- **Observation**: With a large batch size (e.g., 128), throughput per GPU remains relatively flat even with increasing pipeline parallel size.
- **Conclusion**: Batch sizes are key to hiding the bubble. Otherwise, pipeline rapidly degrades performance.

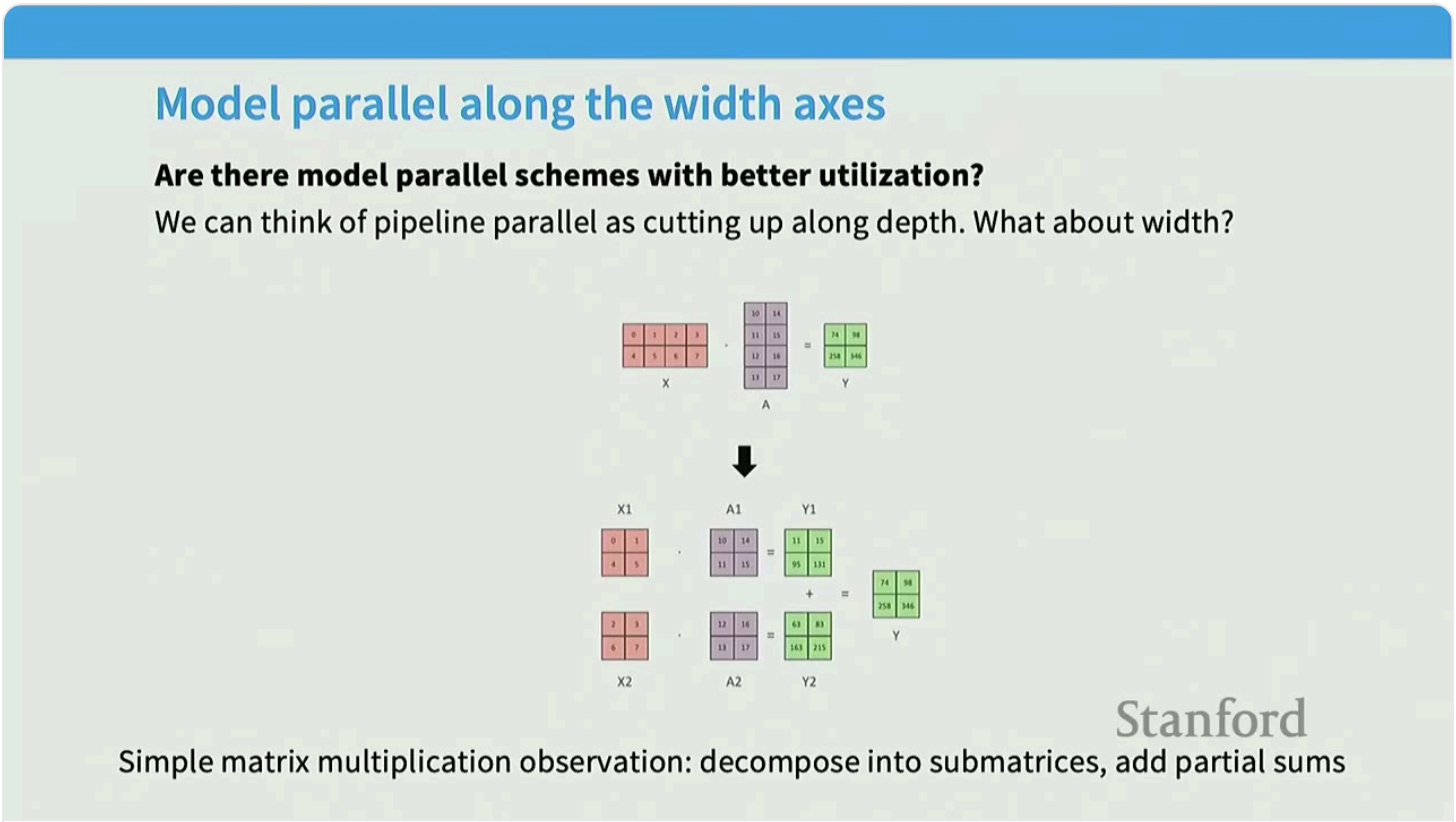
[51:24] Trading Communication Bandwidth for Utilization

More complex pipeline patterns (e.g., interleaving forward and backward passes, assigning multiple stages to each device) can improve utilization, but at the cost of increased communication bandwidth.

[51:54] "Zero Bubble" Pipelining (Dualpipe)

This is an advanced technique to eliminate the bubble in pipeline parallelism. * **Core Idea**: Split the backward pass into two parts: 1. Backpropagating activations (computing $\frac{\partial L}{\partial z}$ and $\frac{\partial L}{\partial x}$). 2. Computing weight gradients (computing $\frac{\partial L}{\partial W}$). * The second part (computing weight gradients) can be done whenever, as it doesn't have serial dependencies on subsequent layers' activation backpropagation. * By carefully rescheduling the weight gradient computations into the idle "bubble" time, utilization can be significantly improved. * **Challenge**: This is extremely complicated to implement, requiring deep intervention in the autodiff system and careful scheduling.

[52:23] Model Parallel Along the Width Axes (Tensor Parallel)



55:40 Diagram showing model parallel along width axes, decomposing matrix multiplication into submatrices.

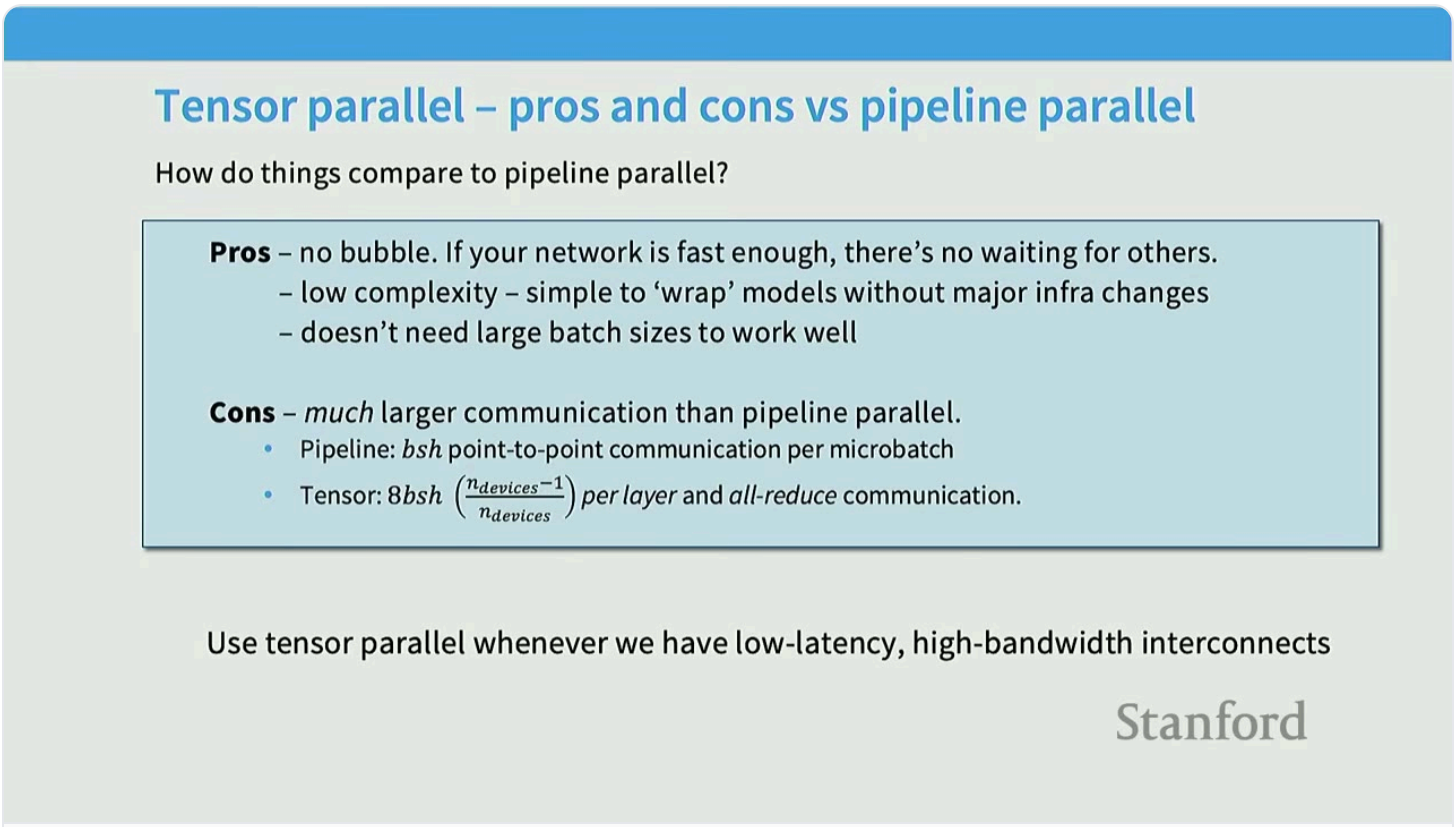
Question: Are there model parallel schemes with better utilization than pipeline parallel? * Pipeline parallel cuts along the depth dimension (layers). What about cutting along the width dimension? * **Observation**: Most of the computation in LLMs is matrix multiplies. * **Tensor Parallel Idea**: Decompose large matrix multiplies into smaller submatrices that can be processed in parallel, and then combine partial sums. * **Example**: For $X \times A = Y$, split A into A_1, A_2 (columns) and X into X_1, X_2 (rows). Then $Y = X_1A_1 + X_2A_2$. * Alternatively, split A into A_1, A_2 (rows) and X into X_1, X_2 (columns). Then $Y = XA_1 + XA_2$. * **Implementation in LLMs (e.g., Transformer layer)**: * Assign columns of weight matrices (A1, A2) and rows of weight matrices (B1, B2) to separate GPUs. * **Forward pass**: Input X is copied to all GPUs. Each GPU computes its part (XA_1 and XA_2). Then, an `all_reduce` is performed to sum the partial results (e.g., for the output of a feed-forward layer). * **Backward pass**: Gradients for the output are copied to all GPUs. Each GPU computes its part of the gradient. Then, an `all_reduce` is performed to sum the partial gradients.

[57:50] When Do We Tensor Parallel?

- **Rule of thumb**: Tensor parallel is typically applied *within a node* (up to 8 GPUs) due to high-speed interconnects (NVLink).

- **Performance**: Tensor parallel is very bandwidth-hungry. While it offers memory scaling, it introduces significant communication overhead.
    - Example from Hugging Face: Throughput per GPU drops by ~10-12% for TP=2 or TP=4.
    - The drop becomes much steeper for higher TP values (e.g., 42.7% for TP=16, 65.6% for TP=32).
- **Conclusion**: Tensor parallel is often limited to the number of GPUs within a single node (e.g., 8 GPUs) to leverage the fastest available communication.

[1:00:44] Tensor Parallel - Pros and Cons vs Pipeline Parallel



62:10 Slide comparing pros and cons of tensor parallel versus pipeline parallel communication.

Pros of Tensor Parallel: * No bubble (unlike PP). * Low complexity: Simple to "wrap" models without major infra changes. * Doesn't need large batch sizes to work well.

Cons of Tensor Parallel: * Much larger communication than pipeline parallel. * **Pipeline**: Batch size * hidden_dim point-to-point communication per microbatch. * **Tensor**: 8 * batch_size * hidden_dim / N_devices per layer and all-reduce communication. * **Rule of thumb**: Use tensor parallel whenever we have low-latency, high-bandwidth interconnects (e.g., within a GPU node).

[1:03:33] A Final Complexity - Memory is Dynamic! (Activation Memory)

Memory isn't just the static bits (parameters, optimizer states), but also dynamic activations. Activations can be very large. * **Observation**: A memory profile of a standard forward/backward pass shows dynamic memory usage. * Parameters (static) and optimizer state (static after iteration 0) form a baseline. * Activations (red, blue) grow during the forward pass and are freed during the backward pass. * Gradients (yellow) accumulate during the backward pass. * **Peak memory**: The peak memory usage often occurs mid-backward pass, where some activations are still present, and gradients are accumulating.

[1:04:58] A Final Complexity - Activation Memory

- Tensor and pipeline parallel can linearly reduce parameter and optimizer state memory.
- **But what about activations?** Activation memory doesn't parallelize as cleanly.
    - As models get bigger, activation memory (green bars) grows significantly, even with aggressive parameter/optimizer state sharding.
    - Some parts of activation memory (e.g., for LayerNorm, Dropout, MLP inputs) don't parallelize well with TP/PP.

[1:05:55] What's the Activation Memory Per Layer?

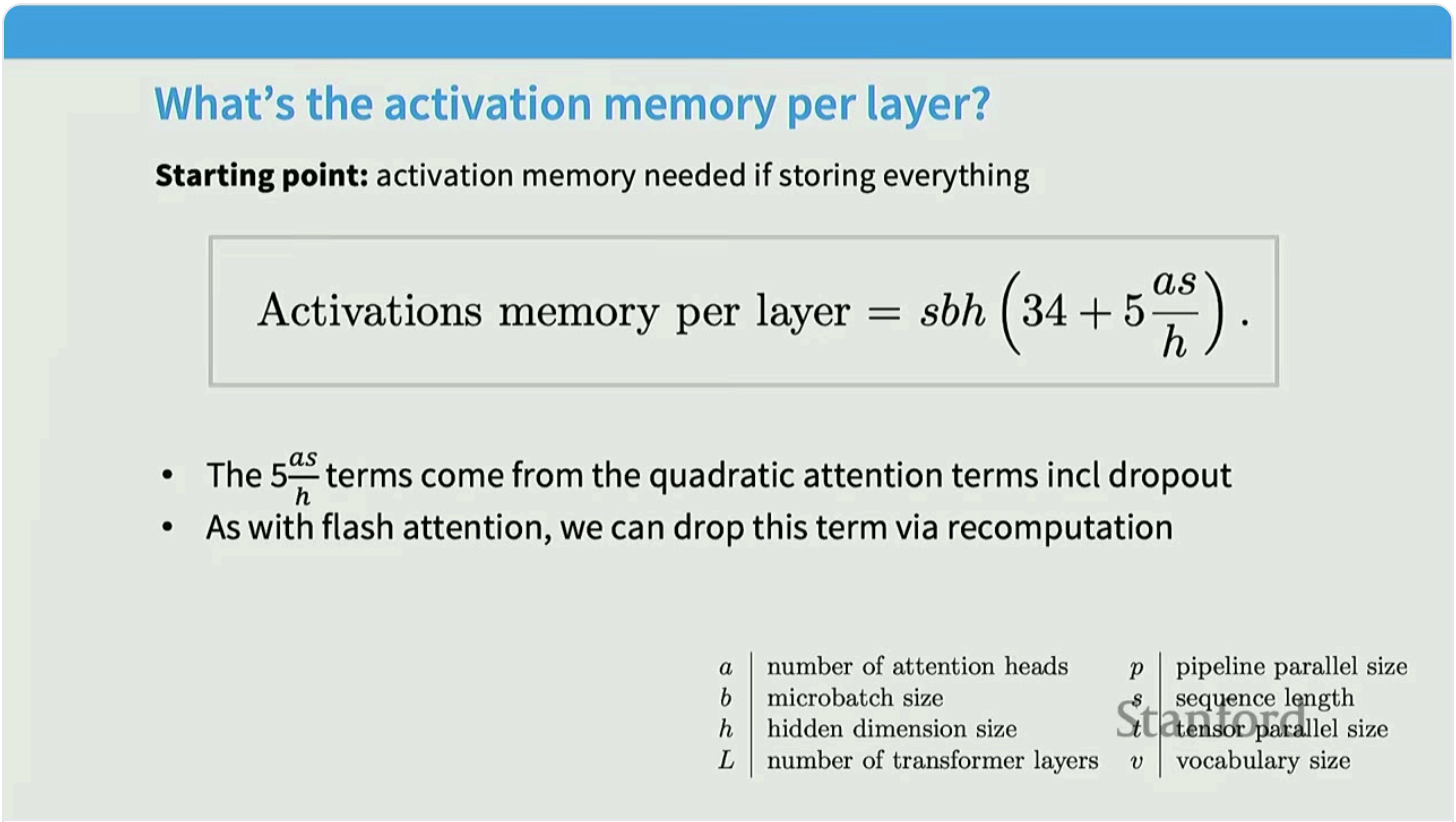
For a transformer layer, the activation memory per layer (if storing everything) is:

$$\text{activations memory per layer} = \text{sbh}\left(34 + \frac{as}{h} + \frac{5}{h}\right)$$

Where: * `s` : sequence length * `b` : microbatch size * `h` : hidden dimension size * `a` : number of attention heads

- The $\frac{5as}{h}$ term comes from the quadratic attention terms (e.g., attention scores, values, keys) and includes dropout.
- The 34 term comes from MLP and other pointwise operations.
- With flash attention, we can drop the $\frac{as}{h}$ term via recomputation.

[1:07:04] Activation Under Tensor Parallel



67:00 Equation for activations memory per layer, with definitions of variables.

If we apply tensor parallel (splitting matrix multiplies in attention + MLP), the activation memory per layer becomes:

$$\text{activations memory per layer} = \text{sbh}\left(10 + \frac{24}{t} + \frac{5}{ht}\right)$$

Where `t` is the number of devices for tensor parallelism. * The remaining `10` term is for LayerNorm (4sbh), Dropout (2sbh), and inputs to the attention and MLP (4sbh). * These terms will continue to grow with size and are not divided by `t`. They represent pointwise operations that don't parallelize well with tensor parallel.

[1:07:59] Making Memory Truly Linear - Sequence Parallel

Observation: All the `10sbh` terms (from the previous slide) are pointwise ops over the sequence. * **Idea**: Split up the layer norm/dropout terms along the sequence axis. This is called **Sequence Parallelism (SP)**. * **Forward pass**: `all_gather` (g) is used to combine results from different sequence chunks. * **Backward pass**: `reduce_scatter` (g') is used to combine gradients for different sequence chunks. * In the backward pass, the two are reversed.

This allows parallelizing the remaining non-matrix multiply components of activation memory.

Putting it together to get full linear scaling for memory: By combining tensor parallel, sequence parallel, and selective activation recomputation (like Flash Attention), the activation memory per transformer layer can be reduced to:

$$\text{sbh}\left(\frac{34}{t}\right)$$

This achieves true linear scaling of activation memory with the number of devices.

[1:13:13] Recap: LLM Parallelism Table

Strategy	Sync Overhead	Memory	Bandwidth	Batch Size	Easy to Use?
DDP/ZeRO1	Per-batch	No scaling	`2 * #param`	Linear	Very
FSDP (ZeRO3)	Per-batch	Linear	`3 * #param`	Linear	Very
Pipeline	Per-pipeline	Linear	Activations	No Impact	NO
Tensor + Seq	Per-transformer block	Linear	`8 * activations per layer + all_reduce`	No Impact	YES

Key takeaway: You have to balance limited resources: memory, bandwidth, batch size.

[1:14:57] Model vs Tensor Parallel (TPU Book)

- **Key Quantity**: Global batch size (divided by GPU).
- **Observation**: If batch size is too small (e.g., <400), no single parallelism scheme is efficient. You're communication bound.
- **Observation**: As batch size increases, different strategies become optimal:
  - Small batch size: FSDP only.
  - Medium batch size: Mixed FSDP + MP (Tensor Parallel).
  - Large batch size: Both FSDP + MP and pure FSDP work well.
  - Very large batch size: Pure FSDP is sufficient.

- This illustrates why you mix parallelism strategies and how batch size influences the optimal choice.

[1:16:42] "3D Parallelism" - Putting It All Together

Simple rules of thumb from the literature: 1. **Until your model fits in memory**: * Use Tensor Parallel up to `N_GPUs_per_machine` (e.g., 8). * Use Pipeline Parallel across machines (or ZeRO-3, depending on bandwidth). 2. **Then until you run out of GPUs**: * Scale the rest of the way with Data Parallel. * If your batch size is small, use gradient accumulation to trade batch size for better communication efficiency.

[1:18:47] Scaling Strategies from Narayanan 2021 (Megatron-LM)

This paper shows how Megatron-LM trained models from 1.7B to 1T parameters using a combination of parallelism strategies. * **Tensor Parallel (TP)**: Starts at 1, then goes up to 8, and caps out at 8. They use TP first. * **Pipeline Parallel (PP)**: Stays at 1 initially, but once models get big enough (can't fit), PP increases to compensate. * **Data Parallel (DP)**: Starts as large as possible (e.g., 32), then slowly decreases as PP increases. DP is used to scale the rest of the way.

They achieve 40-52% of theoretical peak FLOPs, which is very good.

[1:20:09] Careful '3D' Parallelism Gives Linear Gains

- Careful 3D parallelism (combining TP, PP, DP) gives flat achieved TFLOPs/GPU, meaning linear scaling in total FLOPs as more GPUs are added.
- TP=8 is often optimal for throughput per GPU, even with varying batch sizes.

[1:20:48] Activation Recomputation Can Pay for Itself (via Memory)

- Activation recomputation enables larger batch sizes (by reducing activation memory).
- Larger batch sizes, in turn, improve throughput (e.g., by hiding pipeline bubbles or reducing communication overhead).
- So, even though recomputation adds FLOPs, it can pay for itself by allowing larger batches and thus better overall throughput.

[1:21:10] Recent LMs - What Do They Do?

- **Dolma (7B, FSDP)**: Primarily uses FSDP (ZeRO3), likely intra-node.

- **DeepSeek (V3 - 16B)**: Uses ZeRO stage 1 + Tensor Parallel + Sequence Parallel + Pipeline Parallel (16-way PP, 64-way Expert Parallel, 8-way TP).
- **Yi (34B)**: Uses ZeRO stage 1 + Tensor Parallel + Pipeline Parallel. For MOE (Mixture of Experts) models, they use "E-lightning" which replaces TP with Expert Parallelism.
- **Llama 3 405B**: Uses TP=8, CP=1 (Context Parallel, for long context training), PP=16, DP=128. This is a complex network-aware parallelism configuration.

[1:23:09] Side Note - GPU Failures at This Scale!

- Training large models on thousands of GPUs for weeks/months inevitably encounters hardware failures.
- Llama 3 405B reported significant interruptions:
  - Faulty GPU: 30.1%
  - Faulty HBM3 Memory: 17.2%
  - Software Bug: 12.9%
  - Network Switch/Cable: 8.4%
  - Unplanned Host Maintenance: 7.6%
- **Key takeaway**: Fault tolerance and robust checkpointing are crucial for large-scale distributed training. Data corruption (silent GPU failures) is an even scarier problem.

[1:24:20] Recap for the Whole Lecture

- Scaling beyond a certain point requires multi-GPU, multi-node parallelism.
- There's no single simple solution to the parallelism problem; you probably want all 3 approaches (DP, MP, AP).
- There are simple, interpretable rules of thumb for combining different forms of parallelism in practice.

Practical Takeaways

- **Memory is the primary bottleneck**: For large LLMs, fitting the model (parameters, optimizer states, activations) into GPU memory is the first challenge.
- **ZeRO/FSDP is essential for DP**: It allows data parallelism to scale memory by sharding optimizer states, gradients, and even parameters without significant communication

overhead.

- **Model Parallelism for huge models**: When FSDP is not enough, MP (PP or TP) is needed to distribute the model itself across devices.
- **TP for intra-node, PP for inter-node**: Tensor Parallelism is efficient within a node due to high-bandwidth interconnects. Pipeline Parallelism is better for slower inter-node links, especially when memory saving is critical.
- **Batch size is a resource**: It can be used to hide communication latency in PP or to improve optimization efficiency in DP.
- **Activations are a growing concern**: As models and sequence lengths increase, activation memory becomes a major bottleneck, requiring techniques like Sequence Parallelism and Activation Recomputation.
- **3D/4D Parallelism is the norm**: Combining DP, MP (TP/PP), and AP (SP/Recomputation) is necessary for optimal performance and memory efficiency in state-of-the-art LLM training.
- **Fault tolerance is critical**: Hardware failures are inevitable at scale, necessitating robust checkpointing and recovery mechanisms.

Open Questions / Things to Remember

- The choice of parallelism strategy is highly dependent on the specific hardware topology (interconnect bandwidths, latency) and model characteristics (size, depth, sequence length).
- Implementing advanced parallelism (like zero-bubble pipelining) is extremely complex and often requires deep intervention in the framework's autodiff system.
- The trade-offs between communication overhead, memory savings, and compute utilization are constant considerations.
- The "rules of thumb" provide a practical guide, but fine-tuning often involves empirical testing and profiling.
- Data corruption (silent failures) is a hidden danger in large-scale distributed systems, potentially ruining training runs without explicit errors.## TL;DR
- Training large language models (LLMs) necessitates multi-GPU and multi-node parallelism due to memory and compute limitations of single GPUs.
- Parallelism strategies are broadly categorized into Data Parallelism (DP), Model Parallelism (MP), and Activation Parallelism (AP).
- ZeRO (Zero Redundancy Optimizer) is a key DP technique that shards optimizer states, gradients, and parameters across GPUs to save memory, effectively making DP memory-

scalable.
- Model Parallelism (Pipeline and Tensor) splits the model itself across devices, trading communication overhead for memory scalability.
- Activation Parallelism (Sequence Parallelism, Activation Recomputation) addresses the significant memory footprint of intermediate activations, which can become a bottleneck for very large models or long sequences.
- Effective large-scale training often combines multiple parallelism techniques (3D/4D parallelism) to balance memory, compute, and communication costs.

Key Concepts

- Multi-GPU/Multi-node Parallelism
- Collective Communication (all_reduce, reduce_scatter, all_gather, broadcast)
- Data Parallelism (DP)
- ZeRO (Zero Redundancy Optimizer) stages 1, 2, 3 (FSDP)
- Model Parallelism (MP)
- Pipeline Parallelism (PP)
- Tensor Parallelism (TP)
- Activation Parallelism
- Sequence Parallelism (SP)
- Activation Recomputation
- Batch Size as a Resource
- Communication vs. Computation Bound
- Hardware Hierarchy (NVLink, InfiniBand, Toroidal Mesh)

[00:00] Introduction: The Need for Multi-Machine Parallelism

The lecture focuses on multi-machine optimization, specifically parallelism across machines for training huge models.

Goals: * Understand the systems complexities of training huge models. * Explore different parallelization paradigms and why people use multiple approaches at once. * Describe what large-scale training runs often look like.

Outline: * **Part 1**: Basics of networking for LLMs. * **Part 2**: Different forms of parallel LLM training. * **Part 3**: Scaling and training big LMs with parallelism.

[00:05] Limits to GPU-based Scaling - Compute

While single-GPU compute (FLOPs/GPU) has seen impressive exponential growth, training the largest LLMs today requires more than a single GPU can offer. The world's fastest supercomputers already operate at exaFLOPs of compute, indicating the need for distributed systems.

[02:43] Limits to GPU-based Scaling - Memory

Models are getting extremely large, with billions and even trillions of parameters (e.g., GPT-3 175B, Megatron-Turing NLG 530B). A single GPU often cannot fit these models into its memory. This presents both compute and memory constraints that necessitate multi-machine parallelism.

[03:24] Multi-GPU, Multi-Machine Parallelism Hardware Overview

Modern GPU servers (like NVIDIA DGX systems) are designed with a hierarchy of communication speeds: * **Intra-GPU parallelism (within a single machine)**: GPUs within the same server are connected via very high-speed interconnects (e.g., NVLink, xGMI). This allows for very fast communication between GPUs on the same node. * **Inter-node parallelism (across machines)**: Communication between GPUs on different machines (nodes) is slower, typically going through network switches (e.g., HDR InfiniBand, PCIe Express). This creates a tiered communication latency. * **TPU Networking**: Google's TPUs use a toroidal mesh topology, where each chip primarily communicates with its immediate neighbors very quickly. This design is highly scalable but limits direct communication to non-neighbors. In contrast, GPUs often use an all-to-all connection up to a certain number of GPUs (e.g., 256), beyond which communication becomes slower due to hierarchical switching.

This hardware hierarchy dictates how models are effectively parallelized in practice.

[04:57] Basics of Collective Communication

Collective communication operations are fundamental for distributed training. * **All Reduce**: Each rank (GPU/machine) has an input (`inX`). All inputs are combined (e.g., summed), and the final result (`out`) is copied to all ranks. Cost is roughly `2 * size_of_data` . * **Broadcast**: A root

rank's input (`in`) is copied to all other ranks' outputs (`out`). Cost is roughly `1 * size_of_data` . * **Reduce**: Similar to all_reduce, but the combined result (`out`) is only sent to one root rank. * **All Gather**: Each rank's input (`inX`) is appended to form a larger output, and this full combined output is copied to all ranks. * **Reduce Scatter**: Each rank's input (`inX`) is combined (e.g., summed) with parts of other ranks' inputs, and then a *portion* of the combined result is sent to each rank.

[07:03] Important Detail: all_reduce vs reduce_scatter-gather

An important equivalence: an `all_reduce` operation can be implemented as two steps: a `reduce_scatter` followed by an `all_gather` . * **All Reduce**: All GPUs contribute their data (A, B, C, D), and all GPUs receive the sum (A+B+C+D). * **Reduce-Scatter + All-Gather**: 1. **Reduce-Scatter**: Each GPU sums a *portion* of the data from all other GPUs. For example, GPU0 receives (A0+B0+C0+D0), GPU1 receives (A1+B1+C1+D1), etc. (where A0 is the first part of A, B0 is the first part of B, etc.). 2. **All-Gather**: Each GPU then broadcasts its combined portion to all other GPUs, so every GPU eventually reconstructs the full sum (A+B+C+D).

Key Insight: In the bandwidth-limited regime, this two-step approach (reduce_scatter + all_gather) is often the best you can do, and it has the same communication cost as a direct all_reduce. This equivalence is crucial for understanding the performance characteristics of parallelization algorithms.

[08:32] TPUs vs GPUs - Design Differences at the Communication Level

- **TPU Networking**: Uses a toroidal mesh, where chips primarily communicate with neighbors. This is highly scalable but limits direct communication to non-neighbors.
- **GPU Networking**: Often uses an all-to-all connection up to ~256 GPUs (within a superpod/rack) via high-bandwidth NVLink/InfiniBand. Beyond this, communication typically goes through slower leaf/spine switches.

This difference implies that for collective communications, TPUs might be more efficient for certain patterns due to their direct neighbor connections, while GPUs excel at all-to-all communication within their high-bandwidth clusters.

[10:27] Part 1 Recap

- **New unit of compute**: The datacenter (not just a single GPU).

- **What we want from multi-machine scaling**:
  - **Linear memory scaling**: Max model parameters scales linearly with the number of GPUs.
  - **Linear compute scaling**: Model FLOPs scale linearly with the number of GPUs.
- **Simple collective comms primitives**: Algorithms are built by combining these primitives, so performance analysis often boils down to counting communication operations.

[12:33] Part 2 - Standard LLM Parallelization Primitives

How do we parallelize LLMs? There are three important ideas:

1. **Data Parallelism (DP)**:
   - Naive data parallel
   - ZeRO levels 1-3
2. **Model Parallelism (MP)**:
   - Pipeline parallel
   - Tensor parallel
3. **Activation Parallelism**:
   - Sequence parallel

These techniques, when combined, provide the tools to scale both compute and memory gracefully across many machines.

[14:17] Naive Data Parallelism

Starting Point: Imagine we are doing naive Stochastic Gradient Descent (SGD).

$$\theta_{t+1} = \theta_t - \eta \nabla_B f(x_i)$$

Naive Parallelism: Split the B-sized batch across M machines. Exchange gradients to synchronize.

How does this do? * **Compute scaling**: Each GPU processes `B/M` examples. This is good if `B` is large enough to saturate GPU compute. * **Communication overhead**: Transmits `2 * #params` every batch (for all_reduce). This is okay if batches are big, as computation can mask communication. * **Memory scaling**: **None**. Every GPU needs `#params` at least (for model weights, gradients, and optimizer states).

[16:10] What's Wrong with Naive Data Parallel? - Memory

Memory is a significant problem. A single GPU often runs out of memory. In naive DP, we copy the entire model parameters to each GPU.

Memory Situation is Terrible: Depending on precision (e.g., BF16/FP32), a single parameter requires multiple bytes for different components: * 2 bytes for FP/BF16 model parameters * 2 bytes for FP/BF16 gradients * 4 bytes for FP32 master weights (the thing you accumulate into SGD) * 4 (or 2) bytes for FP32/BF16 Adam first moment estimates * 4 (or 2) bytes for FP32/BF16 Adam second moment estimates

This means we need ~5 copies of weights, totaling ~16 bytes per parameter. This quickly exhausts GPU memory, especially for large models.

[17:57] ZeRO - Solving the Memory Overhead Issue of DP

Core idea: Split up the expensive parts (state) and use the reduce-scatter equivalence.

Let's visualize memory usage for a 7.5B parameter model distributed over 64 accelerators (GPUs) with naive DP: * **Baseline**: Total memory consumed is ~120GB. This includes parameters (P), gradients (G), and optimizer states (OS). The optimizer states (OS) are the largest component.

ZeRO stages: * **ZeRO Stage 1 (P_os)**: Optimizer state sharding. * **High-level idea**: Split up the optimizer state (first + second moments) across GPUs. Everyone still has the parameters and gradients. Each worker is responsible for updating a subset of parameters (corresponding to its slice). * **Memory consumed**: $(2 + 2 + K)\frac{\Psi}{N_d}$ where K is the optimizer state size, Ψ is total parameters, N_d is number of devices. This reduces memory from 120GB to ~31.4GB. * **How it works**: 1. Everyone computes a full gradient on their subset of the batch. 2. `ReduceScatter` the gradients: Each GPU (rank) receives the sum of gradients for the parameters it *owns*. Communication cost: `2 * #params`. 3. Each machine updates their parameter slice using their gradient + state. 4. `AllGather` the updated parameters: Each GPU broadcasts its updated parameter slice to all other GPUs, so everyone has the full updated model. Communication cost: `2 * #params`. * **Comparison with Naive DDP**: * **Communication primitive**: Naive DDP uses `all_reduce` (one gradient). ZeRO stage 1 uses `reduce_scatter` (gradients) + `all_gather` (parameters). * **Communication cost**: Both are `2 * #params`. * **Memory**: Naive DDP: `(4 + K) * #params`. ZeRO stage 1: `(4 + K/Ngpu) * #params`. * **Key point**: ZeRO stage 1 is **free** in the bandwidth-limited regime (same communication cost as all_reduce) but provides significant memory savings.

- **ZeRO Stage 2 (P_os+g)**: Optimizer state + gradient sharding.
    - **High-level idea**: Also keep the gradients (pink slices) sharded across the machines.
    - **Memory consumed**: $(2 + K)\frac{\Psi}{N_d}$. This reduces memory further to ~16.6GB.
    - **Complexity**: We can never instantiate a full gradient vector. Each worker must compute a full gradient (since we're doing a data parallel run). This means during the backward pass, as soon as a layer's gradients are computed, they must be immediately reduced/scattered to the owning GPU and then freed. This requires incremental computation/communication.
    - **How it works**:
        1. Everyone incrementally goes backward on the computation graph.
        2. After computing a layer's gradients, immediately `reduce_scatter` this to the right worker (the GPU owning that parameter slice).
        3. Once gradients are not needed in the backward graph, immediately free them.
        4. Each machine updates their parameter using their gradient + state.
        5. `AllGather` the parameters.
    - **Communication cost**: Still `2 * #params` (total, not per layer).
- **ZeRO Stage 3 (P_os+g+p)**: Optimizer state + gradient + parameter sharding (aka FSDP - Fully Sharded Data Parallel).
    - **High-level idea**: Shard everything, including parameters. Use the same incremental communication/computation ideas. Send and request parameters on demand while stepping through the compute graph.
    - **Memory consumed**: $(2 + K)\frac{\Psi}{N_d}$. This reduces memory further to ~1.9GB.
    - **How it works (baby version)**:
        1. **Forward Pass**: For each layer, `all_gather` the weights needed for that layer. Perform the forward computation. Immediately `free` the weights. This happens incrementally layer by layer.
        2. **Backward Pass**: For each layer, `all_gather` the weights. Perform the backward computation. `Reduce_scatter` the gradients to the owning GPU. Immediately `free` the weights and gradients.
        3. **Update Weights**: Each GPU updates its owned parameter slice.
        4. **All-Gather Parameters**: `All_gather` the updated parameters so all GPUs have the full updated model.

    - **Communication cost**: `2 * all_gather` (for all parameters) + `1 * reduce_scatter` (for all parameters). Total `3 * #params`.
    - **Key point**: The surprising part is that this is possible with **low overhead** due to overlapping communication and computation. All-gathers can happen at once while forward computation proceeds, masking the communication cost. This is achieved by queuing requests for weights/gradients before they are actually needed.

[18:28] What's the Point? (ZeRO Summary)

- **Distributed data parallel costs**: `2 * #param` communication (for naive DP).
- **ZeRO Stage 1**: `2 * #param` - it's free! (same comm cost, but memory savings). You might as well always do it.
- **ZeRO Stage 2**: `2 * #param` - almost free (ignoring overhead from incremental processing).
- **ZeRO Stage 3**: `3 * #param` - 1.5x comm cost, but that's not bad! (ignoring latency, which is masked by overlapping).

ZeRO in Practice - Will it fit? On an 8x A100 80GB node: * **Baseline (Naive DP)**: Max model size ~6.66B parameters. Formula for B/param: 12. * **ZeRO Stage 1**: Max model size ~16B parameters. Formula for B/param: 5. * **ZeRO Stage 2**: Max model size ~24.62B parameters. Formula for B/param: 2 (param) + 10 (grad + state) / 8. * **ZeRO Stage 3**: Max model size ~53.33B parameters. Formula for B/param: 12/8.

ZeRO allows fitting significantly larger models into memory by sharding the model state.

[30:31] Issues Remain with Data Parallel - Compute Scaling

- With data parallel, if `#machines > batch size` (and near this, communication overhead is high), efficiency drops.
- There are diminishing returns to batch sizes for optimization. Beyond a "critical batch size", increasing the batch size yields little benefit for optimization.
- This means data parallel alone cannot achieve arbitrarily large parallelism. Batch size becomes a critical resource that needs to be managed.

[44:00] Issues Remain with Data Parallel - Models Don't Fit

- ZeRO stages 1 and 2 don't scale memory for activations.

- ZeRO stage 3 is good in principle but can be slow and does not reduce activation memory.
- This highlights the need for other parallelism strategies when models become too large to fit even with ZeRO.

[45:14] Beyond Data Parallel - Model Parallelism

What is Model Parallelism? * It splits the parameters across GPUs (like ZeRO3). * But it communicates **activations** (while ZeRO3 sends parameters).

We cover two different types of model parallelism: 1. **Pipeline Parallelism (PP)** 2. **Tensor Parallelism (TP)**

[46:09] Layer-wise Parallelism (Pipeline Parallel)

Concept: Cuts up layers, assigns some subset to GPUs. Activations and partial gradients are passed back and forth. * **Example**: Layer 0 on GPU0, Layer 1 on GPU1, etc. For a forward pass, GPU0 computes Layer 0, sends activations to GPU1. GPU1 computes Layer 1, sends activations to GPU2, and so on. The backward pass reverses this.

What's wrong with layer-wise parallelism? * **Terrible utilization**: With `N` GPUs, each GPU is active `1/N` of the time. * **The "Bubble"**: GPUs are idle most of the time, waiting for the forward pass to propagate through the pipeline and the backward pass to propagate back. This creates a large "bubble" of idle time.

[47:54] A Solution: Pipeline Parallel (Micro-batching)

Solution: Process 'micro-batches'. * Instead of waiting for the entire batch to clear the pipeline, split the batch into smaller micro-batches. * As soon as the first micro-batch completes a stage, send its activations to the next GPU and start computing the second micro-batch on the first GPU. * This allows overlapping computation and communication, reducing the bubble size. * **Ratio of bubble time to useful compute**: $\frac{N_{stages}-1}{N_{microbatches}}$. * **Key point**: If you have a large number of micro-batches (which means a large batch size), the bubble can be hidden, leading to better utilization. Batch size becomes a resource that can be spent to improve pipeline parallel efficiency.

[49:27] Why Pipeline Parallel?

Pipelines seem terrible due to the bubble, so why do we use them? 1. **Pipelines save memory (compared to DDP)**: By distributing layers, each GPU only needs to store the parameters and activations for its assigned layers, not the entire model. 2. **Pipelines can have good communication properties (compared to FSDP)**: It depends only on activations (batch_size * sequence_length * hidden_dim) and communication is point-to-point. This can be favorable on slower network links (e.g., inter-node communication across racks or data centers).

Rule of thumb: Generally, pipeline parallel is used on slower network links (i.e., inter-node) as a way to get better memory-wise scaling.

[50:48] Pipeline Performance is Highly Dependent on Batch Size

- **Observation**: With a small batch size (e.g., 8), increasing pipeline parallel size (number of GPUs involved in the pipeline) rapidly degrades throughput per GPU.
- **Observation**: With a large batch size (e.g., 128), throughput per GPU remains relatively flat even with increasing pipeline parallel size.
- **Conclusion**: Batch sizes are key to hiding the bubble. Otherwise, pipeline rapidly degrades performance.

[51:24] Trading Communication Bandwidth for Utilization

More complex pipeline patterns (e.g., interleaving forward and backward passes, assigning multiple stages to each device) can improve utilization, but at the cost of increased communication bandwidth.

[51:54] "Zero Bubble" Pipelining (Dualpipe)

This is an advanced technique to eliminate the bubble in pipeline parallelism. * **Core Idea**: Split the backward pass into two parts: 1. Backpropagating activations (computing $\frac{\partial L}{\partial z}$ and $\frac{\partial L}{\partial x}$). 2. Computing weight gradients (computing $\frac{\partial L}{\partial W}$). * The second part (computing weight gradients) can be done whenever, as it doesn't have serial dependencies on subsequent layers' activation backpropagation. * By carefully rescheduling the weight gradient computations into the idle "bubble" time, utilization can be significantly improved. * **Challenge**: This is extremely complex to implement, requiring deep intervention in the autodiff system and careful scheduling.

[52:23] Model Parallel Along the Width Axes (Tensor Parallel)

Question: Are there model parallel schemes with better utilization than pipeline parallel? * Pipeline parallel cuts along the depth dimension (layers). What about cutting along the width dimension? * **Observation**: Most of the computation in LLMs is matrix multiplies. * **Tensor Parallel Idea**: Decompose large matrix multiplies into smaller submatrices that can be processed in parallel, and then combine partial sums. * **Example**: For $X \times A = Y$, split A into A_1, A_2 (columns) and X into X_1, X_2 (rows). Then $Y = X_1A_1 + X_2A_2$. * Alternatively, split A into A_1, A_2 (rows) and X into X_1, X_2 (columns). Then $Y = XA_1 + XA_2$. * **Implementation in LLMs (e.g., Transformer layer)**: * Assign columns of weight matrices (A1, A2) and rows of weight matrices (B1, B2) to separate GPUs. * **Forward pass**: Input X is copied to all GPUs. Each GPU computes its part (XA_1 and XA_2). Then, an `all_reduce` is performed to sum the partial results (e.g., for the output of a feed-forward layer). * **Backward pass**: Gradients for the output are copied to all GPUs. Each GPU computes its part of the gradient. Then, an `all_reduce` is performed to sum the partial gradients.

[57:50] When Do We Tensor Parallel?

- **Rule of thumb**: Tensor parallel is typically applied *within a node* (up to 8 GPUs) due to high-speed interconnects (NVLink).
- **Performance**: Tensor parallel is very bandwidth-hungry. While it offers memory scaling, it introduces significant communication overhead.
    - Example from Hugging Face: Throughput per GPU drops by ~10-12% for TP=2 or TP=4.
    - The drop becomes much steeper for higher TP values (e.g., 42.7% for TP=16, 65.6% for TP=32).
- **Conclusion**: Tensor parallel is often limited to the number of GPUs within a single node (e.g., 8 GPUs) to leverage the fastest available communication.

[1:00:44] Tensor Parallel - Pros and Cons vs Pipeline Parallel

Pros of Tensor Parallel: * No bubble (unlike PP). * Low complexity: Simple to "wrap" models without major infra changes. * Doesn't need large batch sizes to work well.

Cons of Tensor Parallel: * Much larger communication than pipeline parallel. * **Pipeline**: Batch size * sequence_length * hidden_dim point-to-point communication per microbatch. * **Tensor**: 8 * batch_size * hidden_dim / N_devices per layer and all-reduce communication. * **Rule of thumb**: Use tensor parallel whenever we have low-latency, high-bandwidth interconnects (e.g., within a GPU node).

[1:03:33] A Final Complexity - Memory is Dynamic! (Activation Memory)

Memory isn't just the static bits (parameters, optimizer states), but also dynamic activations. Activations can be very large. * **Observation**: A memory profile of a standard forward/backward pass shows dynamic memory usage. * Parameters (static) and optimizer state (static after iteration 0) form a baseline. * Activations (red, blue) grow during the forward pass and are freed during the backward pass. * Gradients (yellow) accumulate during the backward pass. * **Peak memory**: The peak memory usage often occurs mid-backward pass, where some activations are still present, and gradients are accumulating.

[1:04:58] A Final Complexity - Activation Memory

- Tensor and pipeline parallel can linearly reduce parameter and optimizer state memory.
- **But what about activations?** Activation memory doesn't parallelize as cleanly.
    - As models get bigger, activation memory (green bars) grows significantly, even with aggressive parameter/optimizer state sharding.
    - Some parts of activation memory (e.g., for LayerNorm, Dropout, MLP inputs) don't parallelize well with TP/PP.

[1:05:55] What's the Activation Memory Per Layer?

For a transformer layer, the activation memory per layer (if storing everything) is:

$$\text{activations memory per layer} = \text{sbh}\left(34 + \frac{as}{h} + \frac{5}{h}\right)$$

Where: * `s` : sequence length * `b` : microbatch size * `h` : hidden dimension size * `a` : number of attention heads

- The $\frac{5as}{h}$ term comes from the quadratic attention terms (e.g., attention scores, values, keys) and includes dropout.
- The 34 term comes from MLP and other pointwise operations.
- With flash attention, we can drop the $\frac{as}{h}$ term via recomputation.

[1:07:04] Activation Under Tensor Parallel

If we apply tensor parallel (splitting matrix multiplies in attention + MLP), the activation memory per layer becomes:

$$\text{activations memory per layer} = \text{sbh}\left(10 + \frac{24}{t} + \frac{5}{ht}\right)$$

Where `t` is the number of devices for tensor parallelism. * The remaining `10` term is for LayerNorm (4sbh), Dropout (2sbh), and inputs to the attention and MLP (4sbh). * These terms will continue to grow with size and are not divided by `t`. They represent pointwise operations that don't parallelize well with tensor parallel.

[1:07:59] Making Memory Truly Linear - Sequence Parallel

Observation: All the `10sbh` terms (from the previous slide) are pointwise ops over the sequence. * **Idea**: Split up the layer norm/dropout terms along the sequence axis. This is called **Sequence Parallelism (SP)**. * **Forward pass**: `all_gather` (g) is used to combine results from different sequence chunks. * **Backward pass**: `reduce_scatter` (g') is used to combine gradients for different sequence chunks. * In the backward pass, the two are reversed.

This allows parallelizing the remaining non-matrix multiply components of activation memory.

Putting it together to get full linear scaling for memory: By combining tensor parallel, sequence parallel, and selective activation recomputation (like Flash Attention), the activation memory per transformer layer can be reduced to:

$$\text{sbh}\left(\frac{34}{t}\right)$$

This achieves true linear scaling of activation memory with the number of devices.

[1:13:13] Recap: LLM Parallelism Table

Strategy	Sync Overhead	Memory	Bandwidth	Batch Size	Easy to Use?
DDP/ZeRO1	Per-batch	No scaling	`2 * #param`	Linear	Very
FSDP (ZeRO3)	Per-batch	Linear	`3 * #param`	Linear	Very

Strategy	Sync Overhead	Memory	Bandwidth	Batch Size	Easy to Use?
Pipeline	Per-pipeline	Linear	Activations	No Impact	NO
Tensor + Seq	Per-transformer block	Linear	`8 * activations per layer + all_reduce`	No Impact	YES

Key takeaway: You have to balance limited resources: memory, bandwidth, batch size.

[1:14:57] Model vs Tensor Parallel (TPU Book)

- **Key Quantity**: Global batch size (divided by GPU).
- **Observation**: If batch size is too small (e.g., <400), no single parallelism scheme is efficient. You're communication bound.
- **Observation**: As batch size increases, different strategies become optimal:
  - Small batch size: FSDP only.
  - Medium batch size: Mixed FSDP + MP (Tensor Parallel).
  - Large batch size: Both FSDP + MP and pure FSDP work well.
  - Very large batch size: Pure FSDP is sufficient.
- This illustrates why you mix parallelism strategies and how batch size influences the optimal choice.

[1:16:42] "3D Parallelism" - Putting It All Together

Simple rules of thumb from the literature: 1. **Until your model fits in memory**: * Use Tensor Parallel up to `N_GPUs_per_machine` (e.g., 8). * Use Pipeline Parallel across machines (or ZeRO-3, depending on bandwidth). 2. **Then until you run out of GPUs**: * Scale the rest of the way with Data Parallel. * If your batch size is small, use gradient accumulation to trade batch size for better communication efficiency.

[1:18:47] Scaling Strategies from Narayanan 2021 (Megatron-LM)

This paper shows how Megatron-LM trained models from 1.7B to 1T parameters using a combination of parallelism strategies. * **Tensor Parallel (TP)**: Starts at 1, then goes up to 8, and caps out at 8. They use TP first. * **Pipeline Parallel (PP)**: Stays at 1 initially, but once models get

big enough (can't fit), PP increases to compensate. * **Data Parallel (DP)**: Starts as large as possible (e.g., 32), then slowly decreases as PP increases. DP is used to scale the rest of the way.

They achieve 40-52% of theoretical peak FLOPs, which is very good.

[1:20:09] Careful '3D' Parallelism Gives Linear Gains

- Careful 3D parallelism (combining TP, PP, DP) gives flat achieved TFLOPs/GPU, meaning linear scaling in total FLOPs as more GPUs are added.
- TP=8 is often optimal for throughput per GPU, even with varying batch sizes.

[1:20:48] Activation Recomputation Can Pay for Itself (via Memory)

- Activation recomputation enables larger batch sizes (by reducing activation memory).
- Larger batch sizes, in turn, improve throughput (e.g., by hiding pipeline bubbles or reducing communication overhead).
- So, even though recomputation adds FLOPs, it can pay for itself by allowing larger batches and thus better overall throughput.

[1:21:10] Recent LMs - What Do They Do?

- **Dolma (7B, FSDP)**: Primarily uses FSDP (ZeRO3), likely intra-node.
- **DeepSeek (V3 - 16B)**: Uses ZeRO stage 1 + Tensor Parallel + Sequence Parallel + Pipeline Parallel (16-way PP, 64-way Expert Parallel, 8-way TP).
- **Yi (34B)**: Uses ZeRO stage 1 + Tensor Parallel + Pipeline Parallel. For MOE (Mixture of Experts) models, they use "E-lightning" which replaces TP with Expert Parallelism.
- **Llama 3 405B**: Uses TP=8, CP=1 (Context Parallel, for long context training), PP=16, DP=128. This is a complex network-aware parallelism configuration.

[1:23:09] Side Note - GPU Failures at This Scale!

- Training large models on thousands of GPUs for weeks/months inevitably encounters hardware failures.
- Llama 3 405B reported significant interruptions:
    - Faulty GPU: 30.1%
    - Faulty HBM3 Memory: 17.2%
    - Software Bug: 12.9%
    - Network Switch/Cable: 8.4%

    - Unplanned Host Maintenance: 7.6%
- **Key takeaway**: Fault tolerance and robust checkpointing are crucial for large-scale distributed training. Data corruption (silent GPU failures) is an even scarier problem.

[1:23:47] Gemma 2

- **Gemma 2 (9B, 27B)**: Uses ZeRO3 + Model Parallelism + Data Parallelism. For TPUs, they leverage the TPUv4 and TPUv5p models. They use 512-way data replication and 1-way sharding. The optimizer state is further sharded using techniques similar to ZeRO-3.

[1:24:20] Recap for the Whole Lecture

- Scaling beyond a certain point is going to require multi-GPU, multi-node parallelism.
- There's no single simple solution to the parallelism problem; you probably want all 3 approaches (DP, MP, AP).
- There are simple and interpretable rules of thumb for combining different forms of parallelism in practice.

Practical Takeaways

- **Memory is the primary bottleneck**: For large LLMs, fitting the model (parameters, optimizer states, activations) into GPU memory is the first challenge.
- **ZeRO/FSDP is essential for DP**: It allows data parallelism to scale memory by sharding optimizer states, gradients, and even parameters without significant communication overhead.
- **Model Parallelism for huge models**: When FSDP is not enough, MP (PP or TP) is needed to distribute the model itself across devices.
- **TP for intra-node, PP for inter-node**: Tensor Parallelism is efficient within a node due to high-bandwidth interconnects. Pipeline Parallelism is better for slower inter-node links, especially when memory saving is critical.
- **Batch size is a resource**: It can be used to hide communication latency in PP or to improve optimization efficiency in DP.
- **Activations are a growing concern**: As models and sequence lengths increase, activation memory becomes a major bottleneck, requiring techniques like Sequence Parallelism and

Activation Recomputation.

- **3D/4D Parallelism is the norm**: Combining DP, MP (TP/PP), and AP (SP/Recomputation) is necessary for optimal performance and memory efficiency in state-of-the-art LLM training.
- **Fault tolerance is critical**: Hardware failures are inevitable at scale, necessitating robust checkpointing and recovery mechanisms.

Open Questions / Things to Remember

- The choice of parallelism strategy is highly dependent on the specific hardware topology (interconnect bandwidths, latency) and model characteristics (size, depth, sequence length).
- Implementing advanced parallelism (like zero-bubble pipelining) is extremely complex and often requires deep intervention in the framework's autodiff system.
- The trade-offs between communication overhead, memory savings, and compute utilization are constant considerations.
- The "rules of thumb" provide a practical guide, but fine-tuning often involves empirical testing and profiling.
- Data corruption (silent failures) is a hidden danger in large-scale distributed systems, potentially ruining training runs without explicit errors.

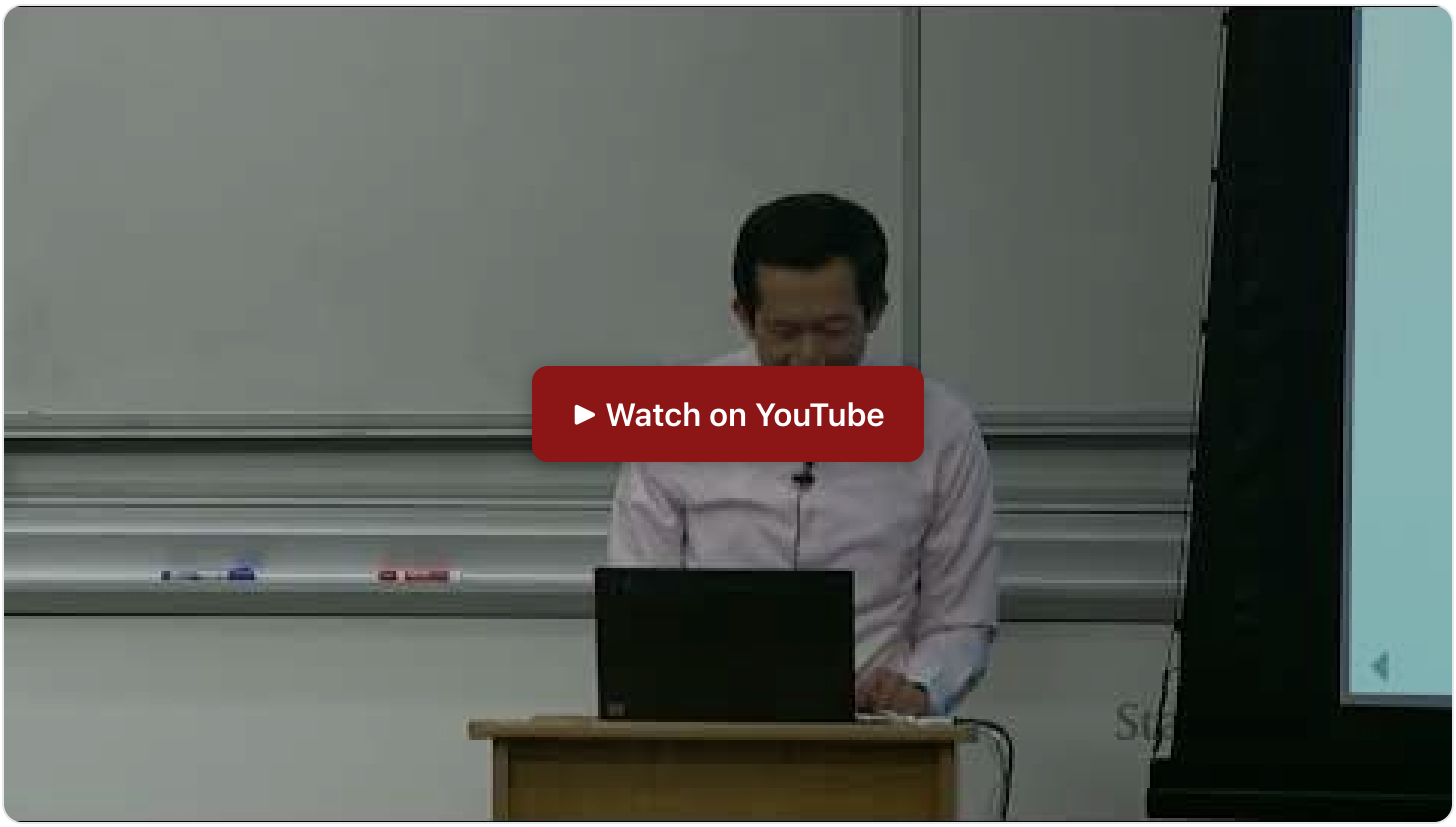
← Lec 06

Lec 08 →

Lecture 08: Parallelism 2

Stanford CS336 · Spring 2025 · 1:15:10 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



TL;DR

- **Data Transfer Bottleneck**: The primary challenge in distributed training is minimizing data transfer bottlenecks, as moving data between different memory hierarchies (L1 cache, HBM, NVLink, NVSwitch, Ethernet) is significantly slower than computation.
- **Collective Operations**: These are fundamental primitives (broadcast, scatter, gather, reduce, all-gather, reduce-scatter) for distributed programming, abstracting complex point-to-point communications across multiple nodes and GPUs.
- **Hardware Evolution**: Modern GPU architectures (like NVIDIA's H100 with NVLink and NVSwitch) are designed to bypass CPUs and Ethernet for direct, high-bandwidth GPU-to-

GPU communication, significantly improving performance for deep learning workloads.

- **Software Stack**: NVIDIA's NCCL (Collective Communication Library) translates high-level collective operations into low-level hardware-optimized packet transfers. PyTorch's `torch.distributed` library provides a Pythonic interface to NCCL and supports various backends (NCCL for GPU, Gloo for CPU).
- **Distributed Training Strategies**: Data parallelism, tensor parallelism, and pipeline parallelism are different ways to shard models and data. Data parallelism cuts data along the batch dimension, tensor parallelism cuts models along the hidden dimension, and pipeline parallelism cuts models along layers.
- **Trade-offs**: Implementing distributed training involves managing complex tradeoffs between recomputation vs. memory/communication, and balancing computational efficiency with architectural complexity. Higher-level frameworks (like Jax/TPUs or PyTorch's FSDP) abstract much of this complexity, allowing users to focus on model definition.

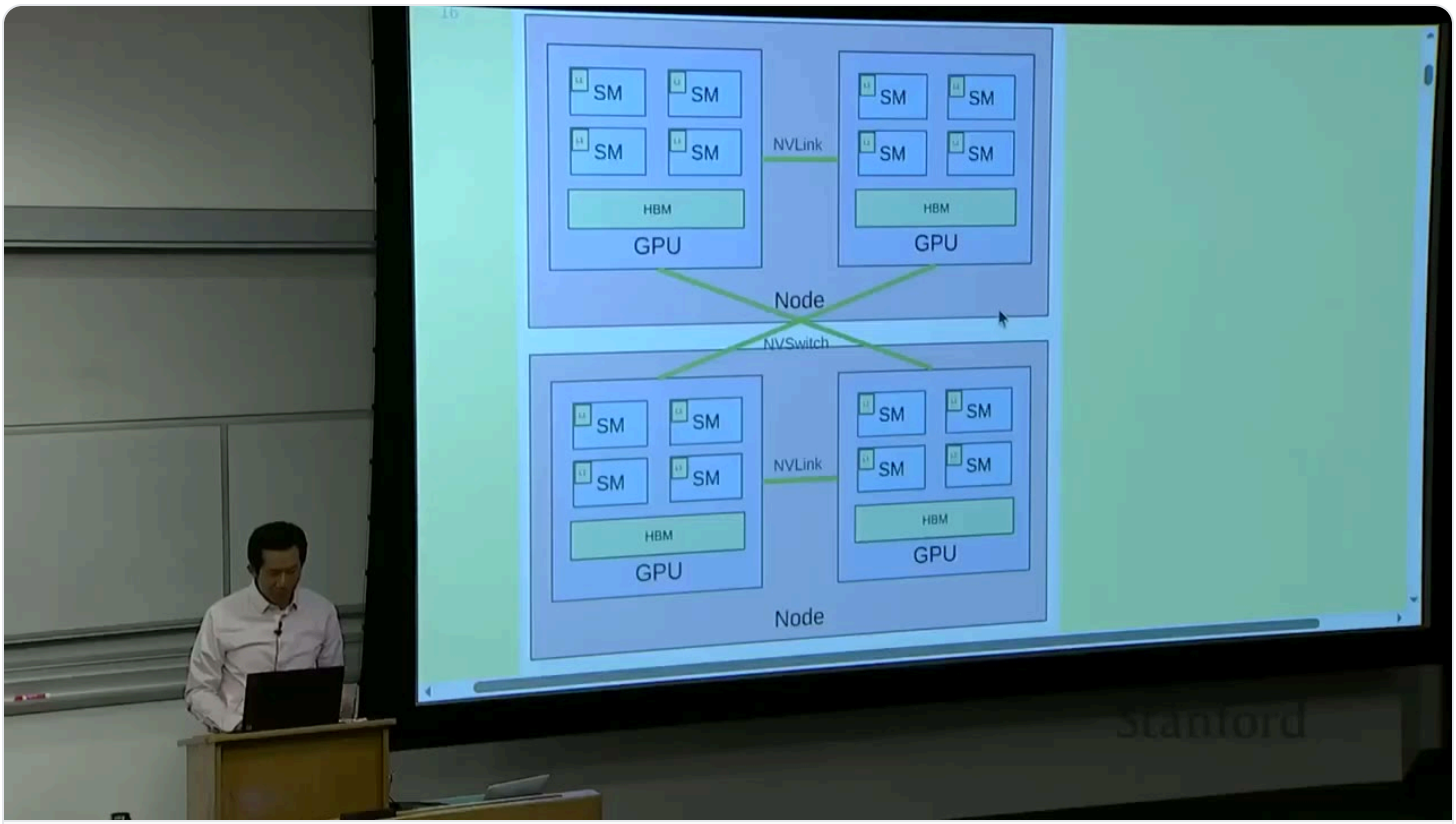
Key Concepts

- Data transfer bottleneck
- Arithmetic intensity
- L1 cache, HBM, NVLink, NVSwitch, Ethernet
- Collective operations (broadcast, scatter, gather, reduce, all-gather, reduce-scatter)
- World size, rank
- NCCL (NVIDIA Collective Communication Library)
- `torch.distributed` (PyTorch distributed library)
- Data parallelism (DDP)
- Tensor parallelism
- Pipeline parallelism
- Communication/computation overlap
- Activation checkpointing
- Hardware-software co-design
- Jax/TPUs and compiler-driven sharding

[0:00] Introduction: Optimizing Deep Learning Workloads

This is week two of the systems lecture, focusing on leveraging hardware to accelerate model training. Last week, we discussed parallelism within a single GPU. This week, the topic is parallelism across multiple GPUs.

[0:23] Hardware Architecture Overview



0:29 Diagram of multi-node GPU architecture with NVLink and NVSwitch connections.

The fundamental hardware setup involves: - **Nodes**: These are individual computers, typically containing multiple GPUs (e.g., 8). - **GPUs**: Each GPU contains several Streaming Multiprocessors (SMs). - **SMs**: These are the arithmetic logic units (ALUs) that perform the actual computations.

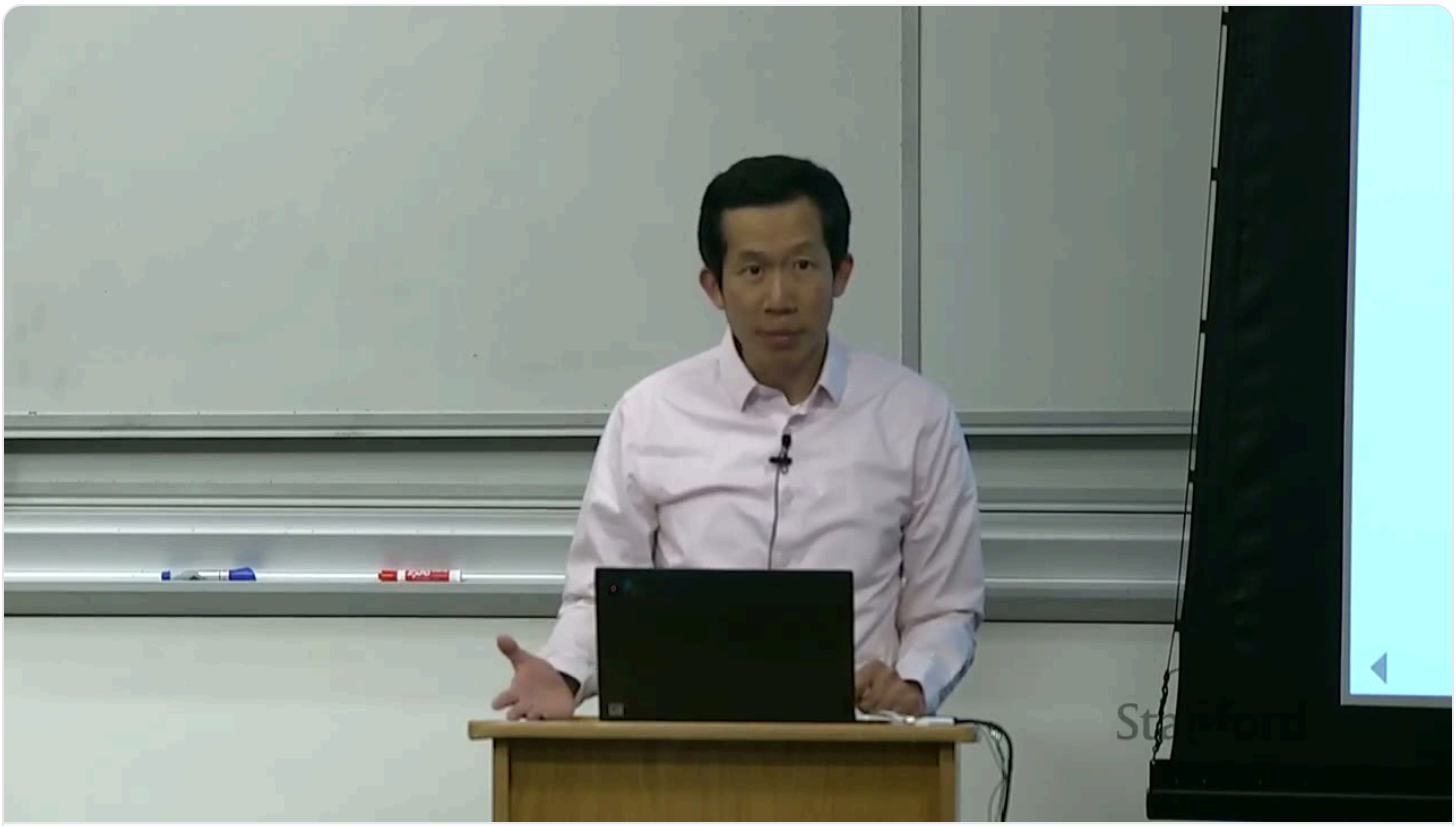
Memory and Communication Hierarchy (from small/fast to big/slow): - **Within an SM**: Very small L1 cache. - **On a GPU**: High Bandwidth Memory (HBM), which is larger than L1 cache. - **Between GPUs on the same node**: NVLink connects GPUs directly. - **Between GPUs on different nodes**: NVSwitch connects GPUs directly (bypassing CPU and Ethernet).

The core idea is that compute happens within the SMs, and these computations require inputs and produce outputs. The further away the data is from the SM, the slower the access. The goal is to structure computations to avoid data transfer bottlenecks and keep arithmetic intensity high, saturating the GPUs.

Last week, techniques like fusion and tiling were discussed to reduce memory access within a single GPU by loading data into faster, local caches (L1, shared memory) and performing multiple operations before writing back to HBM.

This week, the focus shifts to reducing communication across GPUs and nodes through replication and sharding of models, parameters, and optimizer states.

[1:02] Unifying Theme: Orchestrate Computation to Avoid Data Transfer Bottlenecks



1:42 Professor presenting, gesturing with his right hand, looking at the audience.

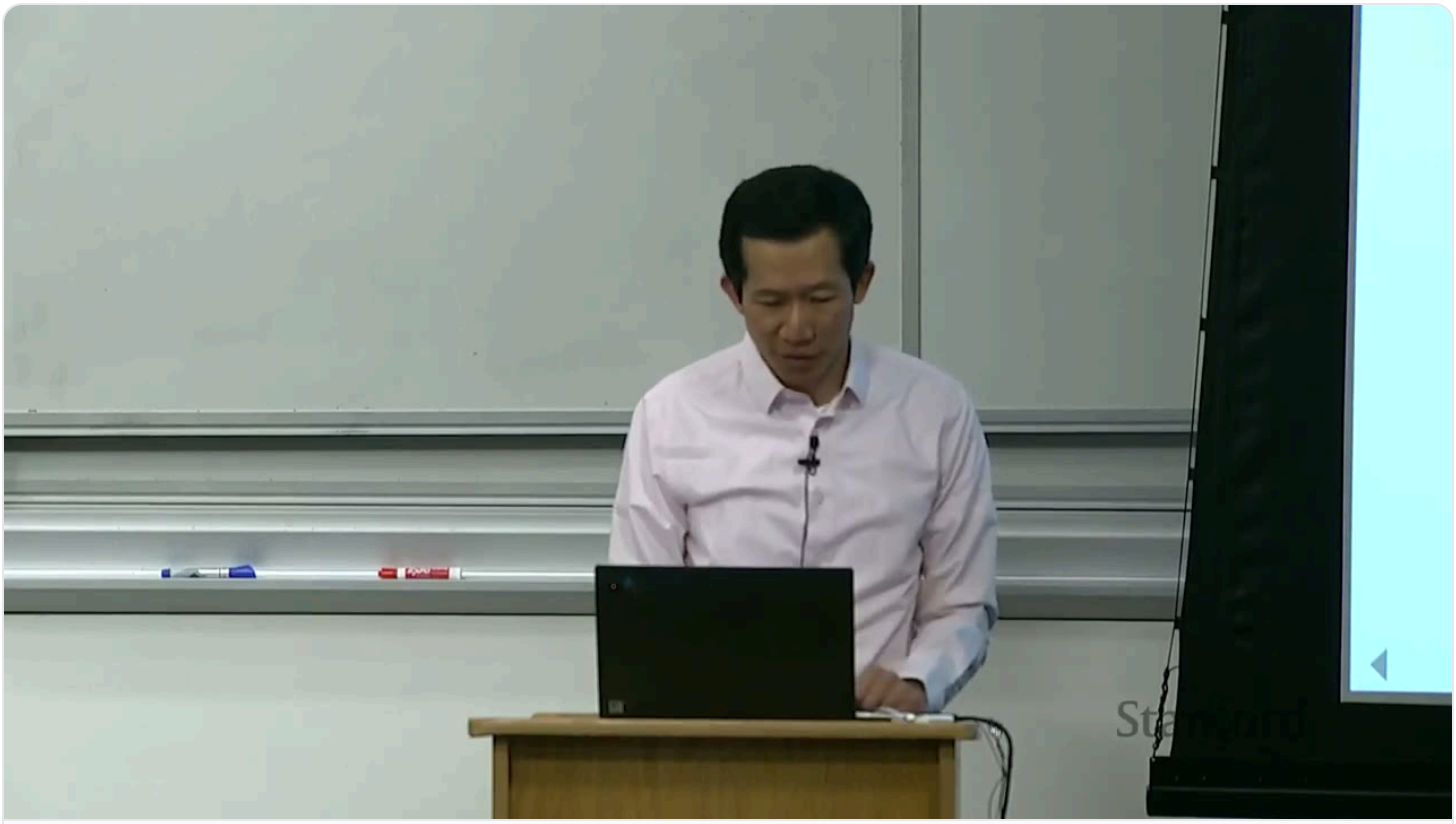
The overarching theme is to orchestrate computation to avoid data transfer bottlenecks. Data transfer is generally much slower than computation, making it the primary bottleneck.

The hierarchy of memory access speeds, from fastest to slowest: - **Single node, single GPU**: L1 cache / shared memory (extremely fast, very small) - **Single node, single GPU**: HBM (fast, larger) - **Single node, multi-GPU**: NVLink (slower than HBM, but faster than inter-node communication) - **Multi-node, multi-GPU**: NVSwitch (slowest, largest scale)

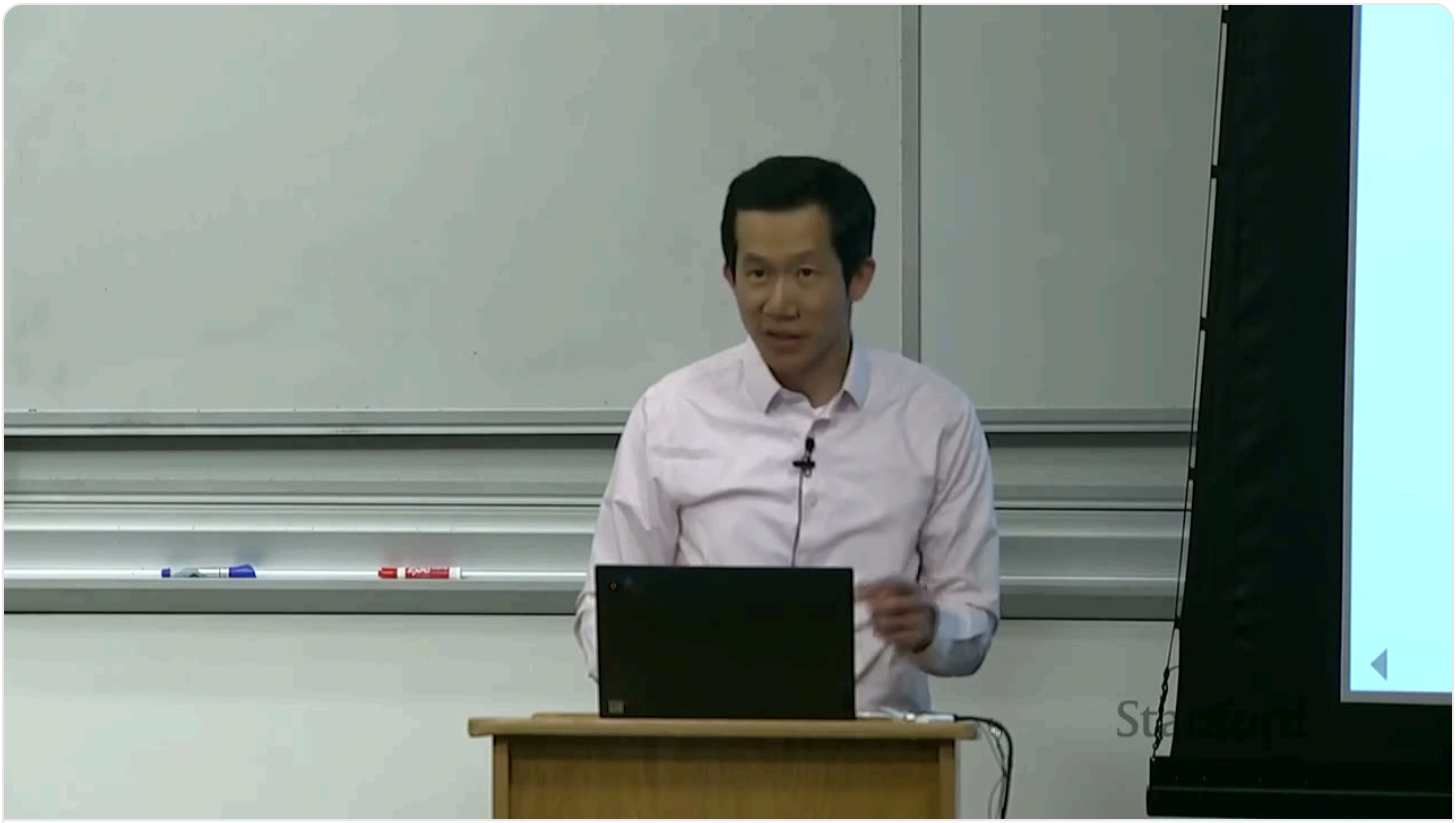
This lecture will concretize these concepts in code.

[2:30] Part 1: Building Blocks of Distributed Communication/Computation

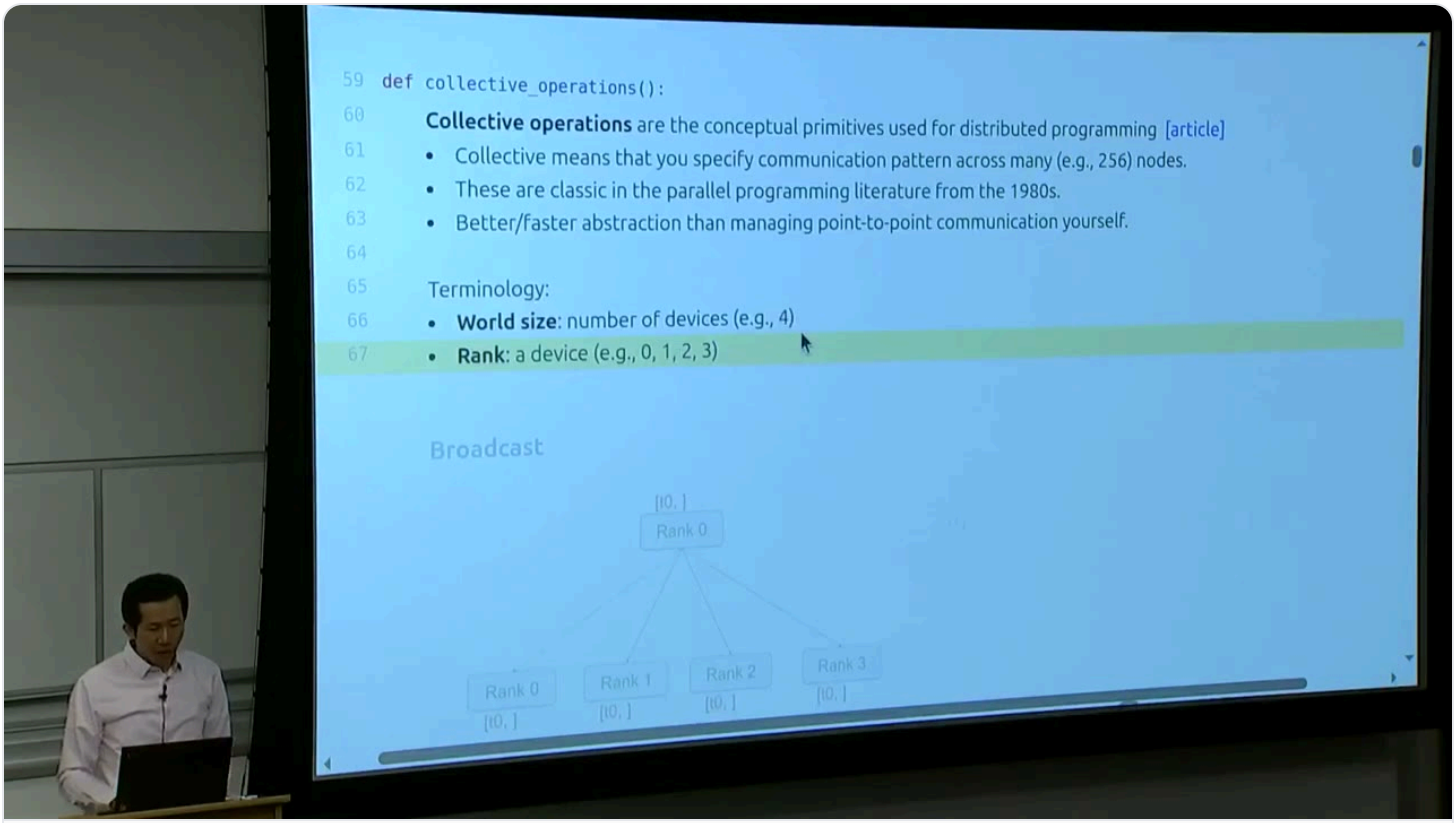
[2:39] Collective Operations



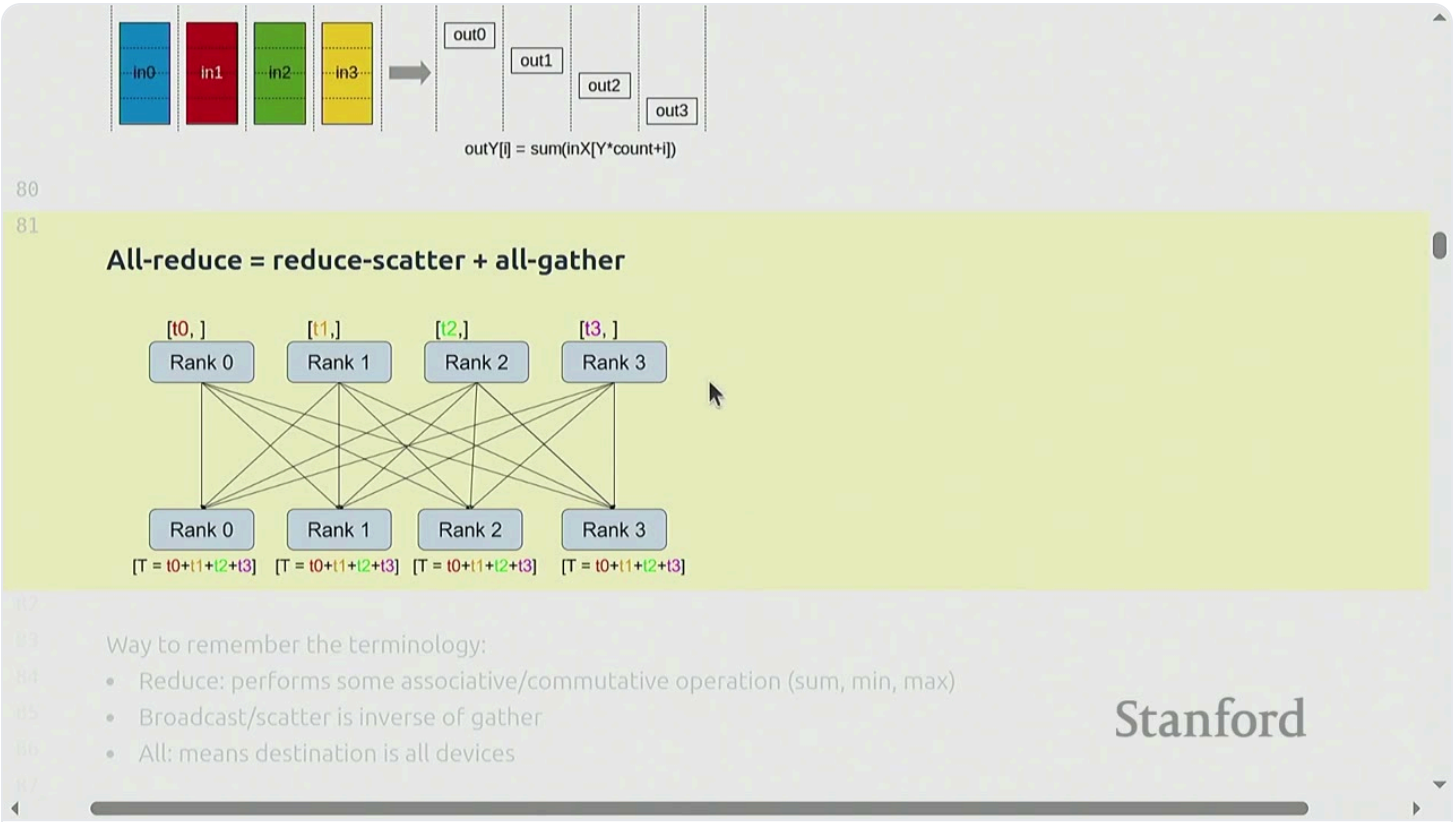
3:51 Professor looking down at his laptop on the podium.



4:00 Professor looking up from his laptop, speaking to the audience.



5:30 Slide defining collective operations, world size, and rank in distributed programming.



7:18 Diagram illustrating the "all-reduce" operation as reduce-scatter plus all-gather.

Collective operations are conceptual primitives used for distributed programming. "Collective" means they specify a communication pattern across many (e.g., 256) nodes. These primitives are classic in parallel programming literature from the 1980s and provide a better/faster abstraction than managing point-to-point communication yourself. They are tried and true methods for distributed communication.

Terminology: - **World size**: The total number of devices (e.g., 4). - **Rank**: A device's unique identifier (e.g., 0, 1, 2, 3 for a world size of 4).

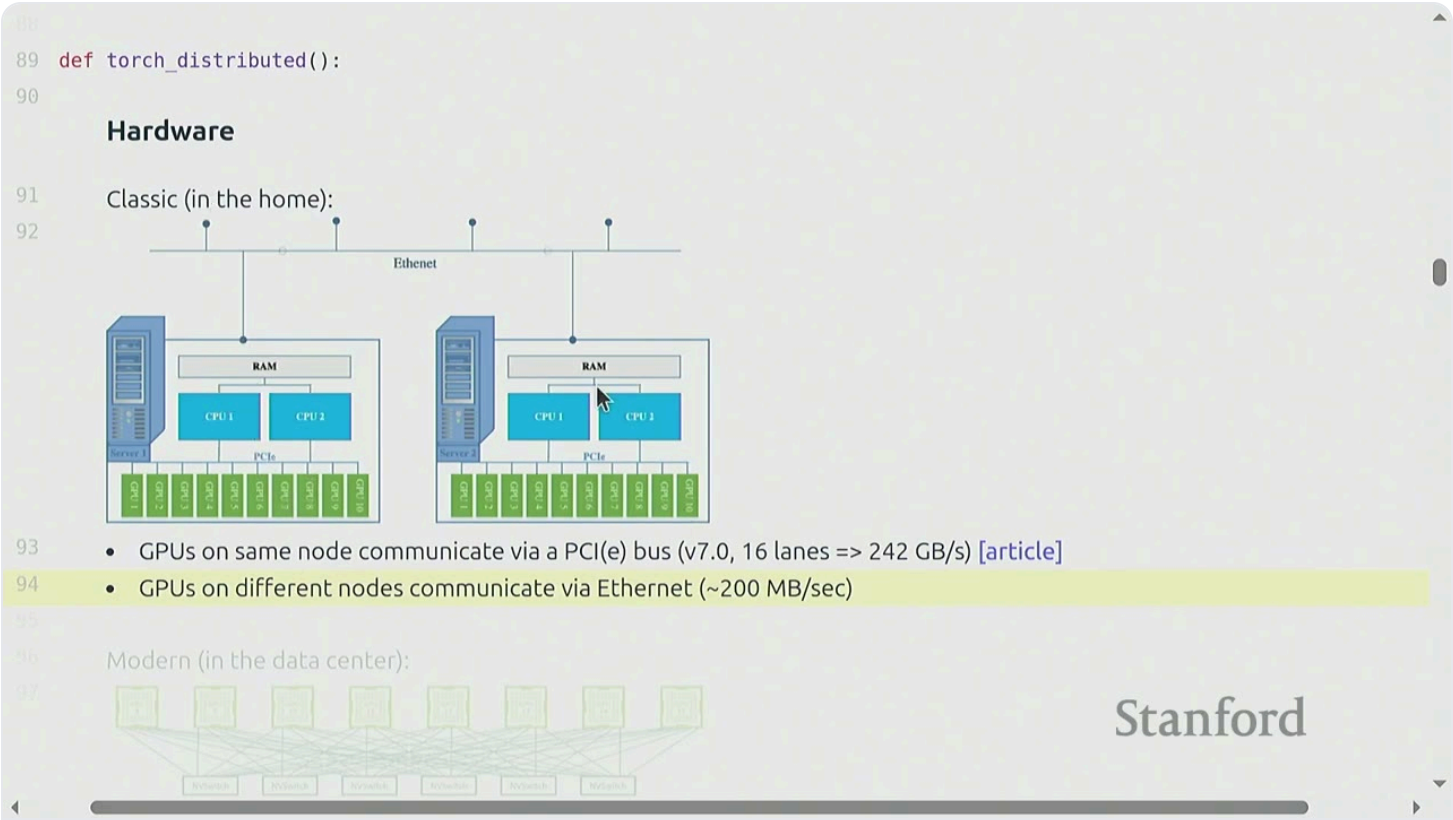
Types of Collective Operations: - **Broadcast**: Data from one rank (e.g., Rank 0) is sent to all other ranks. - Initial state: Rank 0 has `[t0]`, others have `[]`. - Final state: All ranks have `[t0]`. - **Scatter**: Data from one rank (e.g., Rank 0 has `[t0, t1, t2, t3]`) is distributed such that each rank receives a different part (e.g., Rank 0 gets `[t0]`, Rank 1 gets `[t1]`, etc.). - Initial state: Rank 0 has `[t0, t1, t2, t3]`, others have `[]`. - Final state: Rank 0 has `[t0]`, Rank 1 has `[t1]`, Rank 2 has `[t2]`, Rank 3 has `[t3]`. - **Gather**: The inverse of scatter. Data from all ranks (e.g., Rank 0 has `[t0]`, Rank 1 has `[t1]`, etc.) is collected onto a single rank (e.g., Rank 0). - Initial state: Rank 0 has `[t0]`, Rank 1 has `[t1]`, etc. - Final state: Rank 0 has `[t0, t1, t2, t3]`, others have `[]`. - **Reduce**: Similar to gather, but instead of concatenating, an associative/commutative operation (e.g., sum, min, max, average) is performed on the collected

data. - Initial state: Rank 0 has `[t0]`, Rank 1 has `[t1]`, etc. - Final state: Rank 0 has `[T = t0 + t1 + t2 + t3]`, others have `[]`. - **All-gather**: Similar to gather, but the collected data is distributed to *all* ranks. - Initial state: Rank 0 has `[t0]`, Rank 1 has `[t1]`, etc. - Final state: All ranks have `[t0, t1, t2, t3]`. - **Reduce-scatter**: A combination of reduce and scatter. Each rank contributes a portion of its data to a global reduction, and the result is scattered back such that each rank receives a portion of the reduced data. - Initial state: Each rank `i` has `in_i`. - Final state: Each rank `i` receives `out_i`, where `out_i` is a segment of the global reduction `SUM(in_0, in_1, ..., in_N)`. - All-reduce is equivalent to reduce-scatter + all-gather.

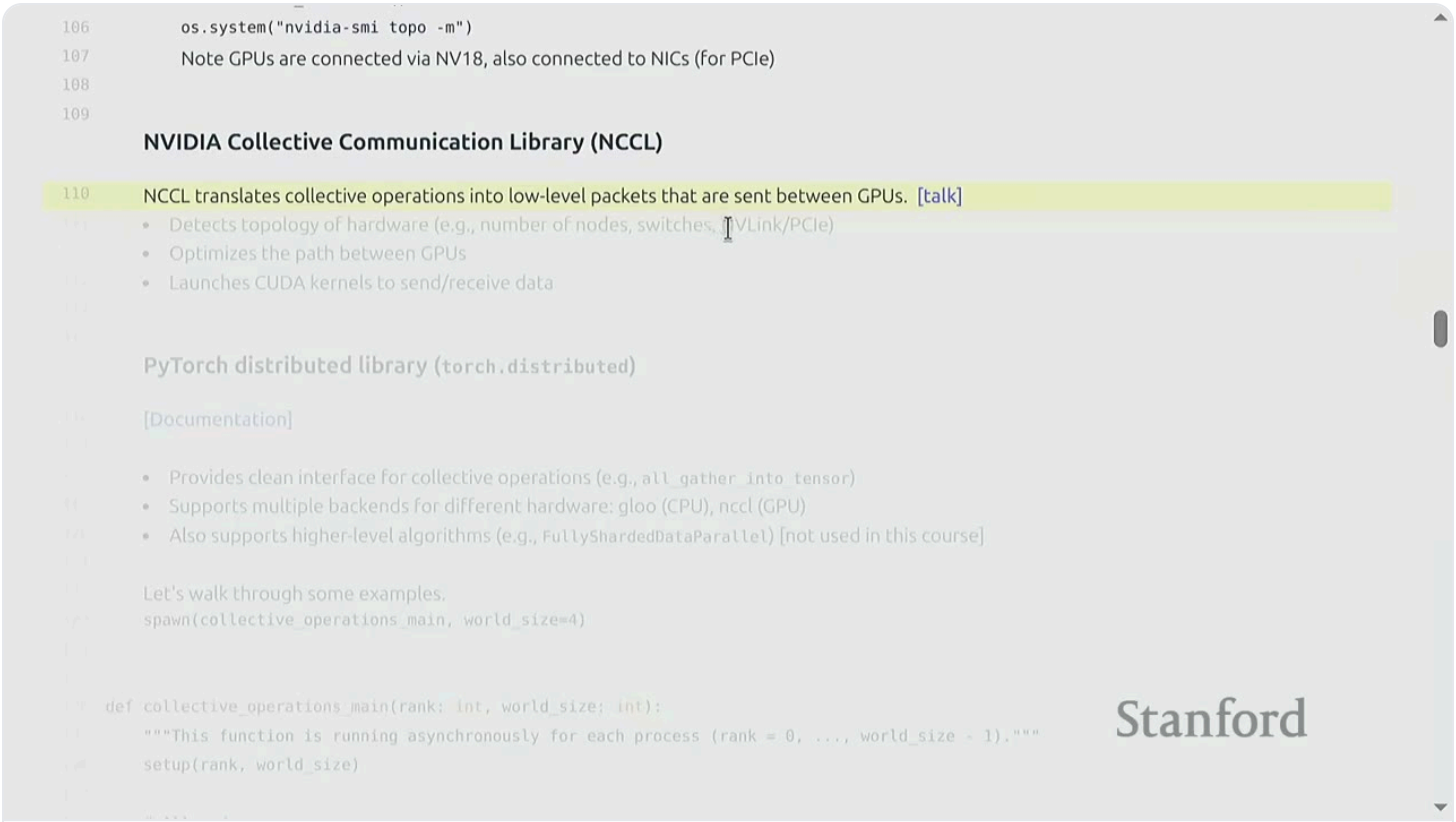
How to remember the terminology: - **Reduce**: Performs some associative/commutative operation (sum, min, max). - **Broadcast/scatter**: Is inverse of gather. - **All**: Means destination is all devices.

[8:15] Hardware and Software for Distributed Communication

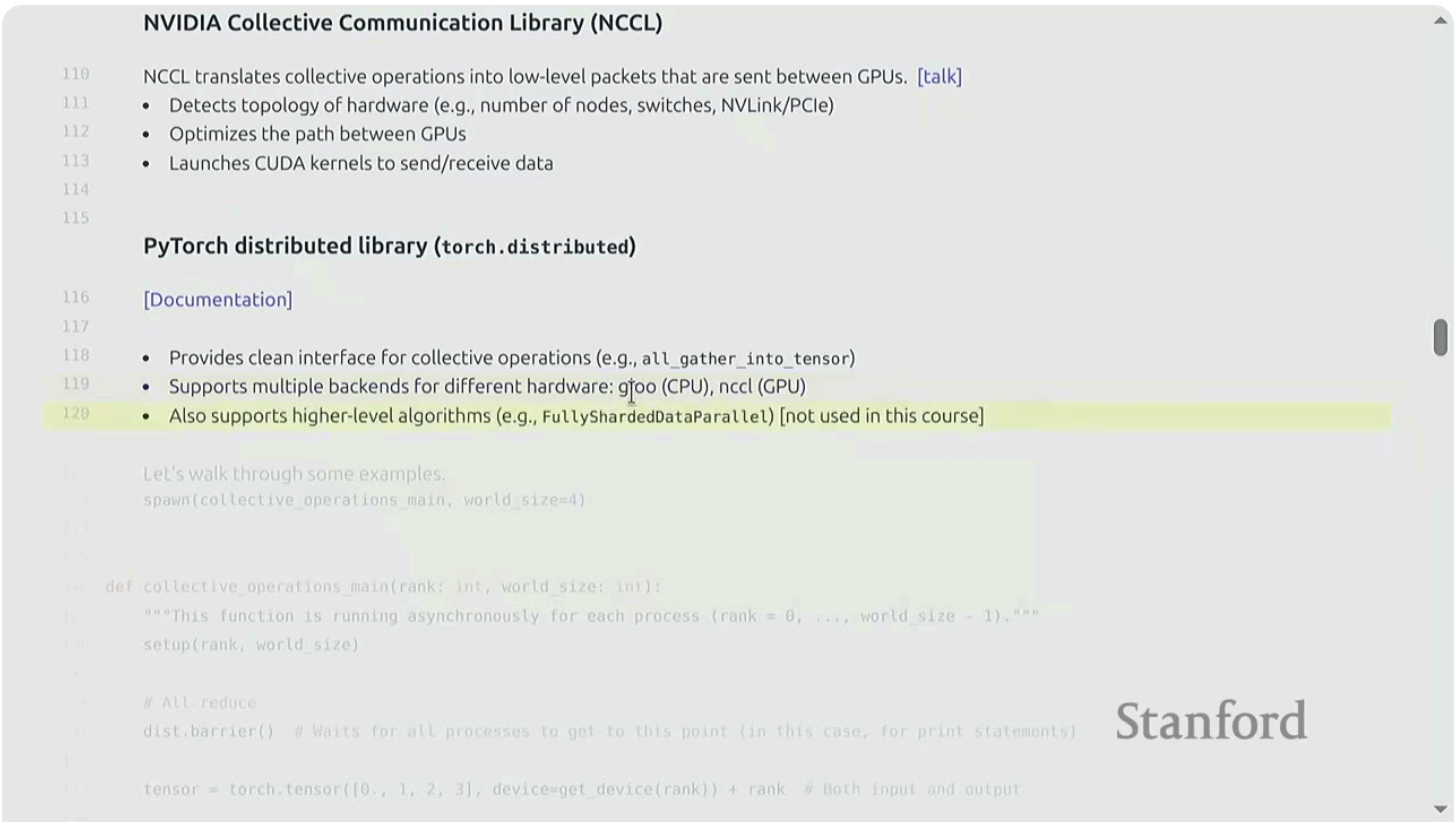
[8:20] Hardware Setup



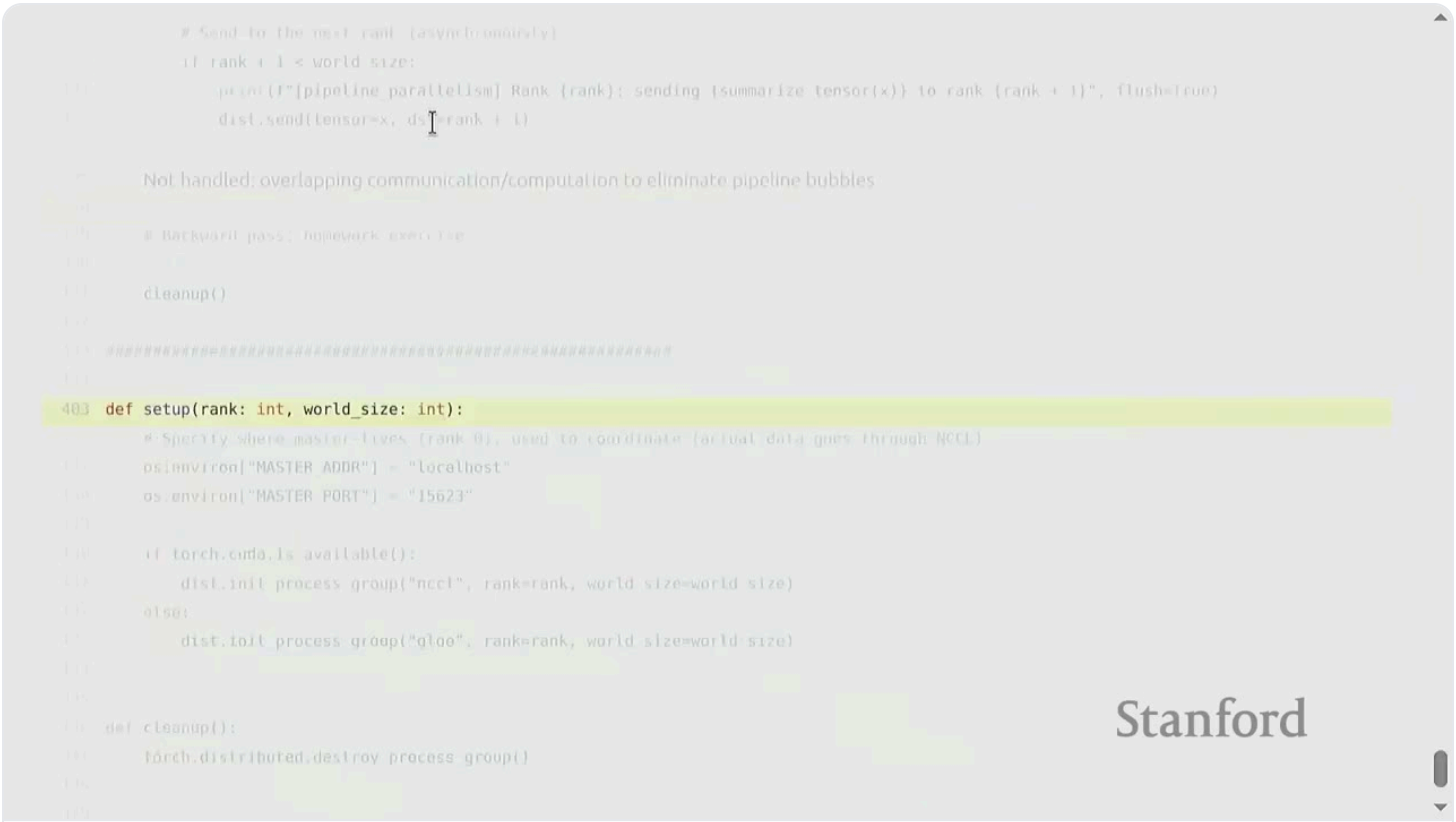
8:55 Diagram comparing classic (Ethernet) and modern (data center) hardware communication.



13:46 Slide explaining NVIDIA Collective Communication Library (NCCL) and its functions.



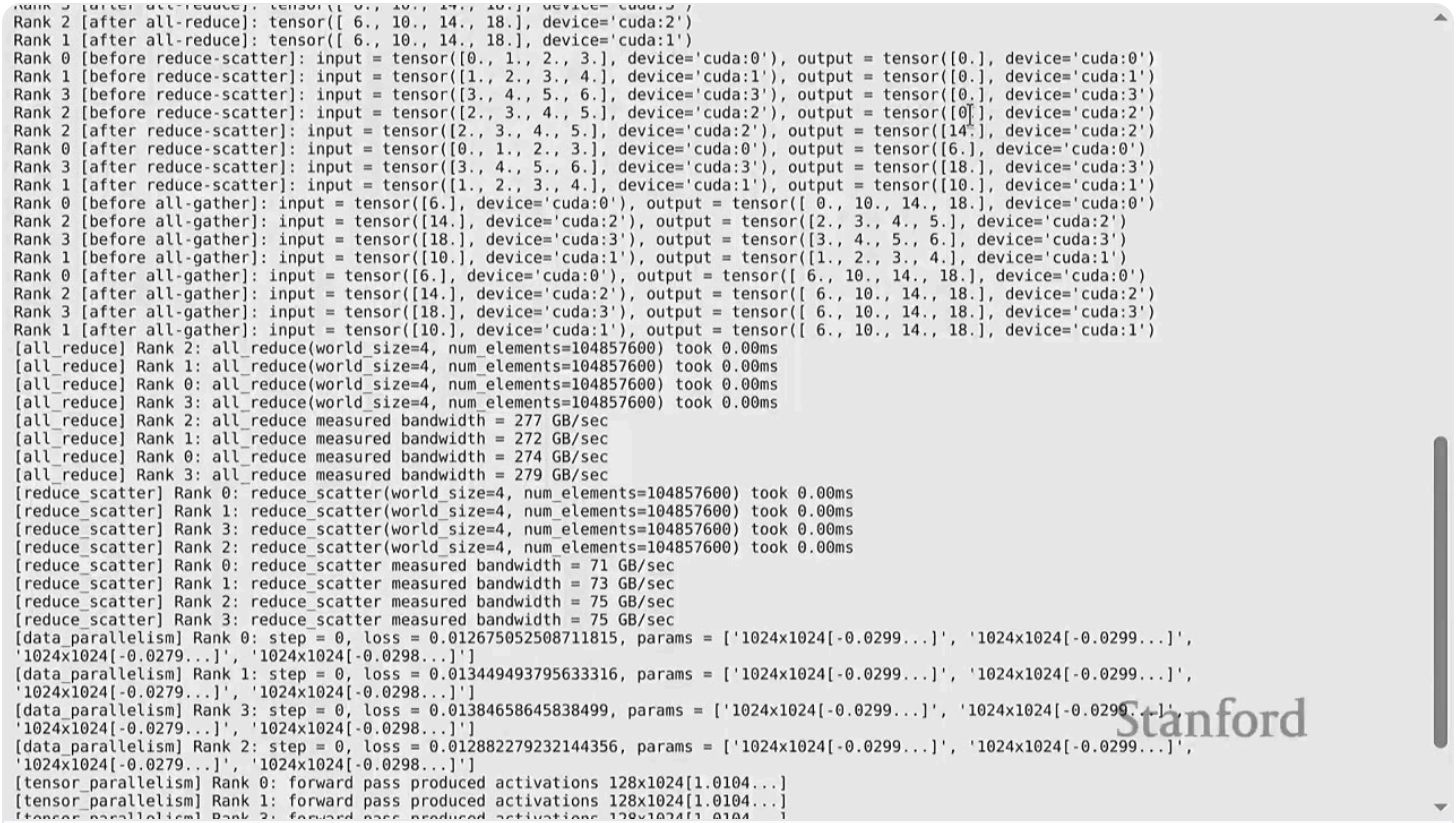
15:33 Slide detailing PyTorch distributed library (torch.distributed) capabilities.



17:08 Code snippet defining the `setup` function for distributed training.



18:44 Terminal output showing NIC legend and results of all-reduce and reduce-scatter operations.



20:30 Terminal output continuing with all-gather, all-reduce bandwidth, and data parallelism logs.

- **Classic (in the home)**: GPUs on the same node communicate via a PCIe bus (v7.0, 16 lanes => 242 GB/s). GPUs on different nodes communicate via Ethernet (~200 MB/s). This setup is suboptimal for deep learning due to high overhead (data copying, kernel involvement, slow transport).
- **Modern (in the data center)**:
  - **Within a node**: NVLink connects GPUs directly, bypassing the CPU.
  - **Across nodes**: NVSwitch connects GPUs directly, bypassing Ethernet.
  - Example: Each H100 GPU has 18 NVLink 4.0 links, for a total of 900 GB/s. This is significantly faster than PCIe or Ethernet.
  - For comparison, memory bandwidth for HBM is 3.9 TB/s (still faster than NVLink).

To check your hardware setup, you can use `os.system("nvidia-smi topo -m")`. This command shows how GPUs are connected via NVLink and NICs (for PCIe).

[1:10:50] NVIDIA Collective Communication Library (NCCL)

NCCL is a low-level library provided by NVIDIA that handles collective operations. It: - Detects the topology of hardware (e.g., number of nodes, switches, NVLink/PCIe). - Optimizes the path between GPUs for efficient data transfer. - Launches CUDA kernels to send/receive data.

NCCL abstracts away the complexities of low-level communication, allowing programmers to use high-level collective operations without worrying about hardware specifics.

[1:13:58] PyTorch Distributed Library (`torch.distributed`)

`torch.distributed` provides a clean Pythonic interface for collective operations (e.g., `all_gather_into_tensor`). It supports multiple backends for different hardware (e.g., Gloo for CPU, NCCL for GPU). This allows for greater portability, as the same code can run on different hardware configurations, though performance will vary. It also supports higher-level algorithms (e.g., FullyShardedDataParallel - FSDP), but this course focuses on implementing from scratch.

[1:16:00] Collective Operations Examples in PyTorch

Let's walk through some examples of `torch.distributed` collective operations. The examples use a utility function `spawn(collective_operations_main, world_size=4)` which runs the `collective_operations_main` function asynchronously on `world_size` number of processes, each with a unique rank.

```
def collective_operations_main(rank: int, world_size: int):
    """This function is running asynchronously for each process (rank = 0, ..., world_size - 1)
    setup(rank, world_size)

    # All-reduce
    dist.barrier()  # Wait for all processes to get to this point (in this case, for print state

    tensor = torch.tensor([0., 1, 2, 3], device=get_device(rank)) + rank  # Both input and outpu
    print(f"Rank {rank} [before all-reduce]: {tensor}", flush=True)

    dist.all_reduce(tensor=tensor, op=dist.ReduceOp.SUM, async_op=False)  # Modifies tensor in p
    print(f"Rank {rank} [after all-reduce]: {tensor}", flush=True)

    # Reduce-scatter
    dist.barrier()

    input = torch.arange(world_size, dtype=torch.float32, device=get_device(rank)) + rank  # Inp
    output = torch.empty(1, device=get_device(rank))  # Allocate output

    print(f"Rank {rank} [before reduce-scatter]: input={input}, output={output}", flush=True)
    dist.reduce_scatter_tensor(output=output, input=input, op=dist.ReduceOp.SUM, async_op=False)
    print(f"Rank {rank} [after reduce-scatter]: input={input}, output={output}", flush=True)

    # All-gather
    dist.barrier()

    input = output  # Input is the output of reduce-scatter
    output = torch.empty(world_size, device=get_device(rank))  # Allocate output
```

```
    print(f"Rank {rank} [before all-gather]: input={input}, output={output}", flush=True)
    dist.all_gather_into_tensor(tensor_list=activations, tensor=x, async_op=False)
    print(f"Rank {rank} [after all-gather]: input={input}, output={output}", flush=True)

    # Indeed, all-reduce = reduce-scatter + all-gather!
    cleanup()
```

Setup (`setup(rank, world_size)`): - Each process needs to initialize itself and find other processes. - They connect to a single host (e.g., `localhost:15623`) for coordination. This is not where data flows, but where processes discover each other. - `dist.init_process_group("nccl", rank=rank, world_size=world_size)` initializes the process group using NCCL (or Gloo for CPU).

`dist.barrier()` : - A synchronization primitive that waits for all processes in the process group to reach that point. Useful for ensuring print statements are grouped or for specific synchronization needs.

All-reduce Example: 1. Each rank creates a tensor: `tensor = torch.tensor([0., 1, 2, 3], device=get_device(rank)) + rank` . - Rank 0: `[0., 1., 2., 3.]` - Rank 1: `[1., 2., 3., 4.]` - Rank 2: `[2., 3., 4., 5.]` - Rank 3: `[3., 4., 5., 6.]` 2. `dist.all_reduce(tensor=tensor, op=dist.ReduceOp.SUM, async_op=False)` performs an in-place sum reduction across all ranks and distributes the result to all ranks. - After all-reduce, all ranks will have `[6., 10., 14., 18.]` (sum of corresponding elements across ranks).

Reduce-scatter Example: 1. Each rank creates an input tensor: `input = torch.arange(world_size, dtype=torch.float32, device=get_device(rank)) + rank` . 2. Each rank allocates an empty output tensor: `output = torch.empty(1, device=get_device(rank))` . 3. `dist.reduce_scatter_tensor(output=output, input=input, op=dist.ReduceOp.SUM, async_op=False)` performs a sum reduction and scatters the result. - Rank 0's output will be the sum of the 0th elements from all inputs: `0+1+2+3 = 6` . - Rank 1's output will be the sum of the 1st elements from all inputs: `1+2+3+4 = 10` . - And so on. - This operation produces the same reduced values as all-reduce, but each rank only receives its designated portion.

All-gather Example: 1. The input for all-gather is the output from reduce-scatter. 2. Each rank allocates an empty output tensor of `world_size` elements. 3. `dist.all_gather_into_tensor(tensor_list=activations, tensor=x, async_op=False)`

collects all inputs and distributes the full collected tensor to all ranks. - After all-gather, all ranks will have `[6., 10., 14., 18.]`.

This demonstrates that `all_reduce` is indeed equivalent to `reduce_scatter` followed by `all_gather`.

[3:55] Benchmarking Collective Operations

Benchmarking helps measure actual NCCL bandwidth. - **Warmup**: Run the operation once to ensure CUDA kernels are loaded and any lazy computations are performed. - **Synchronization**: Use `torch.cuda.synchronize()` and `dist.barrier()` to ensure all processes and CUDA kernels are finished before starting the timer. - **Timing**: Use `time.time()` to measure the duration.

Measuring Bandwidth for All-reduce: - **Sent bytes**: `tensor.element_size() * tensor.numel() * 2` - The `* 2` factor accounts for both sending the input and receiving the output. Each rank sends its data, and then receives the aggregated data. - **Total duration**: `world_size * duration` (assuming all ranks contribute equally). - **Bandwidth**: `sent_bytes / total_duration`.

For an all-reduce of 100 million elements on 4 GPUs, the measured bandwidth was ~277 GB/s. This is lower than the theoretical 900 GB/s for H100s, highlighting that real-world performance depends on tensor size, number of devices, and other factors. Benchmarking is crucial to understand actual performance.

Measuring Bandwidth for Reduce-scatter: - **Sent bytes**: `tensor.element_size() * tensor.numel() * 1` - The `* 1` factor is used because in reduce-scatter, each rank sends its input, but only receives a *portion* of the reduced output. Effectively, it's like a reduction where data flows in one direction to be aggregated. - For a reduce-scatter of 100 million elements on 4 GPUs, the measured bandwidth was ~75 GB/s.

The difference in bandwidth between all-reduce and reduce-scatter (277 GB/s vs. 75 GB/s) can be attributed to several factors, including potential hardware accelerations in NVIDIA's network for all-reduce operations.

[2:30] Part 2: Distributed Training Strategies

This section walks through bare-bones implementations of distributed training strategies on deep MLPs. MLPs are used as a representative architecture because compute bottlenecks in Transformers often resemble those in MLPs.

[3:06] Data Parallelism (DDP)

In DDP, the data is sharded along the batch dimension. Each rank gets a slice of the data. - **Losses**: Different across ranks (computed on local data). - **Gradients**: All-reduced to be the same across ranks. - **Parameters**: Therefore, parameters remain the same across ranks.

DDP Implementation Details: 1. **Generate Sample Data**: `batch_size = 128`, `num_dim = 1024`. 2. **Slice Data**: Each rank calculates its `local_batch_size` (`batch_size / world_size`) and determines its `start_index` and `end_index` to access its specific slice of the data. 3. **Create MLP Parameters**: Each layer's parameters are `num_dim` by `num_dim` matrices. 4. **Optimizer**: `torch.optim.AdamW` is used. 5. **Training Loop**: - **Forward Pass**: Data is passed through the MLP layers (`x = x @ param`, `x = F.gelu(x)`). - **Compute Loss**: `loss = x.square().mean()`. - **Backward Pass**: `loss.backward()`. - **Synchronize Gradients**: This is the key step for DDP. For each parameter, `dist.all_reduce(tensor=param.grad, op=dist.ReduceOp.AVG, async_op=False)` is called to average the gradients across all ranks. This ensures all ranks have the same gradients. - **Update Parameters**: `optimizer.step()`.

DDP effectively runs `world_size` number of SGD runs, but because gradients are synchronized, all models converge to the same state. The `all_reduce` operation acts as a synchronization point, ensuring all ranks are at the same step before proceeding.

[4:40] Tensor Parallelism

In tensor parallelism, the model is sharded along the hidden dimension. The data remains the same across all ranks. - Each rank gets a part of each layer. - All data/activations need to be transferred between ranks.

Tensor Parallelism Implementation Details (Forward Pass only): 1. **Generate Sample Data**: `batch_size = 128`, `num_dim = 1024`. 2. **Shard `num_dim`**: `local_num_dim = num_dim / world_size = 256`. 3. **Create MLP Parameters**: Each layer's parameter matrix is now `num_dim` by `local_num_dim`. 4. **Forward Pass Loop (for each layer):** - **Compute Activations**: `x = x @ param`, `x = F.gelu(x)`. Note that `x` is `batch_size` by `local_num_dim`. - **Allocate Memory for Activations**: `activations = torch.empty(batch_size, local_num_dim,`

`device=get_device(rank))`. - **Send Activations via All-gather**: `dist.all_gather_into_tensor(tensor_list=activations, tensor=x, async_op=False)`. This collects all the sharded activations from each rank and distributes the full set of activations to all ranks. - **Concatenate Activations**: `x = torch.cat(activations, dim=1)`. This reconstructs the full activation tensor (`batch_size` by `num_dim`).

This process is repeated for each layer. As seen, tensor parallelism involves significant communication (all-gather for activations) at each layer, requiring high-bandwidth interconnects.

[6:25] Pipeline Parallelism

In pipeline parallelism, the model is sharded along layers. All ranks get all the data. - Each rank gets a subset of the layers. - All data/activations need to be transferred between ranks.

Pipeline Parallelism Implementation Details (Forward Pass only): 1. **Generate Sample Data**: `batch_size = 128` , `num_dim = 1024` . 2. **Shard Layers**: `local_num_layers = num_layers / world_size = 2` . Each rank gets two layers. 3. **Create MLP Parameters**: Parameters are allocated only for the layers assigned to each rank. 4. **Forward Pass Loop (for each microbatch):** - **Break into Microbatches**: To minimize pipeline bubbles, the batch is divided into microbatches (e.g., `micro_batch_size = 32`). - **Process Microbatches**: - **Receive Activations**: If `rank > 0` , `dist.recv(tensor=x, src=rank-1)` receives activations from the previous rank. - **Compute Layers**: Apply the assigned layers to the received activations. - **Send Activations**: If `rank < world_size - 1` , `dist.send(tensor=x, dst=rank+1)` sends activations to the next rank.

Missing from this basic implementation: - **Overlapping Communication/Computation**: The current implementation uses synchronous `send` and `recv` . Asynchronous operations (`isend` , `irecv`) would allow computation to overlap with communication, reducing pipeline bubbles. - **Backward Pass**: Implementing the backward pass requires careful scheduling to interleave forward and backward steps across the pipeline.

[6:55] What's Missing?

- **More General Models**: The examples used simple MLPs. Real-world models like Transformers have more complex architectures (e.g., attention mechanisms) that require more sophisticated sharding strategies.

- **More Communication/Computation Overlap**: The current implementations are basic and don't fully exploit asynchronous operations to overlap communication with computation, which is crucial for performance.
- **More Complex Code with More Bookkeeping**: Implementing these strategies for arbitrary models requires significant bookkeeping to manage parameters, optimizer states, and activations across devices.

[7:50] Jax/TPUs: A Different Approach

Jax/TPUs offer a different paradigm. You define the model and the sharding strategy, and the Jax compiler handles the rest. This provides a higher-level abstraction compared to PyTorch, where you build up from primitives.

Example: FSDP in Jax (Levanter project) - You define your model and optimizer states. - You specify which axes to shard for parameters (e.g., `embed` for embedding, `data` for batch). - The Jax compiler automatically handles the sharding and communication. - This allows for FSDP in ~10 lines of code.

Example: Tensor Parallelism in Jax - You specify which axes to shard for tensor parallelism (e.g., `head` for attention heads, `model` for MLP layers). - The Jax compiler handles the rest.

This approach offers conceptual simplicity, as the compiler automatically translates high-level sharding specifications into low-level primitives. However, this course focuses on PyTorch to demonstrate how to build up from primitives.

[8:25] Summary

- **Many ways to parallelize**: Data (batch), tensor (expert/width), pipeline (depth), sequence (length).
- **Re-compute or store**: You can re-compute activations from scratch to save memory or store them in memory/another GPU's memory and communicate them.
- **Hardware is getting faster**: But bigger models will always push the limits of hardware capabilities, necessitating these hierarchical distributed training techniques.

Collective operations are the conceptual primitives used for distributed programming. They are classic in parallel programming literature from the 1980s and provide a better/faster abstraction than managing point-to-point communication yourself.

[8:40] Practical Takeaways

- Understanding the memory and communication hierarchy is key to optimizing distributed deep learning.
- Collective operations are fundamental building blocks, and `torch.distributed` provides a powerful interface for using them.
- Benchmarking is essential to measure actual performance and identify bottlenecks.
- Different parallelism strategies (data, tensor, pipeline) involve different ways of sharding models and data, each with its own communication patterns and trade-offs.
- Overlapping communication and computation is crucial for maximizing efficiency in distributed training.
- While higher-level frameworks (like Jax) abstract much of the complexity, understanding the underlying primitives helps in debugging and optimizing custom architectures.
- Hardware will continue to improve, but models will always grow larger, making efficient distributed training a continuous challenge.

[8:45] Open Questions / Things to Remember

- How to handle non-determinism issues on GPUs for exact reproducibility.
- How to implement asynchronous sends/receives (`isend` , `irecv`) to overlap communication and computation.
- How to manage complex bookkeeping for arbitrary model architectures in distributed settings.
- The trade-offs between recomputation and storing activations in memory/communicating them.
- The role of specialized hardware (like Grock, Cerebras) with large on-chip memory for reducing data movement.
- The fundamental difference between CPU-era control-flow focused programming and data-flow focused deep learning workloads.
- The concept of a static computation graph in deep learning and how it enables smarter optimization and layout of computations.

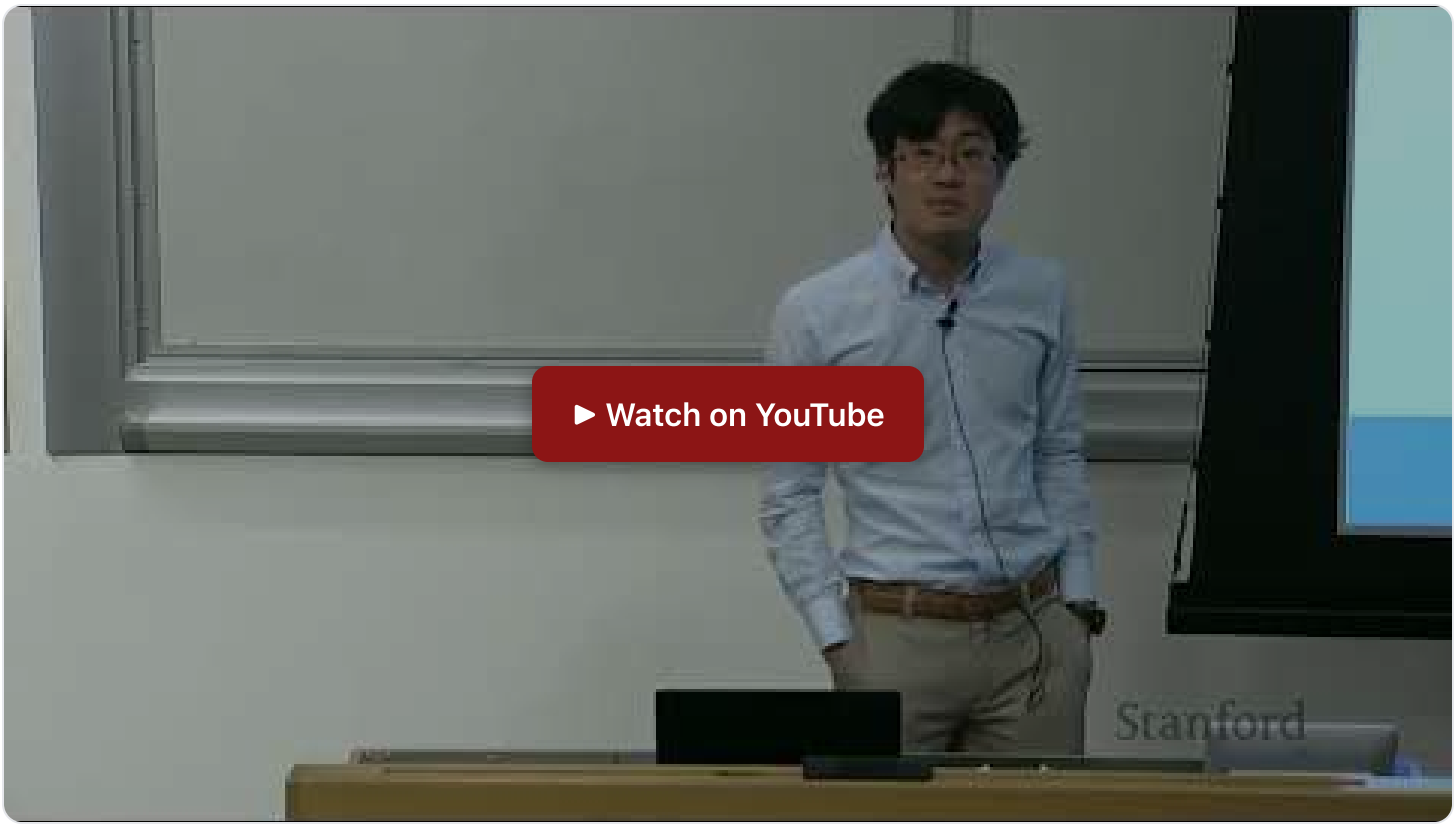
← Lec 07

Lec 09 →

Lecture 09: Scaling Laws 1

Stanford CS336 · Spring 2025 · 1:05:18 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



TL;DR * Scaling laws describe how model performance (loss) changes with resources (data, model size, compute) in a predictable, often log-linear, fashion. * Historically, scaling laws were studied in statistical learning theory (generalization bounds) and early NLP (data scaling for task performance). * Modern scaling laws for LLMs show power-law relationships between test loss and compute, dataset size, or parameters. * These laws enable efficient engineering decisions: predicting optimal hyperparameters, architectures, and resource allocation (e.g., data vs. model size) without expensive large-scale training. * Key insights include the existence of distinct scaling regions (small data, power-law, irreducible error) and the importance of accounting for factors like data composition, repetition, and specific parameter types (e.g., embeddings). * While powerful for pre-training objectives (like perplexity), scaling law predictability can be less reliable for downstream tasks.

Key Concepts * Scaling Laws * Data Scaling * Model Scaling * Compute Scaling * Power-Law Relationships * Log-Log Plots * Generalization Error * Irreducible Error * Small Data Region * Power-Law Region * Data Composition * Data Repetition * Isotropes / IsoFLOPs * Optimal Training Budget * Critical Batch Size * Scale-Aware Initialization (muP) * Downstream Scaling

---

[0:00] Introduction to Scaling Laws

The lecture begins by setting a scenario: imagine you have access to 100,000 H100 GPUs for a month and need to build the best open-source Large Language Model (LLM). We've already covered infrastructure, distributed training, pre-training datasets, and architectures. The question then becomes: how do you make optimal decisions regarding architecture, hyperparameters, and resource allocation to push the frontiers of model performance, rather than just copying existing models?

This is where **scaling laws** come in. * **Goal**: Build simple, predictive "laws" for the behavior of language models. * **Approach**: Train small models, learn from their behavior, and extrapolate to larger ones. * **Old (unpleasant) way**: Tune hyperparameters directly on big models, which is computationally expensive. * **New (optimistic) way**: Tune on small models, extrapolate to large ones, saving significant compute.

The lecture will cover: 1. **History and background of scaling laws**: Contextualizing their origins beyond recent "AGI" hype. 2. **Neural (LLM) scaling behaviors**: Empirical results and practical applications.

[0:05] Part 1. Scaling Laws: History and Background

The speaker emphasizes that scaling laws are more grounded than often portrayed, with a rich history.

[0:05] Data Scaling as Empirical Sample Complexities

From a statistical machine learning perspective, scaling laws describe how model behavior changes with increased data or model size.

- **Theoretical Generalization Bounds**:
    - For learning in a finite set of k hypotheses, the generalization error $\epsilon(\hat{h})$ is bounded by:

$$\epsilon(\hat{h}) \leq (\min_{h \in H} \epsilon(h)) + 2\sqrt{\frac{1}{m} \log \frac{2k}{\delta}}$$

This suggests error should scale as $1/\sqrt{m}$ (where m is the number of samples).

- For generative modeling (e.g., fitting a smooth density), the L_2 error can be bounded by a polynomial in n (number of samples):

$$\sup_{p \in \mathcal{P}_{\beta,L}} E[(\hat{p}_n(x_0) - p(x_0))^2] \leq Cn^{-\frac{\beta}{2\beta+1}}$$

These are theoretical upper bounds, not actual realized loss values.

- **The Leap to Empirical Scaling Laws**: Moving from theoretical bounds to empirically observed performance trends.

[0:05] Earliest (Data) Scaling Law Paper - 1993

The earliest paper that resembles modern scaling law analysis is "Learning Curves: Asymptotic Values and Rate of Convergence" from 1993 by Corinna Cortes, L.D. Jackel, Sara A. Solla, Vladimir Vapnik, and John S. Denker (Bell Labs).

- **Key Idea**: Training classifiers on large datasets is computationally demanding. We need predictive methods to estimate model performance without full training.
- **Functional Form**: The paper proposes that test error (E) can be expressed as:

$$E = A + \frac{B}{\text{train-data-size}^{\beta}}$$

where A is irreducible error, and $B/\text{train-data-size}^{\beta}$ is a polynomially decaying term.
- **Methodology**: They trained small models, fit curves, and accurately predicted performance for larger models. This is precisely the spirit of modern scaling laws.

[0:05] Early History of Scaling Laws - Data Scaling

Other early works also explored data scaling. * **Log-linear scaling with data (Banko and Brill, 2001)**: Studied how NLP system performance scales with data size. * **Observation**: Log-linear relationship between data size (x-axis, log scale) and test accuracy (y-axis). * **Conclusion**: Dramatic performance improvements can be achieved by scaling data. This led to the argument that collecting more data might be more effective than complex algorithm development, a sentiment echoed in modern pre-training.

[0:05] Early Tests of Functional Forms

Even in the early 2010s, researchers were investigating the best functional forms for these scaling relationships. * **Kolachina et al. (2012)**: Tested various functional forms (e.g., exponential, power-law) to predict model behavior based on training data size. * **Key Relationship**: Model capabilities (y-axis) as a function of data (x-axis).

[0:05] Earliest Large-Scale Neural Scaling Work: Hestness et al. 2017

This paper from Baidu (Hestness et al., 2017) is often cited as the earliest large-scale neural scaling work. * **Observation**: For tasks like machine translation, speech, and vision, error rates fall as a power law with increasing data. * **Generalization Error Curve**: The paper proposes a conceptual curve for generalization error vs. training data size, identifying three regions: 1. **Small Data Region**: Initial phase where performance is close to random guessing ("Best Guess Error"). 2. **Power-Law Region**: Predictable scaling where error decreases polynomially with data ("Power-Law"). 3. **Irreducible Error Region**: Asymptotic phase where error approaches the fundamental limit of the task ("Irreducible Error").

[0:05] Hestness II: Ahead of its Time

The Hestness et al. (2017) paper also foreshadowed many "new" phenomena discussed in recent years: * **"Emergence"**: Noted that small data regimes can be hard to predict because models might suddenly "leave the random region" and show significant improvements. * **Scaling by Compute**: Highlighted that if scaling laws are predictable, then scaling by compute (training larger models for longer) becomes a crucial strategy. * **Speed = Accuracy Trade-off**: Suggested that hardware improvements (e.g., quantization) and distributed training could be used to improve compute throughput, allowing for larger models and better accuracy.

[0:05] Part 2. Neural (LLM) Scaling Behaviors

This section delves into empirical scaling behaviors of LLMs, focusing on data, model size, and hyperparameters.

[0:05] Data vs. Performance

What is a data scaling law? * A simple formula that maps dataset size (n) to error. * **Expectation**: Monotonic, logistic-like curves for generalization error (log-scale) vs. training data set size (log-scale), with distinct regions (small data, power-law, irreducible error).

Scaling laws hold on many different kinds of phenomena! * **Kaplan et al. (2020)**: Showed log-linear relationships between test loss and: * Compute (PF-days) * Dataset Size (tokens) * Parameters (non-embedding) * These relationships hold even in non-standard settings (e.g., when training and test data are different).

[0:05] Conceptual Foundations of Data Scaling Laws

Q: Why do scaling laws show up? * We expect error to be monotonic (more data = less error). * **Q: But why is it a power law / linear in log-log?** * **A**: Estimation error naturally decays polynomially.

Toy Example: Mean Estimation * **Input**: $x_1, \ldots, x_n \sim N(\mu, \sigma^2)$ * **Task**: Estimate the average, $\hat{\mu} = \frac{1}{n} \sum x_i$. * **Error**: The expected squared error is $E[(\hat{\mu} - \mu)^2] = \frac{\sigma^2}{n}$. * **This is a "scaling law"**: Taking the logarithm of both sides:

$$\log(\text{Error}) = -\log n + 2 \log \sigma$$

This shows a linear relationship in log-log space with a slope of -1. * **Generalization**: Any polynomial rate $1/n^\alpha$ is a scaling law.

Detour: Scaling Laws for (Nonparametric) Learning * Neural nets can approximate arbitrary functions. Let's consider estimating a function $f(x)$ where x_i are uniformly distributed in a 2D unit box, and $y_i = f(x_i) + N(0, 1)$. * **Approach**: Cut the 2D space into boxes of length $n^{-1/4}$. * **Estimation Error**: Informally, we have $\sqrt{n}$ boxes, each getting $\sqrt{n}$ samples. The error is approximately $1/\sqrt{n}$ (plus other smoothness terms). * **In d-dimensions**: This becomes Error $\approx n^{-1/d}$. * **Meaning**: Scaling means $\log(\text{Error}) = -\frac{1}{d} \log n + C$. * **Takeaway**: "Nonparametric" learning has dimension-dependent scaling. The slope of the scaling law is related to the intrinsic dimensionality of the data.

Scaling Law Exponents: An Intriguing Mystery * We expect "nice round numbers" for the slope (e.g., -1 or -0.5) based on simple models. * **Empirical Findings**: * Machine Translation (Hestness et al.): -0.13 * Speech (Hestness et al.): -0.3 * Language Modeling (Kaplan et al.): -0.095 * These exponents are much slower than expected. This suggests that the "effective" dimensionality of the learning problem for neural networks is very high.

Intrinsic Dimensionality Theory of Data Scaling Laws * **Argument (Bahri 2021)**: 1. Scaling laws arise due to polynomial rates of learning $1/n^\alpha$. 2. Scaling argument α is closely connected to the intrinsic dimensionality of the data. * **Caveat**: Estimators of intrinsic dimension are sketchy, and this is not airtight.

[0:05] Other Data Scaling Laws

Data scaling laws are useful for making engineering decisions. * **Related Question**: How does dataset *composition* affect performance? * **Kaplan et al.** showed that data composition affects the *offset* of the scaling law, not the *slope*. * **Implication**: To pick an optimal data mixture, you can run experiments on small models and extrapolate. * **Example**: Optimal data mixture can be found by analyzing the shape of expected error as a function of data source proportion.

- **Scaling Laws Under Data Repetition**
  - **Question**: In practice, we have finite data. How does repeating examples affect scaling?
  - **Observation**: Return on compute when repeating data diminishes rapidly after a few epochs (e.g., 4 epochs).
  - **Modified Scaling Law**: $D' = U_D + U_D R_D (1 - e^{-R_D/U_D})$, where D' is effective data, U_D is unique tokens, and R_D is repetition.
  - **Implication**: There's an effective sample size, and after about 4 epochs, you get rapidly diminishing returns from repeating data.
- **Data Selection Scaling and Accounting for Finiteness**
  - **Problem**: Given that repeated data is less valuable, data selection should be adaptive to scale.
  - **Approach**: Use scaling laws to trade off between repeating high-quality data and including new, lower-quality data.
  - **Example (CMU folks)**: Showed how to balance quality with repetition rate using scaling laws.

Recap: Data Scaling Laws * Remarkably linear relationship between log-data size and log-error. * Holds across domains and models. * Theory understanding: Similar to generalization bounds (mean estimation example). * Applications: Data collection / curation.

[0:05] Scaling Laws for Model Engineering

Now, let's shift to **model scaling**, which is often more mysterious.

Our Motivation: How can we efficiently design huge LLMs? * **Choices**: LSTMs vs. Transformers, Adam vs. SGD, etc. * **Resource Allocation**: How should we allocate limited resources? * Train

models longer vs. train bigger models? * Collect more data vs. get more GPUs? * **Scaling laws provide a simple procedure to answer these questions.**

[0:05] Hyperparameter Questions

We'll consider some of these choices in the context of the classic Kaplan scaling paper. 1. Architecture 2. Optimizer 3. Aspect ratio / depth 4. Batch size

1. Architecture: Transformers vs. LSTMs * **Q**: Are transformers better than LSTMs? * **Brute force way**: Spend tens of millions to train an LSTM GPT-3. * **Scaling law way**: Train LSTMs and Transformers across various compute levels. * **Observation**: Transformers consistently outperform LSTMs with a constant factor gap in compute efficiency (e.g., 15x more efficient). This gap holds across different numbers of layers.

Many Architectures (Tay et al.) * **Cross-architecture scaling (Tay et al., Google)**: Compared many architectures (ALBERT, DConv, Funnel, Transformer-GLU, LCconv, MLP Mixer, MoS Transformer, Switch Transformer, Universal Transformer) against a Transformer baseline. * **Method**: Scaled each architecture across different FLOPs budgets and plotted negative log-perplexity. * **Observation**: Only architectures like Gated Linear Units (GLU) and Mixture of Experts (MoE) (e.g., Switch Transformer) consistently beat the Transformer baseline. This provides evidence for which architectures are worth scaling up.

2. Optimizer Choice * **Q**: What about Adam vs. SGD? * **Hestness et al. (2017)**: Compared SGD and Adam for Recurrent Highway Networks (RHNs, pre-transformers). * **Observation**: Similar to architecture choice, there's a constant factor gap in compute effectiveness between Adam and SGD. The slopes of the scaling curves are similar, but Adam provides a better offset (lower loss for the same compute).

3. Depth/Width: Number of Layers * **Q**: Does depth or width make a huge difference? * **Kaplan et al.** analysis shows that while 1 layer performs significantly worse, models with 2 or more layers have diminishing returns beyond 10^7 parameters. * **Insight**: There's a wide basin of approximately optimal depth/width ratios, rather than a sharp optimal point.

Depth/Width: But not all parameters are made equal * **Observation**: Embedding layer parameters don't behave the same as non-embedding parameters. * If embedding parameters are included in the total parameter count, the scaling law becomes distorted (bends over). * If only non-embedding parameters are considered, the scaling law is much cleaner. * **Related**

Work: Recent papers on scaling laws for mixtures of experts also explore how different types of parameters contribute to scaling.

Do hyperparameters and other Transformer layers scale equally? * **Kaplan et al.** also analyzed the impact of aspect ratios (Feed-Forward Ratio, Attention Head Dimension) on performance. * **Observation**: The shape of the loss curve (loss increase vs. aspect ratio) remains similar across different model sizes (50M, 274M, 1.5B parameters). * **Implication**: You can tune aspect ratios on small models, and the optimal range will likely transfer to larger models. This highlights the importance of being "scale-aware" in hyperparameter tuning.

4. Batch Size: Critical Batch Size * **Batch size** is known to have strong diminishing returns past a certain point. * **Critical batch size**: The minimum number of examples for target loss / minimum number of steps for target loss. * **Perfect Scaling**: When batch size is smaller than the noise scale, increasing batch size is almost equivalent to taking more gradient steps. This is desirable for parallel processing. * **Ineffective Scaling**: Past the critical batch size, increasing batch size no longer effectively reduces noise, as it's dominated by the curvature of the optimization landscape. * **Empirical Analysis**: The critical batch size can be estimated empirically. * **Observation**: As the loss target gets smaller (better performance), the critical batch size tends to get larger. * **Practical Implication**: Training reports (e.g., Llama 3) often show increasing batch size during training as loss decreases.

Batch Size: Selecting the Optimal Batch * **Q**: As we increase both compute and model size, how should we scale training? * **Kaplan et al.** analysis shows that for a given compute budget, the number of total steps can remain relatively constant while batch sizes increase. * **Good news for data parallel processing**: This allows for efficient scaling of training.

5. Learning Rates: muP and Scale-Aware LR Choices * **Problem**: If we naively scale up, the optimal learning rate depends on scale. * **Standard Practice (left plot)**: As model width increases, the optimal learning rate shifts to the left (smaller values). * **Solution**: We need scale-aware initialization and learning rate scaling. * **muP (Maximal Update Parametrization)**: A reparameterization of the model where learning rates are scaled based on model width and other factors (e.g., variance of initialization, output multipliers). * **Our Work (right plot)**: With muP, the optimal learning rate remains stable across different model widths. * **Implication**: Tune learning rate once on a small model, and it directly transfers to the largest scale. * **Industry Adoption**: Labs (e.g., Meta's "MetaP" for Llama 4) are adopting similar ideas to simplify scaling.

[0:05] Caution - Scaling Behaviors Can Differ Downstream

- **Observation**: Scaling is predictable and depends mainly on parameters.
- **Catch**: Downstream scaling can often be much less predictable.
- **Tay et al. (2023)**:
    - **Left Plot (Pre-training)**: Negative log-perplexity (pre-training loss) shows a very clean log-linear relationship with parameters (a surrogate for compute). This means architecture and hyperparameter choices don't significantly alter the fundamental scaling.
    - **Right Plot (Downstream)**: SuperGLUE accuracy (downstream task) shows a much less predictable relationship with parameters. Different architectures and hyperparameters lead to widely varying performance, and the log-linear trend is not as clear.
- **Implication**: Don't assume pre-training scaling directly translates to downstream task performance. Be cautious when extrapolating.
- **Example**: State-space models (e.g., Mamba) show excellent pre-training scaling but may perform differently on specific in-context learning or QA tasks.

[0:05] Some Surprising Takeaways

- The effect of hyperparameters on big LLMs can be predicted *before training*!
    - Optimizer choice
    - Model depth
    - Architecture choice
- **The scaling law based design procedure**:
    1. Train a few smaller models (spanning a couple orders of magnitude compute).
    2. Establish a scaling law (e.g., ADAM vs SGD scaling law).
    3. Select optimal hyperparameters based on the scaling law prediction.
- In many cases, scaling laws won't vary much (slopes will be similar), meaning small model results transfer surprisingly well to larger models.

[0:05] One Important Use of Scaling Laws

Q: Do we need more data or bigger models? * **Context**: Historically, compute was the limiting resource, not data. The question was how to optimally spend a fixed FLOPs budget. * **Joint Data-Model Scaling Laws**: Describe how data and model size relate to error. * **Rosenfeld et al.**

(2020) & Kaplan et al. (2020): Proposed functional forms where error is a sum of terms decaying polynomially with data ($n^{-\alpha}$) and model size ($m^{-\beta}$), plus an irreducible error term (C). *

$$\text{Error} = n^{-\alpha} + m^{-\beta} + C$$

(Rosenfeld et al.) *

$$\text{Error} = [m^{-\alpha} + n^{-1}]^{\beta}$$

(Kaplan et al.) * These functional forms, while somewhat ad-hoc, provide surprisingly good fits to the observed data-model joint error landscape.

Model-Data Joint Scaling is Accurate * **Rosenfeld et al.** demonstrated that by fitting scaling exponents on small data and small models, they could accurately predict the performance of much larger models and datasets. * **Method**: Train models on a small subset of the data-model space (e.g., small models, small data), fit the joint scaling law, and extrapolate to predict performance for larger models and datasets. * **Accuracy**: The predictions (y-axis) closely match the real values (x-axis) for both ImageNet and WikiText-103, showing high accuracy of joint extrapolation. * **Implication**: This allows trading off data and model size to optimize $n^{-\alpha} + m^{-\beta} + C$ with your costs.

[0:05] Compute Tradeoffs

- **Q**: What about other resources? Compute vs. performance?
- **For a fixed compute budget**: Is a big model that's undertrained better than a small model that's well-trained?
- **Kaplan et al. (2020) & Brown et al. (2020)**: Showed that for a fixed compute budget (FLOPs), there's an optimal balance between model size and training data.
    - **Observation**: Plotting test loss vs. parameters (x-axis) for different compute budgets (colors), the minimum loss for each compute budget forms a scaling law.
- **Scaling laws let us navigate this tradeoff.**

[0:05] Caution - 'Optimal' Scaling Laws Are Hard to Get

- **Problem**: Rosenfeld and Kaplan both predict the relationship of data, model, and performance. However, their initial estimates for optimal scaling were quite different.
- **Chinchilla (Hoffmann et al., 2022)**: Argued that previous fits were quite off.
    - **Kaplan et al. (2020)** predicted an optimal ratio of ~2 tokens per parameter.
    - **Chinchilla** found the optimal ratio to be ~20 tokens per parameter, a 10x difference.

    - **Gopher (Rae et al., 2021)** and **Megatron-Turing NLG (Narayanan et al., 2021)** also had ratios closer to Kaplan's initial estimate.
- **Implication**: Precisely determining the "optimal" scaling laws is challenging and sensitive to methodology.

Main Difference - Accounting for LR Schedules * One key reason for the discrepancy between Kaplan's and Chinchilla's estimates lies in how they accounted for learning rate schedules. * **Cosine Learning Rate Schedules**: Models are typically trained with cosine learning rate schedules, which involve a warm-up phase, a decay phase, and a cool-down phase. * **Problem**: You cannot truncate a cosine learning rate schedule early and expect the same model quality as a full run. A model trained for a shorter duration with a truncated schedule is not equivalent to a model trained from scratch with a schedule designed for that shorter duration. * **Kaplan's Approach**: Assumed that truncating training runs (and thus the learning rate schedule) was equivalent to training a smaller model for a shorter time. This assumption was flawed. * **Chinchilla's Approach**: Accounted for the full cosine learning rate schedule, ensuring that models were trained to convergence for each data/model size combination.

Chinchilla in Depth - 3 Methods Chinchilla authors suggested 3 ways of fitting scaling laws, which mostly (minus method 3) suggest similar constants.

1. **Minimum over training runs (Method 1)**:

    - **Approach**: Overlay training curves for various model sizes and compute budgets. Identify the "lower envelope" of these curves, representing the minimum loss achievable for a given FLOPs budget.
    - **Observation**: The minimum over the union of all training curves is a power law.
    - **Result**: This method yields optimal parameter-to-FLOPs and token-to-FLOPs ratios that are consistent with 0.5 for both coefficients ($N_{opt} \propto C^a$, $D_{opt} \propto C^b$).

2. **IsoFLOPs (Method 2)**:

    - **Approach**: Pick a range of FLOPs budgets. For each budget, vary the total parameter count and take the minimum loss over these convex shapes (IsoFLOP curves).
    - **Observation**: The minima form a power law.
    - **Result**: This method also yields optimal parameter-to-FLOPs and token-to-FLOPs ratios consistent with 0.5. This is considered conceptually straightforward.

3. **Joint Fits (Method 3)**:

    - **Approach**: Run a bunch of models on the size-data grid. Use least squares to fit a joint scaling law (like the Rosenfeld/Kaplan functional forms).
    - **Observation**: This method is messier and yields different coefficients (e.g., 0.73 for a and 0.27 for b in Kaplan's initial work).

Fun Addendum - Errors in Chinchilla Method 3 * **Discovery**: Some authors (Besiroglu et al., 2024) later found that Method 3 in the original Chinchilla paper was likely flawed. * **Process**: They performed data forensics, recovered the raw data, and re-did the fit. * **Result**: The re-fit yielded results more consistent with Methods 1 and 2. The original fit had non-zero mean residuals, indicating a biased fit. Correcting this bias brought Method 3's estimates in line with the other two methods. * **Conclusion**: The original authors had both the idea and the data right, but a minor error in curve fitting led to a misleading result.

[0:05] Important Note - Train-Optimal May Not Be What You Want

- **Chinchilla's Aim**: To tell you what gives the best model for fixed training compute.
- **Reality**: Most of the compute in a real deployment is inference.
- **Implication**: We should "over-train" (train with more tokens per parameter than what Chinchilla suggests for training optimality).
- **Trend**: Over time, the tokens per parameter ratio has steadily increased:
    - GPT-3: 2 tokens / param
    - Chinchilla: 20 tokens / param
    - LLaMA 65B: 22 tokens / param
    - Llama 2 70B: 29 tokens / param
    - Mistral 7B: 110 tokens / param
    - Llama 3 70B: 215 tokens / param
- **Reason**: The more usage we expect, the more it becomes worth it to pay the upfront cost of training with more data to get a better model, rather than incurring higher ongoing inference costs for a less capable model.

[0:05] Recent Example for Different (Diffusion) Models

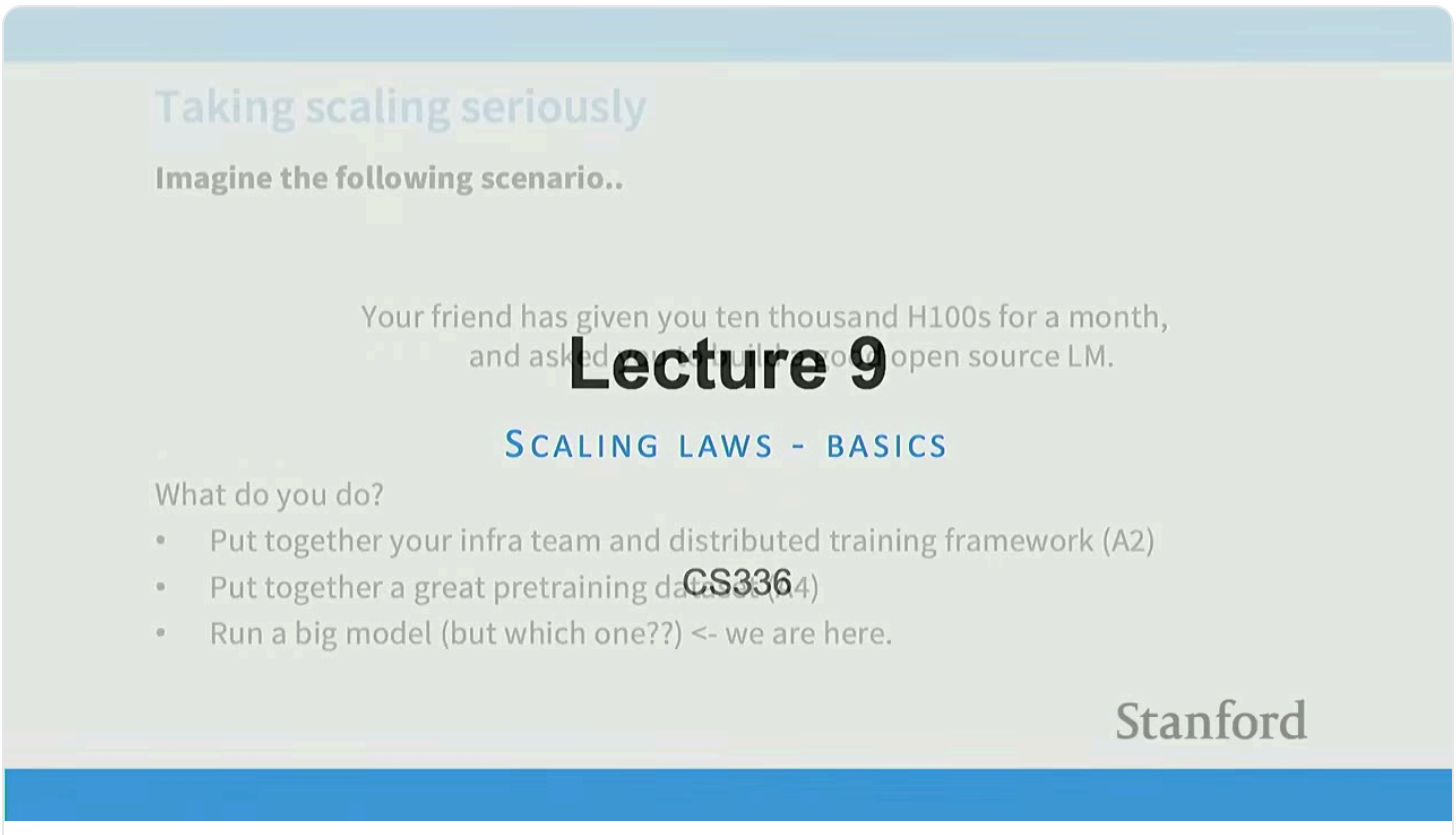
- **Observation**: Methods like IsoFLOPs are pretty easy to execute, and our group has replicated these results.
- **Gulrajani+ (2023)**: Applied IsoFLOP analysis to auto-regressive and diffusion models for text.

    - **Auto-regressive models**: Showed the familiar chinchilla-like curves and optimal scaling.
    - **Diffusion models**: Despite being a totally different generative model, they also show very similar curves and predictable scaling.
    - **Conclusion**: Plotting the minimum across these shows predictable scaling for both, separated by a constant offset.
- **Takeaway**: Scaling laws are not just for cherry-picked examples; they seem to happen pretty naturally as you work on new models or environments.

[0:05] Scaling Laws for Models and Compute

- Log-linearity extends to model parameters and compute!
- **Lets us set the following based on small models**:
    - Pick optimizer
    - Pick architecture and model sizes
- **Also lets us make smart resource tradeoffs**:
    - Big models vs. more data?

[0:05] Recap: Scaling Laws - Surprising and Useful!



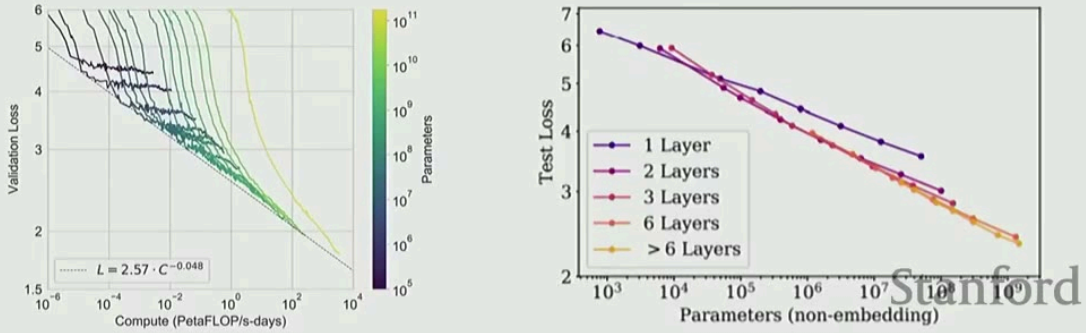
0:14 Lecture 9: Scaling Laws - Basics, introducing a scenario for large-scale language model training.

Today: simple, predictive 'laws' for behaviors of LMs

The approach -

scaling laws which are simple, predictive rules for model performance

Old and unpleasant: tune hyperparameters on big models
New (over?) optimism: tune on small models, extrapolate to large ones



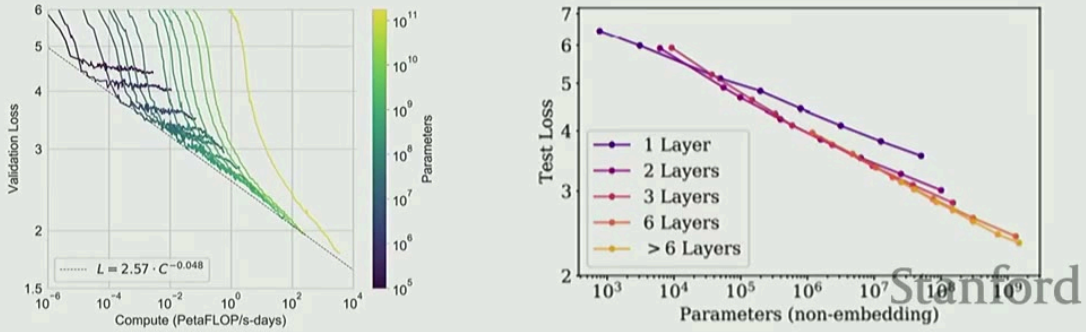
1:42 Slide title: Today: simple, predictive 'laws' for behaviors of LMs, with two graphs showing validation and test loss.

Today: simple, predictive 'laws' for behaviors of LMs

The approach -

scaling laws which are simple, predictive rules for model performance

Old and unpleasant: tune hyperparameters on big models
New (over?) optimism: tune on small models, extrapolate to large ones



2:20 Slide title: Today: simple, predictive 'laws' for behaviors of LMs, with two graphs showing validation and test loss.

Sample complexity and rates

Theorists have thought about 'scaling' for a long time..

$$\epsilon(\hat{h}) \leq \left(\min_{h \in H} \epsilon(h)\right) + 2\sqrt{\frac{1}{m} \log \frac{2k}{\delta}}$$

(learning in a finite set of k hypotheses)

Under the assumptions of Theorem 1.5, the rate of convergence of the estimator $\hat{p}_n(x_0)$ is $\psi_n = n^{-\frac{\beta}{2\beta+1}}$, which means that for a finite constant C and for all $n \geq 1$ we have

$$\sup_{p \in \mathcal{P}(\beta, L)} \mathbb{E}_p\left[(\hat{p}_n(x_0) - p(x_0))^2\right] \leq C\psi_n^2.$$

(generative modeling for smooth densities)

But these are upper bounds, not actual, realized loss values.

Stanford

https://www.cs.cmu.edu/~epxing/Class/10701/slides/lecture16-VC.pdf
Hall, 1989

4:01 Slide title: Sample complexity and rates, showing two mathematical equations related to learning theory.

Earliest (data) scaling law paper – 1993

Learning Curves: Asymptotic Values and Rate of Convergence

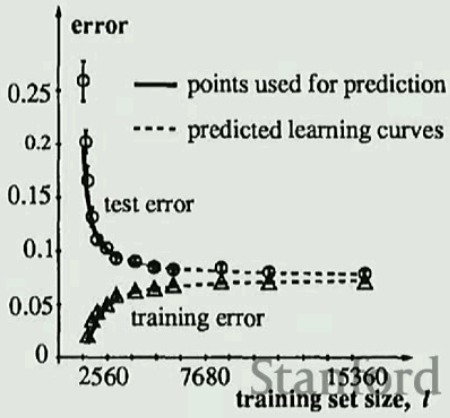
Corinna Cortes, L. D. Jackel, Sara A. Solla, Vladimir Vapnik, and John S. Denker
AT&T Bell Laboratories
Holmdel, NJ 07733

Abstract

Training classifiers on large databases is computationally demanding. It is desirable to develop efficient procedures for a reliable prediction of a classifier's suitability for implementing a given task, so that resources can be assigned to the most promising candidates or freed for exploring new classifier candidates. We propose such a practical and principled predictive method. Practical because it avoids the costly procedure of training poor classifiers on the whole training set, and principled because of its theoretical foundation. The effectiveness of the proposed procedure is demonstrated for both single- and multi-layer networks.

A typical example of learning curves is shown in Fig. 2. The test error is always larger than the training error, but asymptotically they reach a common value, a. We model the errors for large sizes of the training set as power-law decays to the

$$\mathcal{E}_{\text{test}} = a + \frac{b}{l^\alpha} \quad \text{and} \quad \mathcal{E}_{\text{train}} = a - \frac{c}{l^\beta}$$



Stanford

5:28 Slide title: Earliest (data) scaling law paper – 1993, displaying a paper title and a graph of error vs. training set size.

Early history of scaling laws – data scaling

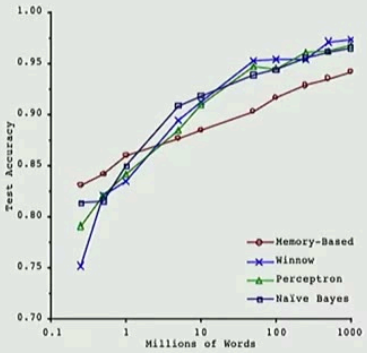


Figure 1. Learning Curves for Confusion Set Disambiguation

these results suggest that we may want to reconsider the trade-off between spending time and money on algorithm development versus spending it on corpus development. At least for the problem of confusable disambiguation, none of the learners tested is close to asymptoting in performance at the training corpus size commonly employed by the field.

Log-linear scaling with data [Banko and Brill '01]

Stanford

7:13 Slide title: Early history of scaling laws – data scaling, showing a graph of test accuracy vs. millions of words.

Hestness et al 2017

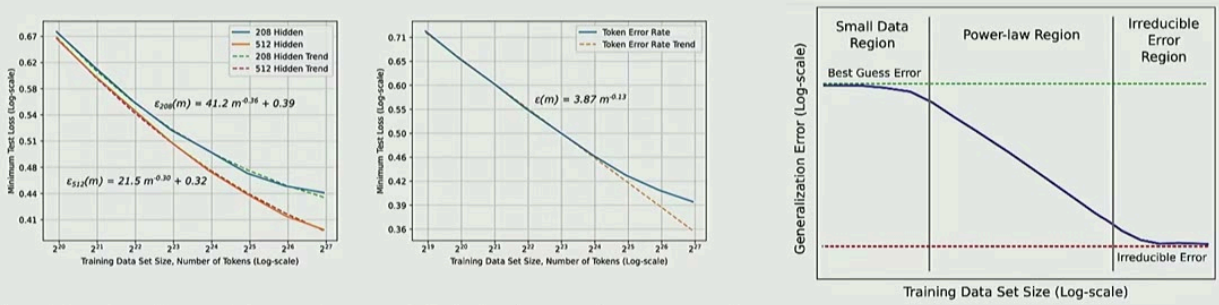


Figure 1: Neural machine translation learning curves. Left: the learning curves for separate models follow $\varepsilon(m) = \alpha m^{\beta_g} + \gamma$. Right: composite learning curve of best-fit model at each data set size.

Earliest 'large scale neural' scaling work: Hestness 2017

Predictable scaling on many tasks (MT, LM, Speech) and hypothesized scaling shape.

Stanford

8:40 Slide title: Hestness et al 2017, presenting three graphs illustrating neural machine translation learning curves and generalization error.

Part 2. Neural (LLM) scaling behaviors

1.Data vs performance

"Are there simple rules that determine how data affects performance?"

2. Data vs model size

Do we train on more data or bigger models?

3. Hyper-parameters vs performance

"How should we set hyperparameters on the big model??"

Stanford

10:50 Slide title: Part 2. Neural (LLM) scaling behaviors, outlining three key areas: data vs performance, data vs model size, and hyperparameters vs performance.

Part 2. Neural (LLM) scaling behaviors

1.Data vs performance
"Are there simple rules that determine how data affects performance?"

2. Data vs model size
Do we train on more data or bigger models?

3. Hyper-parameters vs performance
"How should we set hyperparameters on the big model??"

Stanford

12:21 Slide title: Part 2. Neural (LLM) scaling behaviors, outlining three key areas: data vs performance, data vs model size, and hyperparameters vs performance.

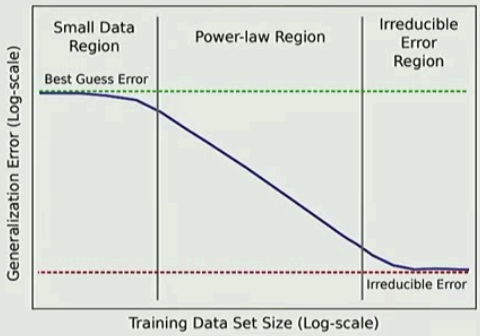
Data vs performance

What's a data scaling law?

Data scaling laws : simple formula that maps dataset size (n) to error

What do we expect out of scaling laws?

Monotonic, logistic-like curves



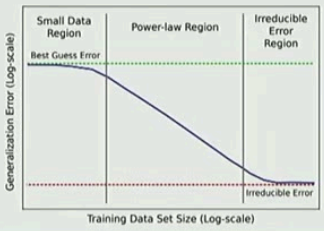
Stanford
[Hestness+ 2017]

13:43 Slide title: Data vs performance, defining data scaling laws and showing a generalization error curve.

Conceptual foundations of data scaling laws.

Q: Why do scaling laws show up?

We know error should be monotone



But why is it a power law / linear in log-log?

A (?): Estimation error naturally decays polynomially.

But this answer may take a moment to understand. Let's work through an example.

Example: If our task is to estimate the mean of a dataset, what's the scaling law?

Stanford

15:13 Slide title: Conceptual foundations of data scaling laws, posing a question about why scaling laws appear.

Toy example: mean estimation

Input: $x_1 \ldots x_n \sim N(\mu, \sigma^2)$

Task: estimate the average as $\hat{\mu} = \frac{\sum_i x_i}{n}$

What's the error? By standard arguments..

$$\mathrm{E}[(\hat{\mu} - \mu)^2] = \frac{\sigma^2}{n}$$

This is a 'scaling law'

$$\log(Error) = -\log n + 2\log\sigma$$

More generally, any polynomial rate $1/n^{\alpha}$ is a scaling law

Stanford

16:57 Slide title: Toy example: mean estimation, presenting equations for estimating the average and its error.

- **Data scaling**: Understand how data affects models, clean theory.
- **Model scaling**: Dramatically reduce costs for training.
- **Scaling as prediction**: Understand what problems can be 'brute forced'.

Practical Takeaways

- **Efficiency**: Scaling laws provide a principled way to make engineering decisions (architecture, hyperparameters, resource allocation) for large models without needing to train them fully.
- **Predictability**: Many aspects of model performance (loss) exhibit predictable power-law relationships with resources (data, parameters, compute) on log-log scales.
- **Resource Allocation**: Scaling laws help optimize the trade-off between model size, dataset size, and compute budget to achieve the best performance for a given cost.
- **Architecture/Optimizer Choice**: Small-scale scaling experiments can reveal which architectures or optimizers are fundamentally more efficient, even if the absolute performance is low.
- **Hyperparameter Tuning**: Scale-aware initialization (like muP) can make optimal learning rates stable across different model scales, simplifying tuning.
- **Data Strategy**: Scaling laws inform decisions about data composition (offset vs. slope) and repetition (diminishing returns).

Open Questions / Things to Remember

- **Theoretical vs. Empirical**: While statistical learning theory provides bounds, empirical scaling laws often reveal different, slower decay rates, suggesting high effective dimensionality.
- **Beyond Perplexity**: Scaling law predictability is strong for pre-training objectives (like perplexity) but can be much less reliable for downstream tasks. This is a critical caution.
- **Not All Parameters Are Equal**: Embedding parameters can behave differently from non-embedding parameters, affecting scaling law linearity.
- **Learning Rate Schedules**: Proper accounting for learning rate schedules (e.g., full cosine cycles) is crucial for accurate scaling law predictions, as shown by the Chinchilla paper's refinement of Kaplan's work.
- **Train-Optimal vs. Inference-Optimal**: The "optimal" scaling for training (minimizing FLOPs for a given loss) might not be the optimal strategy for deployment, where inference cost and

model capability at small sizes are paramount. This has led to a shift towards "over-training" (more tokens per parameter) in recent LLMs.
- **Robustness**: The widespread applicability of scaling laws across different model types (auto-regressive, diffusion) and tasks suggests a fundamental underlying principle in deep learning.

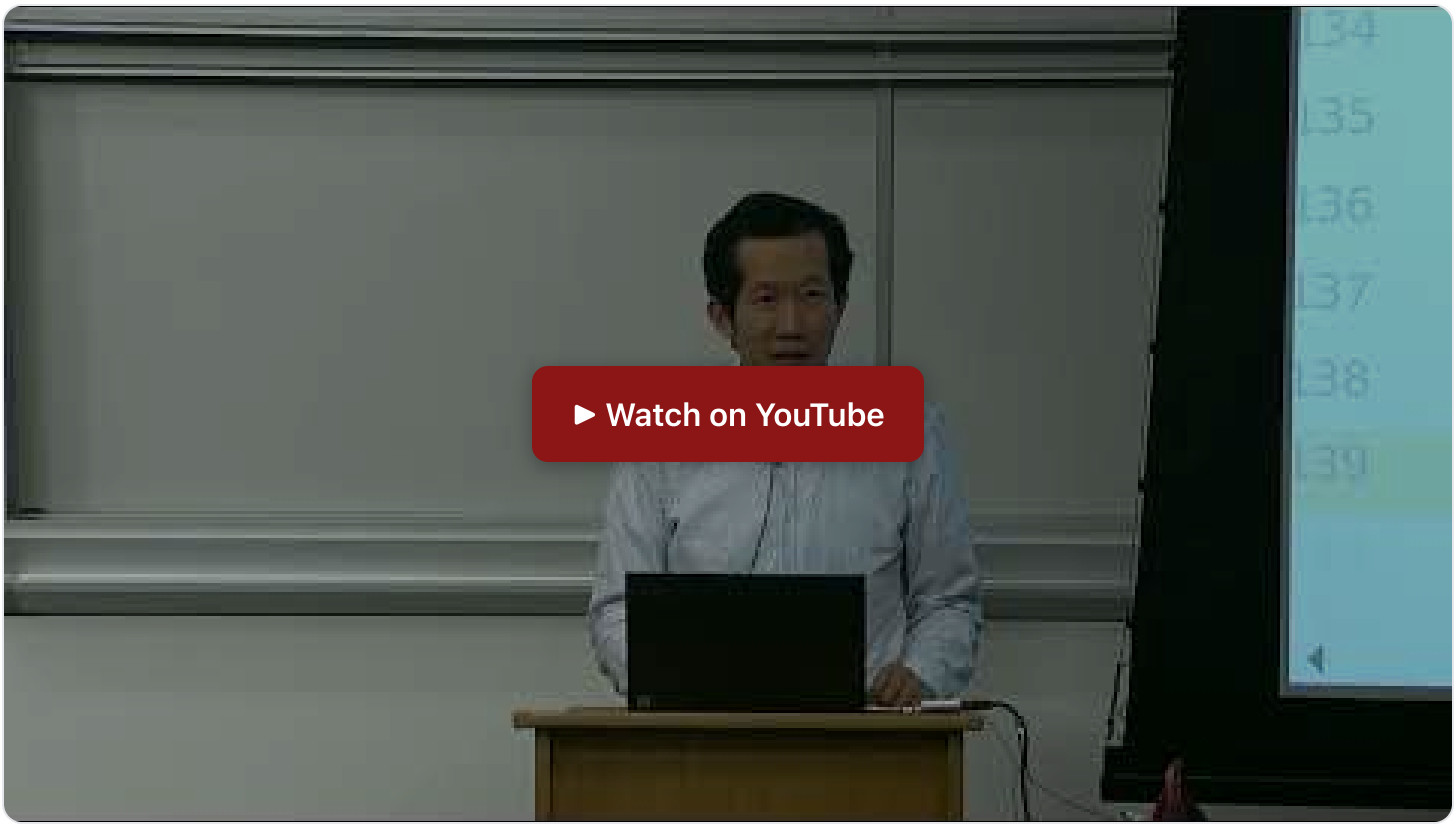
← Lec 08

Lec 10 →

Lecture 10: Inference

Stanford CS336 · Spring 2025 · 1:22:52 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



TL;DR

- Inference is a critical part of language model deployment, distinct from training due to its memory-limited and dynamic nature.
- Key metrics for inference efficiency are Time-to-First-Token (TTFT), latency (tokens/second), and throughput (tokens/second).
- The KV cache is a major bottleneck in transformer inference, leading to memory-bound operations, especially during token generation.
- Architectural innovations (GQA, MLA, local attention, state-space models, diffusion models) and optimization techniques (quantization, pruning, speculative decoding, continuous

batching, PagedAttention) are crucial for improving inference efficiency.
- The future of efficient inference lies in radical architectural changes and leveraging systems-level optimizations.

Key Concepts

- **Inference Workload**: Generating responses from a fixed, trained model.
- **Memory-Limited vs. Compute-Limited**: A workload is memory-limited if its performance is bottlenecked by memory bandwidth, and compute-limited if it's bottlenecked by FLOPs.
- **Arithmetic Intensity**: The ratio of FLOPs to bytes transferred, indicating how much computation is done per byte of data moved.
- **KV Cache**: Stores key and value vectors from previous tokens to avoid recomputing them during autoregressive generation.
- **Prefill vs. Generation**: Prefill is the initial processing of the prompt (compute-limited), while generation is the sequential output of new tokens (memory-limited).
- **Grouped-Query Attention (GQA)**: Reduces KV cache size by sharing key/value heads across multiple query heads.
- **Multi-Head Latent Attention (MLA)**: Reduces KV cache size by projecting key/value vectors into a lower-dimensional latent space.
- **Local Attention**: Restricts attention to a fixed-size window of past tokens, making KV cache size independent of sequence length.
- **State-Space Models (SSMs)**: Alternative architectures to transformers, drawing from signal processing, aiming for sub-quadratic time complexity for long contexts.
- **Diffusion Models for Text Generation**: Generate all tokens in parallel and iteratively refine them, offering potential for significant speedups over autoregressive models.
- **Quantization**: Reduces the precision of numbers (e.g., from FP16 to INT8) to decrease memory footprint and FLOPs, but can impact accuracy.
- **Model Pruning**: Removes unimportant parts of a model to make it smaller and faster, often followed by distillation to recover accuracy.
- **Speculative Decoding**: Uses a smaller, faster draft model to generate multiple tokens in parallel, which are then verified by the larger target model.
- **Continuous Batching / PagedAttention**: Systems-level optimizations to handle dynamic inference workloads by efficiently managing GPU memory and scheduling requests.

Detailed Notes

[00:05] Introduction to Inference



0:05 The slide shows the title of the lecture: "CS336: Language Modeling from Scratch".



0:13 The speaker is standing at a podium with a laptop, facing the audience.

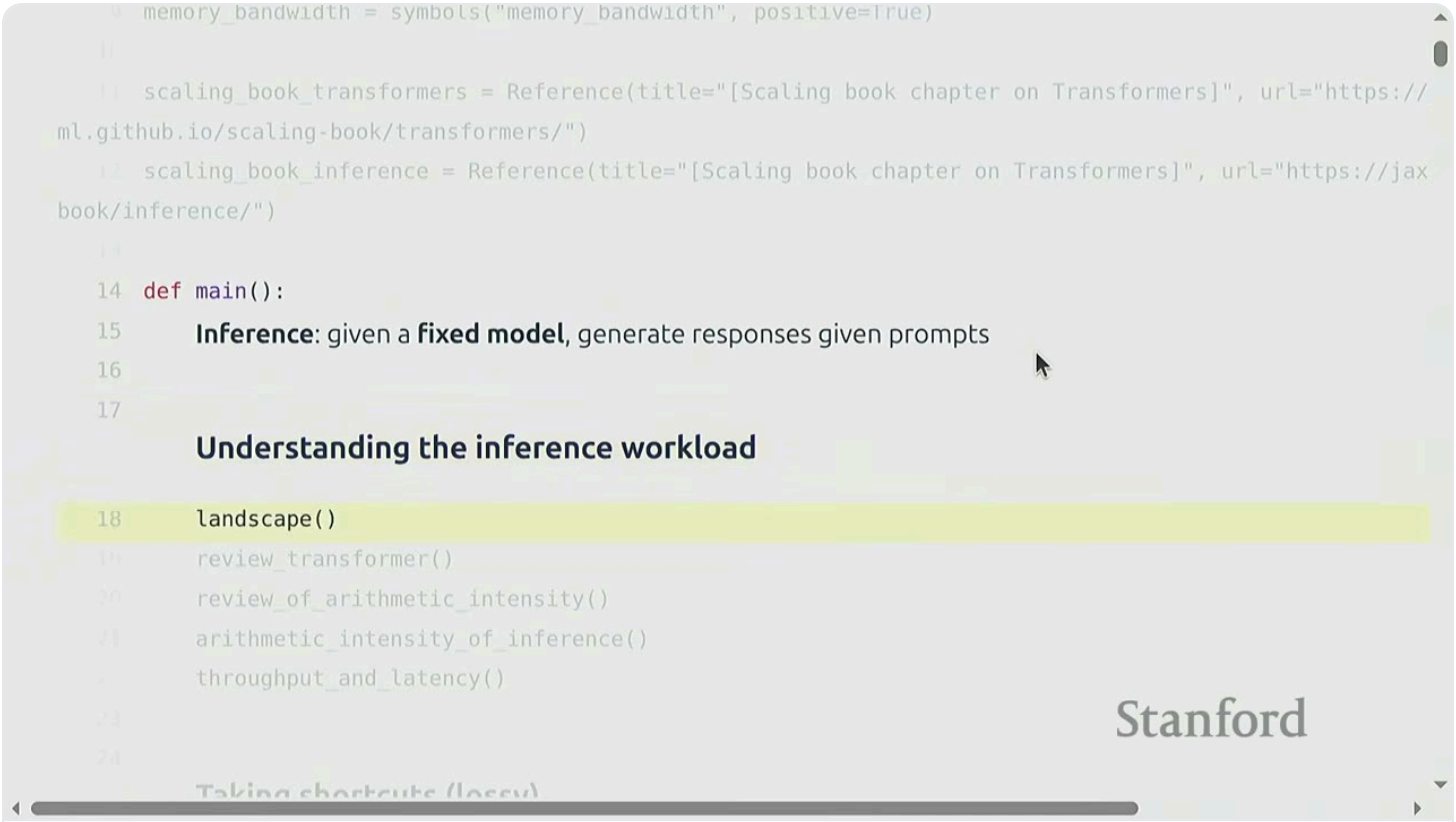
This lecture, Lecture 10, will take a break from scaling laws to discuss **inference**.

Inference: Given a fixed, trained model, generate responses given prompts.

We will cover: 1. Understanding the inference workload. 2. Ways to make inference faster.

Inference is a deep topic, and this lecture will condense many concepts.

[00:54] Where Inference Shows Up



0:54 The slide defines "Inference" and introduces "Understanding the inference workload" as a topic.

Inference is crucial in many applications: - **Actual Use**: Chatbots, code completion (e.g., Cursor), batch data processing using language models. These demand inference to generate tokens. - **Model Evaluation**: Assessing model performance (e.g., on instruction following datasets) requires running inference. - **Test-Time Compute**: Techniques like "thinking more" before outputting a final answer often involve multiple inference steps (e.g., chain-of-thought prompting). - **Training via Reinforcement Learning (RL)**: In RL, models sample responses and evaluate them based on a reward signal, which requires inference.

Inference underpins many fundamental functions of language models, highlighting its importance.

[02:16] Why Efficiency Matters for Inference

- **Training** is a one-time cost.
- **Inference** is repeated many times.

Illustrative Stats: - Sam Altman (OpenAI): "openai now generates about 100 billion words per day. all people on earth generate about 100 trillion words per day." (Feb 9, 2024) - Aman Sanger (Cursor): "Cursor writes almost 1 billion lines of accepted code a day. To put it in perspective, the entire world produces just a few billion lines a day." (Apr 28, 2023)

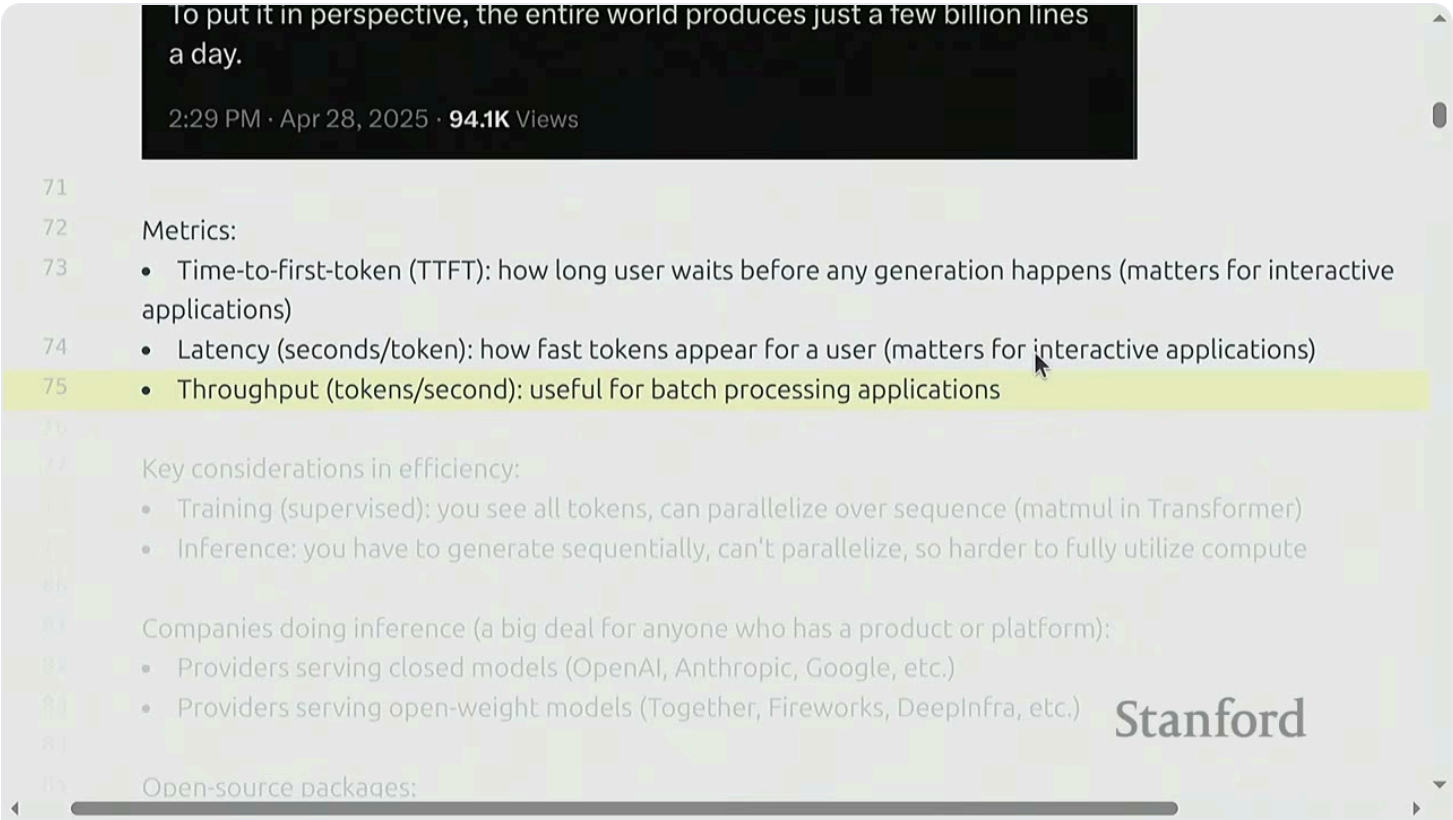
These figures demonstrate the massive scale of inference, making its efficiency a critical concern.

[03:06] Metrics for Inference

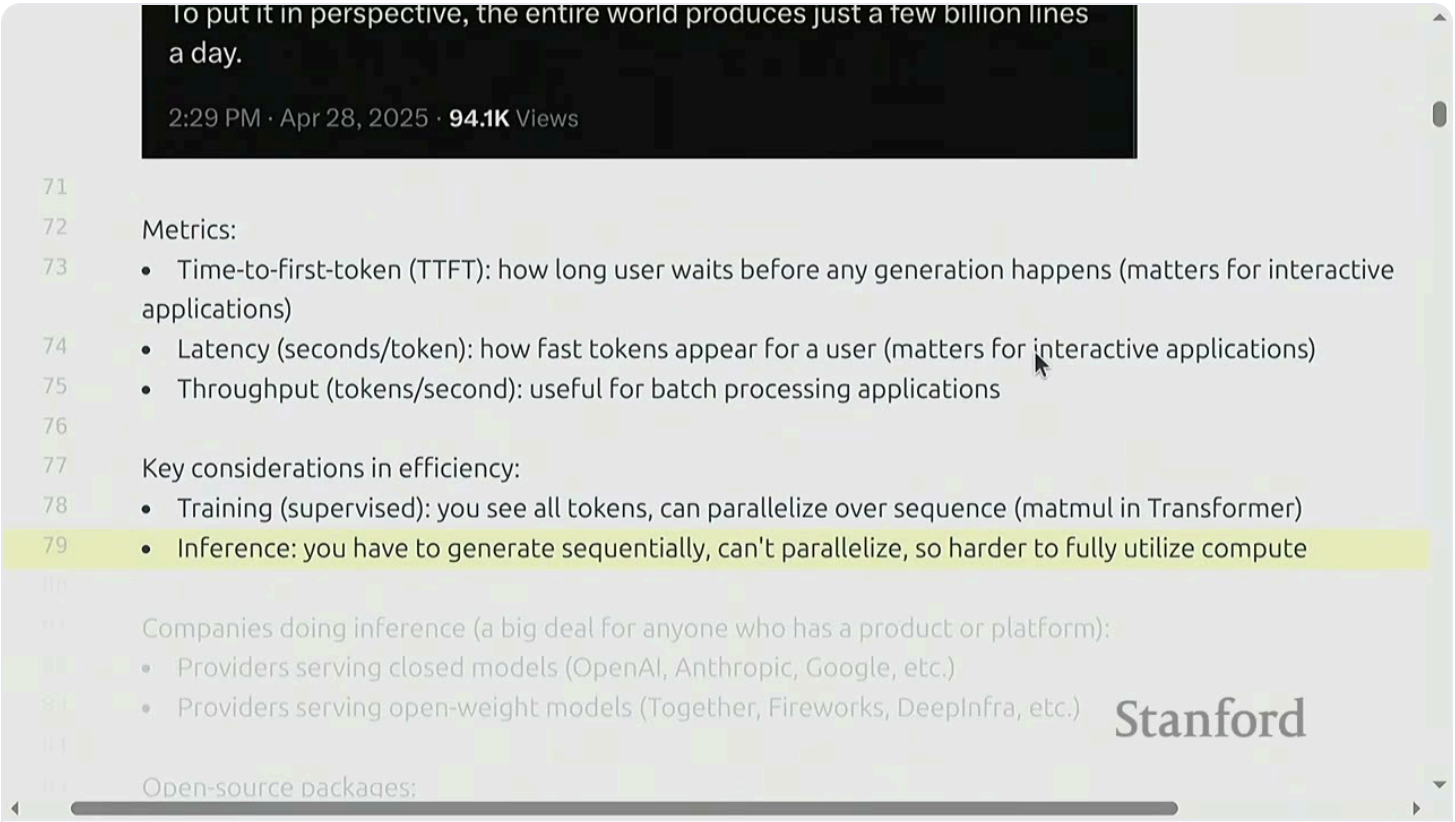
Three key metrics are used to measure inference performance: 1. **Time-to-First-Token (TTFT)**: How long a user waits before *any* generation happens. Crucial for interactive applications where immediate feedback is desired. 2. **Latency (tokens/second)**: How fast tokens appear *after* the first token. Also important for interactive applications to maintain a smooth conversation flow. 3. **Throughput (tokens/second)**: The total number of tokens generated per second across *all* users or requests. This is vital for batch processing and overall system capacity.

Note: High throughput doesn't necessarily imply low latency, as some requests might take a long time to complete while others are processed quickly.

[04:08] Key Considerations in Efficiency



3:54 The slide lists metrics for evaluating inference: TTFT, Latency, and Throughput.



5:06 The slide highlights key considerations for efficiency in training versus inference.

Training (Supervised): - You see all tokens in the sequence simultaneously. - This allows for parallelization over the sequence (e.g., matrix multiplications in Transformers). - This is highly efficient for utilizing hardware.

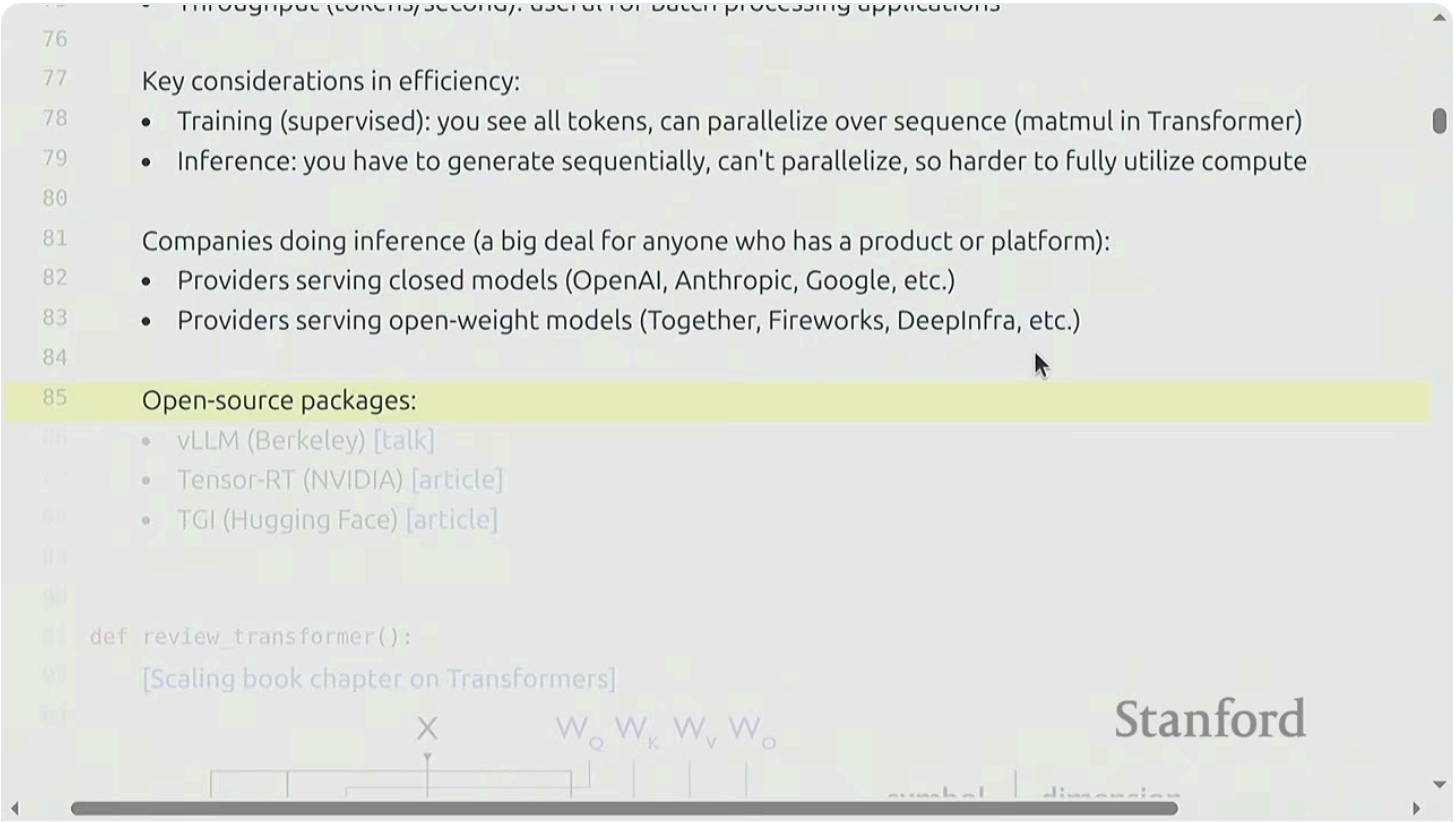
Inference: - You have to generate tokens **sequentially** (autoregressively). - This means you **cannot parallelize** the generation process. - Consequently, it's harder to fully utilize compute and often becomes **memory-limited**.

This sequential nature is the core challenge that makes inference harder and often memory-bound.

Who Cares About Inference Efficiency? - **Companies serving closed models** (OpenAI, Anthropic, Google, etc.) - **Providers serving open-weight models** (Together, Fireworks, DeepInfra, etc.) - Various **open-source packages** (vLLM, Tensor-RT, TGI) are dedicated to optimizing inference.

Academics often focus on training and model performance, but industry practitioners prioritize inference efficiency due to its direct impact on user experience and operational costs.

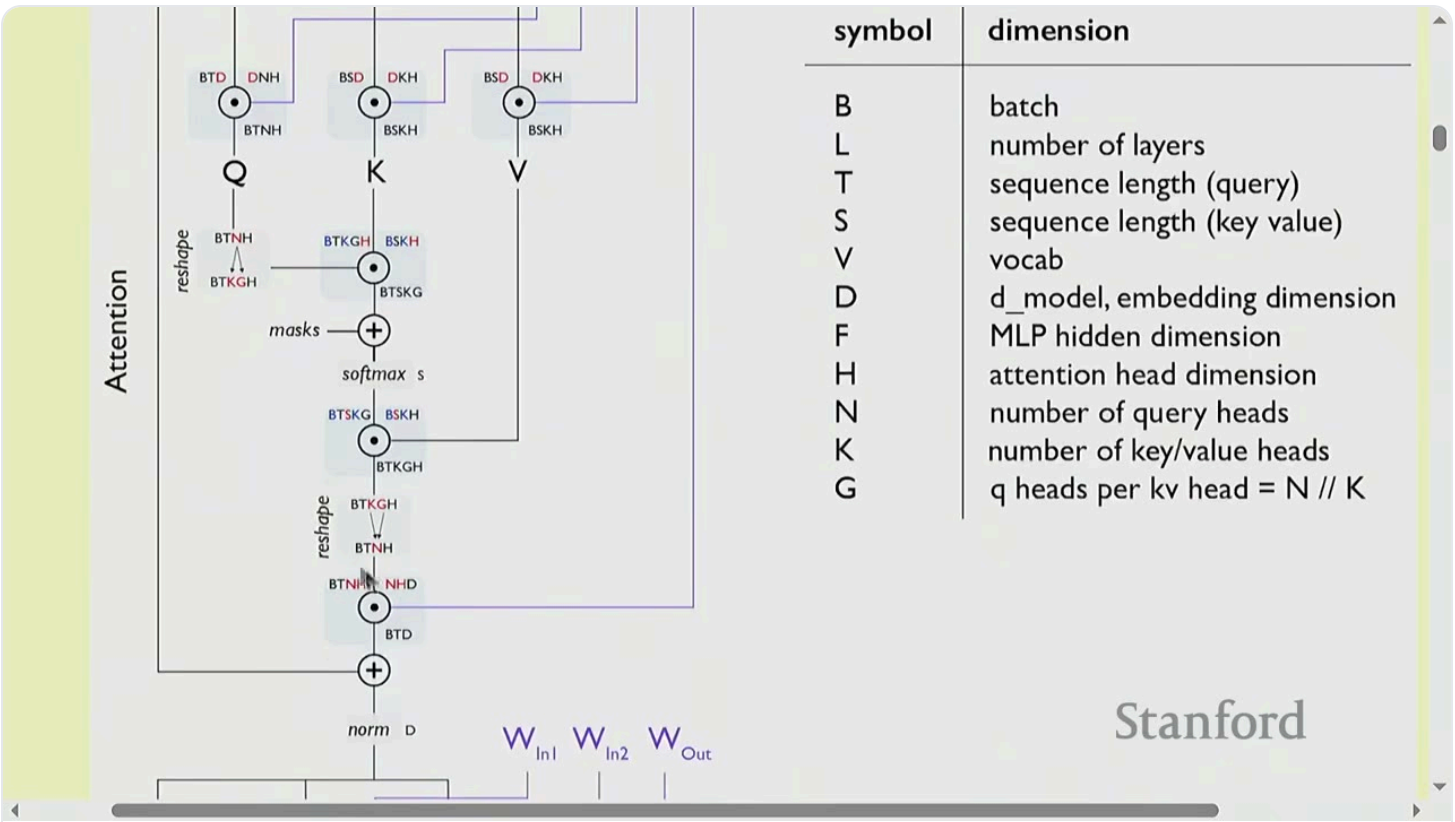
[05:53] Understanding the Inference Workload



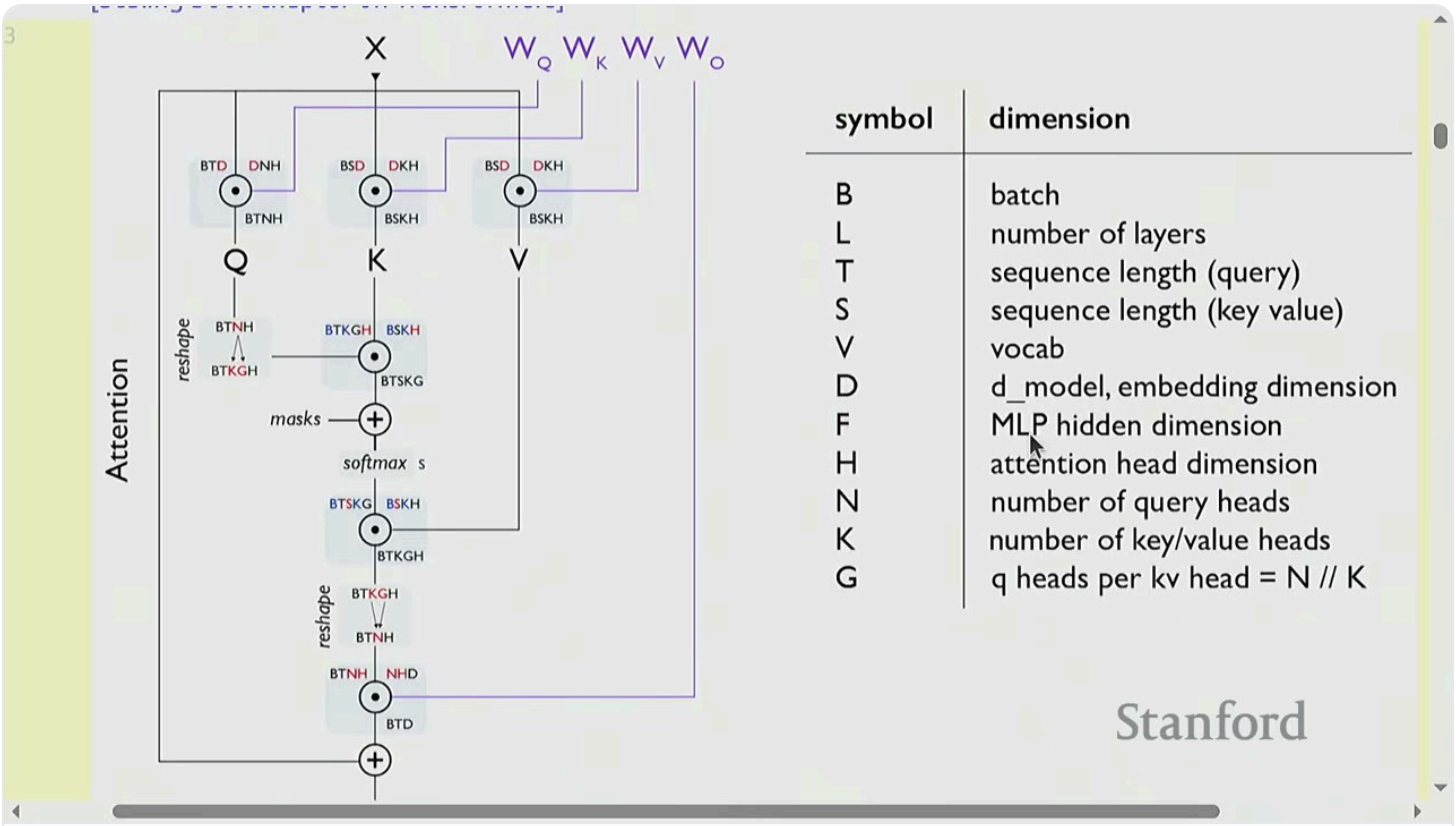
5:39 The slide lists companies doing inference and open-source packages for inference.

To understand inference, we'll review the Transformer architecture and arithmetic intensity.

[06:11] Review of the Transformer

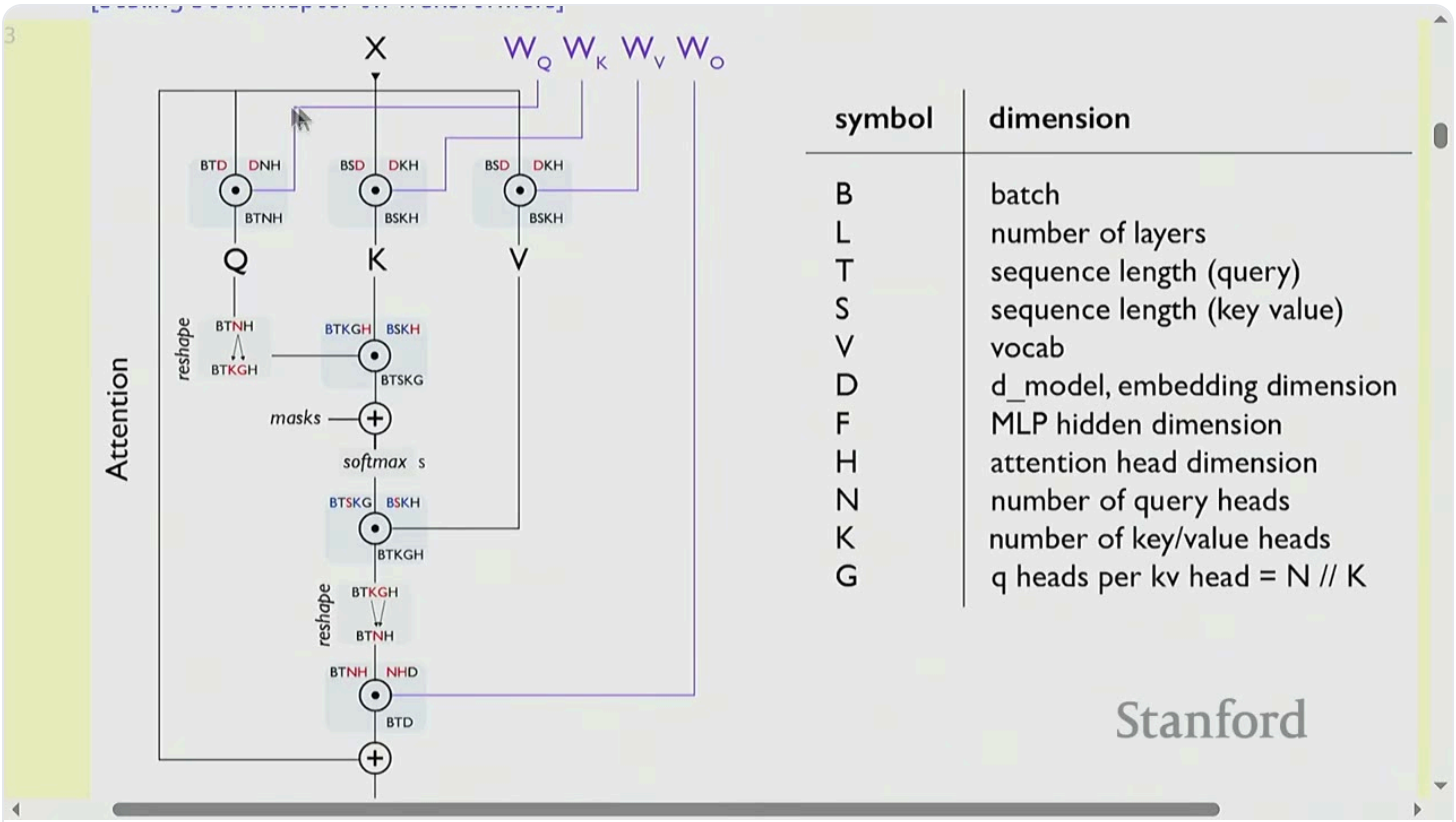


6:36 The slide displays a diagram of the attention mechanism with a table of symbol dimensions.



symbol	dimension
B	batch
L	number of layers
T	sequence length (query)
S	sequence length (key value)
V	vocab
D	d_model, embedding dimension
F	MLP hidden dimension
H	attention head dimension
N	number of query heads
K	number of key/value heads
G	q heads per kv head = N // K

7:08 The slide shows a diagram of the attention mechanism with a table of symbol dimensions.



symbol	dimension
B	batch
L	number of layers
T	sequence length (query)
S	sequence length (key value)
V	vocab
D	d_model, embedding dimension
F	MLP hidden dimension
H	attention head dimension
N	number of query heads
K	number of key/value heads
G	q heads per kv head = N // K

7:37 The slide shows a diagram of the attention mechanism with a table of symbol dimensions.

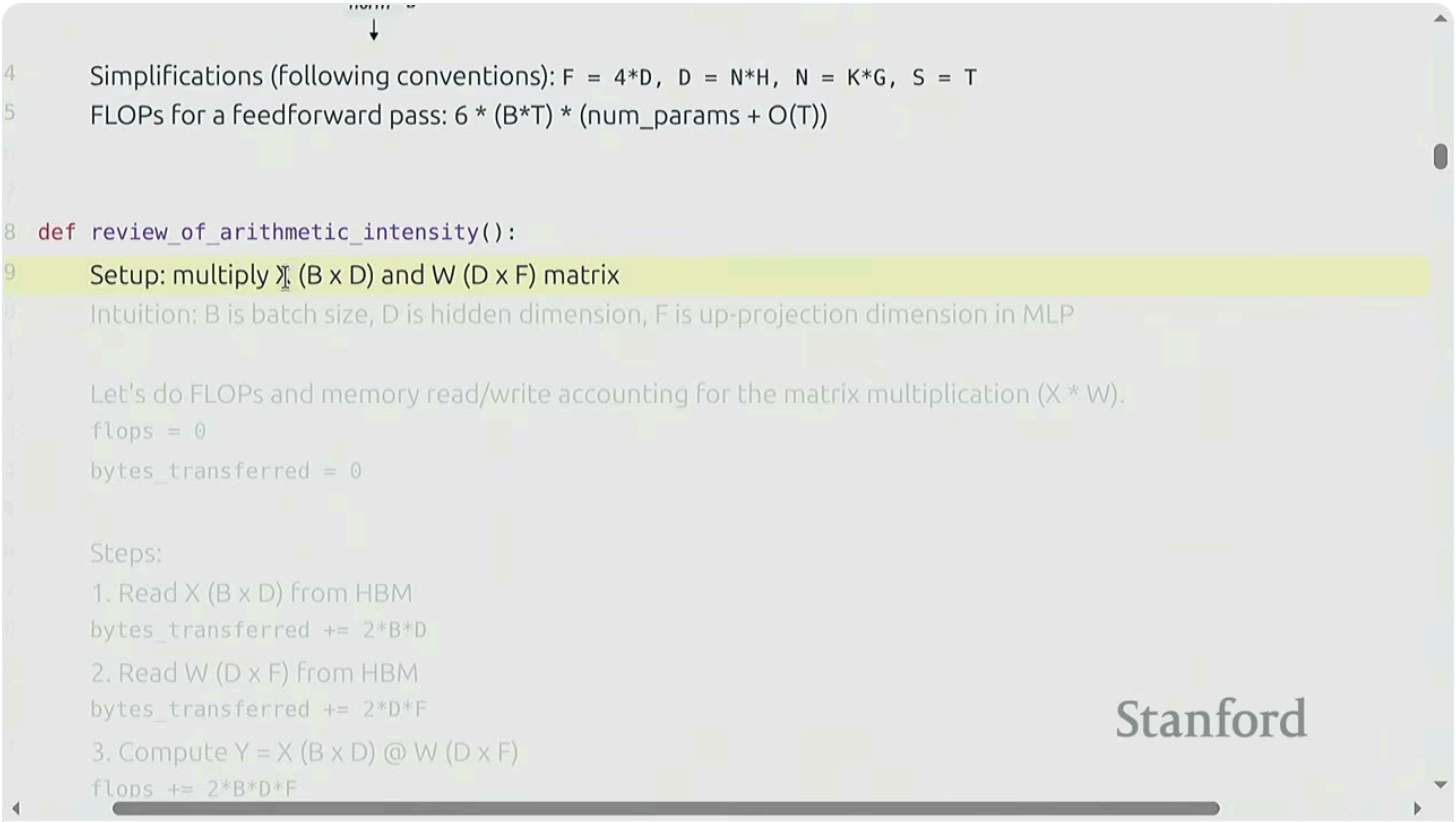
The diagram from the "Scaling book chapter on Transformers" (a recommended resource) illustrates the computation graph of a Transformer.

Notation: - `B` : Batch size (number of sequences) - `L` : Number of layers - `T` : Sequence length (query) - `S` : Sequence length (key/value) - `V` : Vocabulary size - `D` : Model embedding dimension - `F` : MLP hidden dimension (typically $4 \times D$) - `H` : Attention head dimension - `N` : Number of query heads ($N \times H = D$) - `G` : Number of key/value heads (for GQA, $K \times G = N$)

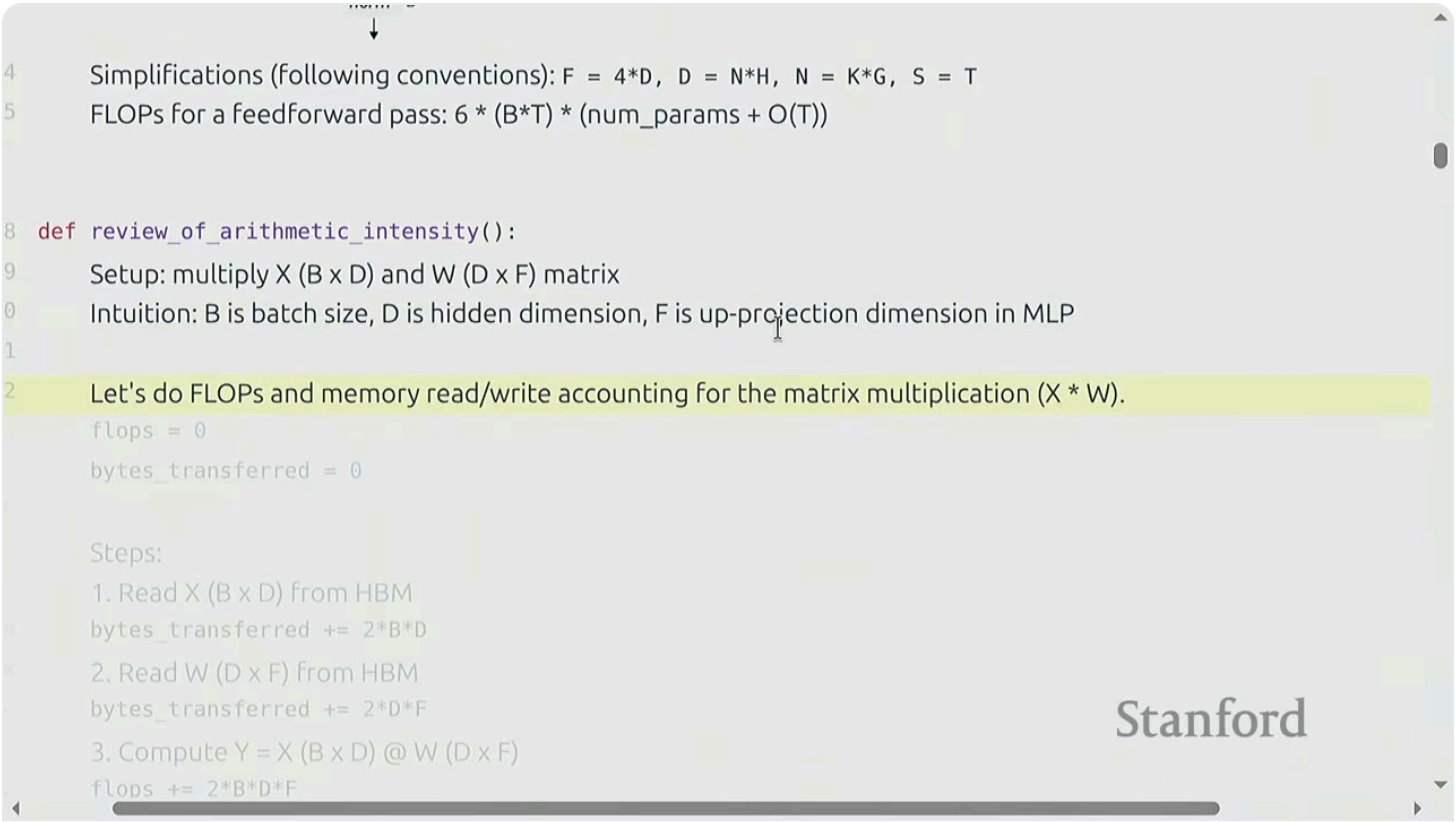
The diagram shows how input `X` passes through attention and MLP layers, involving matrix multiplications with query (Q), key (K), and value (V) matrices.

FLOPs for a feedforward pass: $6 \times (B \times T) \times (\text{num\_params} + O(T))$

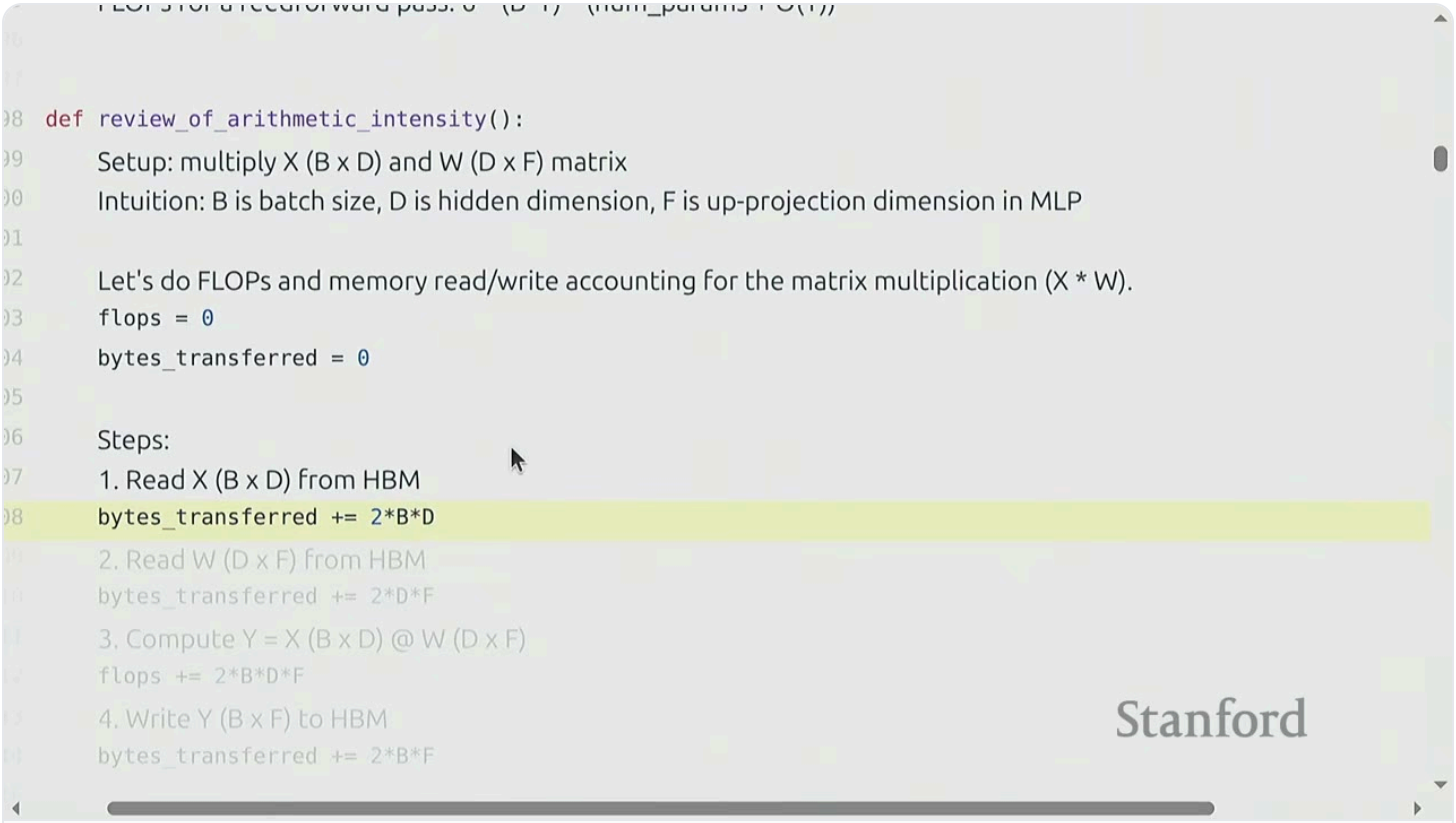
[08:10] Review of Arithmetic Intensity



8:28 The slide shows code for `review_of_arithmetic_intensity` and explains matrix multiplication setup.



8:52 The slide highlights the section on calculating FLOPs and memory read/write for matrix multiplication.



9:13 The slide details the steps for calculating bytes transferred during matrix multiplication from HBM.

Arithmetic intensity helps characterize whether a workload is compute-limited or memory-limited.

Setup: Multiply matrix X (B x D) by matrix W (D x F). - `B`: Batch size - `D`: Hidden dimension - `F`: Up-projection dimension in MLP

Steps to calculate FLOPs and bytes transferred (assuming BF16, so 2 bytes/value): 1. Read `X` (B x D) from HBM: `bytes_transferred += 2 * B * D` 2. Read `W` (D x F) from HBM: `bytes_transferred += 2 * D * F` 3. Compute `Y = X @ W` (B x F): `flops += 2 * B * D * F` 4. Write `Y` (B x F) to HBM: `bytes_transferred += 2 * B * F`

Total FLOPs: `2 * B * D * F` **Total Bytes Transferred**: `2 * B * D + 2 * D * F + 2 * B * F`

Arithmetic Intensity: `flops / bytes_transferred` (want to be high).

Simplification: Assuming `B` is much smaller than `D` and `F` (e.g., `B`=100, `D` = `F`=1000s), the arithmetic intensity simplifies to `B`.

Accelerator Intensity (H100): - `flops_per_second` (FP16): 989×10^{12} - `memory_bandwidth`: 3.35×10^{12} - `accelerator_intensity` = `flops_per_second / memory_bandwidth` ≈ 295

Conclusion: - If `computation_intensity > accelerator_intensity`, it's **compute-limited** (good). - If `computation_intensity < accelerator_intensity`, it's **memory-limited** (bad).

For matrix multiplication, it's compute-limited if `B > 295` on an H100.

Extreme Case (B=1): Matrix-vector product. - Arithmetic intensity: 1 - This is **memory-limited** (bad) because you read a D x F matrix to perform only 2DF FLOPs. - This is essentially what happens with **generation** (token by token processing).

[01:42:19] Arithmetic Intensity of Inference

Naive Sampling: - To generate each token, feed the entire history (prompt + generated tokens) into the Transformer. - **Complexity**: Generating `T` tokens requires $O(T^3)$ FLOPs (since each feedforward pass is $O(T^2)$). - **Observation**: A lot of the work (encoding prefixes) is redundant.

Solution: Store KV cache in HBM.

Sampling with KV Cache: 1. **Prefill**: Given a prompt, encode it into vectors. This stage is parallelizable, like training. - Compute-bound (good). 2. **Generation**: Generate new response tokens sequentially. - Memory-bound (problematic).

Let's compute FLOPs and memory IO for MLP and attention layers separately. - `S`: Number of tokens we're conditioning on (prompt length). - `T`: Number of tokens we're generating. - For prefill: `T = S`. - For generation: `T = 1`.

[01:42:19] MLP Layers (focusing on matrix multiplications)

FLOPs: `6 * B * T * D` **Bytes Transferred**: `4 * B * T * D + 6 * D * F`

Arithmetic Intensity (simplified): `B * T`

For the two stages: - **Prefill (T=S)**: Intensity is `B * S`. This is easy to make compute-limited (good) by making `B * S` large enough (e.g., a long prompt or large batch size). - **Generation (T=1)**: Intensity is `B`. This means `B` (number of concurrent requests) must be large enough to make it compute-limited.

Why is this important? - In MLP layers, every sequence hits the same MLP weights (`Wup`, `Wgate`, `Wdown`). So batching helps!

[02:11:00] Attention Layers (focusing on matrix multiplications with FlashAttention)

FLOPs: `4 * B * S * T * D` **Bytes Transferred**: `4 * B * S * D + 4 * B * T * D`

Arithmetic Intensity (simplified): `S * T / (S + T)`

For the two stages: - **Prefill (T=S)**: Intensity is `S/2`. This is good, as long as `S` (prompt length) is large enough. - **Generation (T=1)**: Intensity is `S / (S + 1)`. This is approximately 1, which is **bad**!

Why is this bad? - Unlike MLPs, batching doesn't help for attention layers during generation. - In MLP layers, every sequence hits the same MLP weights. - In attention layers, every sequence has its own KV cache (Q, K, V), and these depend on `B`. - Mathematically, the `B` term cancels out in the arithmetic intensity for attention layers.

Summary: - **Prefill** is compute-limited, generation is memory-limited. - **MLP intensity** is `B` (requires concurrent requests). - **Attention intensity** is `S * T / (S + T)`. During generation (`T=1`), this becomes `S / (S+1)` (approximately 1), which is impossible to improve by batching.

[02:40:00] Throughput and Latency

Since inference is memory-limited, we can compute the theoretical maximum latency and throughput of a single request. We assume perfect overlap between compute and communication, ignoring overheads.

Memory Usage: - `KV_cache_size` = `S * (K * L * H * 2 * 2)` (for sequence length S, K heads, L layers, H dim, 2 for key/value, 2 for BF16) - `total_memory_usage` = `B * KV_cache_size + parameter_size`

Latency: `memory / memory_bandwidth` (time to read all parameters and KV cache for each step) **Throughput**: `B / latency` (generating `B` tokens in parallel)

Llama 2 13B Config on H100: - `S`: 1024, `D`: 5120, `F`: 13824, `N`: 40, `K`: 40, `H`: 128, `L`: 40, `V`: 32000, `memory_bandwidth`: 3.35×10^{12}

Results for different batch sizes (B): - **B=1**: - `memory`: 26.8 GB - `latency`: 8.08 ms - `throughput`: 124.6 tokens/sec - **B=64**: - `memory`: 79.7 GB - `latency`: 23.79 ms (worse) - `throughput`: 2689.4 tokens/sec (better) - **B=256**: - `memory`: 240.7 GB (doesn't fit in H100's 80GB memory) - `latency`: 70.07 ms (much worse) - `throughput`: 3561.7 tokens/sec (better, but diminishing returns)

Trade-off between latency and throughput: 1. Smaller batch sizes yield better latency but worse throughput. 2. Larger batch sizes yield better throughput but worse latency.

Easy Parallelism: Launch `M` copies of the model. Latency is the same, throughput increases by `M`. **Harder Parallelism**: Shard the model and the KV cache (more details in the "Scaling book chapter on Transformers").

Time-to-First-Token (TTFT): - Essentially a function of prefill. - Use smaller batch sizes during prefill for faster TTFT. - Use larger batch sizes during generation to improve throughput.

[03:28:00] Taking Shortcuts (Lossy)

The KV cache size is the bottleneck. We need to reduce it without hurting accuracy.

[03:30:00] Grouped-Query Attention (GQA)

Multi-head attention (MHA): Each query head has its own key and value heads. **Multi-query attention (MQA)**: All query heads share a single key and value head. (Not very expressive).

Grouped-query attention (GQA): Somewhere in between. `N` query heads, but only `K` key and value heads, interacting with `N/K` query heads.

Latency/throughput improvements: GQA significantly reduces time per sample compared to MHA, especially for larger numbers of GQA groups.

Rigorous Evaluation (Llama 2 13B with GQA): - Original Llama 2 13B: `K=40` (query heads), `B=64` (batch size) - `memory`: 79.7 GB - `throughput`: 2689 tokens/sec - GQA with 1:5 ratio (`K=8`, `B=64`): - `memory`: 33.4 GB (reduced) - `throughput`: 6416 tokens/sec (increased) - GQA with 1:5 ratio (`K=8`, `B=256`): - `memory`: 130.6 GB (fits in memory now!) - `throughput`: 16584 tokens/sec (much increased)

Accuracy Check: GQA does not drop accuracy significantly (as shown in Ainslie+ 2023 paper). Llama 3 adopted GQA.

[03:43:00] Multi-Head Latent Attention (MLA)

MLA (DeepSeek-AI+ 2024) is another way to reduce KV cache size. - **Idea**: Instead of reducing the number of key/value heads (like GQA), project each key and value vector from `N*H` dimensions to `C` dimensions (where `C` is much smaller). - **DeepSeek v2**: Reduces `N*H` from 16384 to `C=512`. - **Winkle**: MLA is not compatible with RoPE, so they add additional dimensions for RoPE. - **Latency/throughput improvements**: Follows similarly from KV cache reduction. - **Accuracy Check**: MLA is better than GQA (DeepSeek-AI+ 2024).

[03:48:00] Local Attention (CLA)

Local attention (Beltagy+ 2020, Child+ 2019, Jiang+ 2023) restricts attention to a local context (e.g., a sliding window). - **Idea**: Only look at the local context, which is most relevant. - **Effective context**: Scales linearly with the number of layers. - **KV cache**: Independent of sequence length! (This is really good for long sequences). - **Problem**: This can still hurt accuracy because long-range dependencies are lost. - **Solution**: Interleave local attention with global attention (hybrid layers). Character.ai uses 1 global layer every 6 layers (in addition to CLA). - **Cross-layer attention (CLA)** (Brandon+ 2024): Idea is to share key-value across layers (just as GQA shares across heads). This empirically improves the Pareto frontier of accuracy and KV cache size.

Summary: - **Goal**: Reduce the KV cache size (since inference is memory-limited) without hurting accuracy. - **Methods**: Lower-dimensional KV cache (GQA, MLA), shared KV cache (CLA), local attention on some layers.

[03:59:00] Alternatives to the Transformer

The Transformer was not designed with inference efficiency in mind. Its autoregressive nature and full attention mechanism are fundamental memory-boundedness bottlenecks. Can we do better by going beyond the Transformer?

State-Space Models (SSMs) (presentation from CS229S): - **Idea**: From signal processing, model long-context sequences in a sub-quadratic time. - **S4** (Gu+ 2021): Based on classic state-space models (linear dynamical systems), good at synthetic long-context tasks. - **Weaknesses**: Bad at solving associative recall tasks important for language (that Transformers do well). - SSMs struggled with tasks like "A B3 C6 F1 E2 -> A? C? F? E? B?". - **Mamba** (Gu+ 2023): Allows SSM parameters to be input-dependent, matching Transformers at 1B scale. - **Jamba** (Lieber+ 2024): Interleaves Transformer-Mamba layers (1:7 ratio) with a 52B MoE. - **BASED** (Arora+ 2024): Uses linear attention + local attention. - **Linear attention**: Approximates attention kernel with Taylor expansion, making it linear in sequence length. - **Minimax-01**: Uses linear attention + full attention (456B parameter MoE). - **Takeaway**: Linear + local attention (still needs some full attention) yields serious SOTA models. Replacing $O(T)$ KV cache with $O(1)$ state is much more efficient for inference.

Diffusion Models: - **Popular for image generation**, but harder to get working for text generation. - **Idea**: Generate each token in parallel (not autoregressively), refine multiple time steps. - Start with random noise (over entire sequence), iteratively refine it. - **Results from Inception Labs**: Diffusion is much faster on coding benchmarks. - Mercury Coder Small/Mini are significantly faster than Transformer-based models like Gemini, Claude, GPT-4o, Jamba, etc. - **Overall**: Significant gains in inference to be made with more radical architecture changes.

[04:59:00] Quantization

Key idea: Reduce the precision of numbers. - Less memory means higher latency/throughput (since inference is memory-limited). - Also requires fewer FLOPs. - Of course, we have to worry about accuracy.

Comparing number formats: - **FP32** (4 bytes): Needed for parameters and optimizer states during training. - **BF16** (2 bytes): Default for inference. - **FP8** (1 byte): Less accurate but cheaper than FP16. - **INT8** (1 byte): Less accurate but cheaper than FP8. Range [-128, 127]. - **INT4** (0.5 bytes): Even less accurate. Range [-8, 7].

Quantization-aware training (QAT): Train with quantization, but doesn't scale up. **Post-training quantization (PTQ)**: Run on sample data to determine scale and zero point.

LLM.int8() (Dettmers+ 2022): Standard quantization (scale by absolute max). - You take an FP16 vector, figure out its dynamic range (max absolute value), and use that to scale and round values to INT8. - To dequantize, you reverse the process. - **Problem**: Outliers (which appear in larger networks) screw everything up. - **Solution**: Extract outliers and process them in FP16. The rest is processed in INT8. This works well but is 15-23% slower than FP16.

Activation-aware quantization (Lin+ 2023): - **Idea**: Select which weights (0.1-1%) to keep in high precision based on activations. - FP16 -> INT3 produces 4x lower memory, 3.2x speedup. - The general idea is to quantize most weights to INT3, but keep a small percentage of "salient" weights in FP16 (determined by activation patterns).

[05:09:00] Model Pruning

Key idea: Just rip out parts of an expensive model to make it cheaper, and then fix it up.

NVIDIA Paper (Muralidharan+ 2024): 1. Identify important {layer, head, hidden dimension} on a small calibration dataset. 2. Remove unimportant layers to get a smaller model. 3. Distill the original model into the pruned model (repair it).

Results (Minintron): Pruning 15B parameter models to 8B with hardly any drop in MMLU score, and down to 4B with some drop.

Summary of taking shortcuts (lossy): Reduce inference complexity without hurting accuracy.

[05:15:00] Speculative Decoding / Speculative Sampling

Recall the two stages of inference: - **Prefill**: Given a sequence, encode tokens in parallel (compute-limited). Also gives probabilities. - **Generation**: Generate one token at a time (memory-limited). Checking is faster than generation.

Speculative Sampling (Leviathan+ 2022, Chen+ 2023): - Use a cheaper **draft model (P)** to guess a few tokens (e.g., 4). - Evaluate with **target model (Q)** (process tokens in parallel), and accept if it looks good.

How it works: - Use draft model (P) to generate `K` tokens autoregressively. - Compute logits for these `K` tokens under the target model (Q). - For each token, accept it with probability `min(1, Q(x_i)/P(x_i))`. - If accepted, move to the next token. If rejected, sample from `Q(x_i) –`

`P(x_i)` (corrected distribution) and stop. - **Key property**: Guaranteed to be an exact sample from the target model.

In practice: - Target model (e.g., 70B parameters), draft model (e.g., 8B parameters). - You want to make the draft model as close to the target as possible (distillation helps). - **Extensions**: Medusa (draft model generates multiple tokens in parallel), EAGLE (draft model takes high-level features from target model).

Summary: Exact sampling from target model (thanks to math!). Exploits asymmetry between checking and generation. Lots of room for innovation on the draft model (involves training).

[05:28:00] Handling Dynamic Workloads

Batching over sequences in live traffic is tricky because: 1. Requests arrive at different times (waiting for batch is bad for early requests). 2. Sequences have shared prefixes (e.g., system prompts, generating multiple samples). 3. Sequences have different lengths (padding is inefficient).

Continuous Batching (Orca: A Distributed Serving System for Transformer-Based Generative Models): - **Problem**: Training gets a dense block of tokens (batch size x sequence length). Inference has requests arrive and finish at different times, leading to a ragged array. - **Solution**: Iteration-level scheduling. - Decode step by step. - Add new requests to the batch as they arrive (don't have to wait until generation). - This means the worker generating tokens needs to hand control back to the scheduler every step. - You generate a token, come back to the scheduler, and if there's new requests, they get stuck in, and then it continues. - You're not wasting any time waiting around for requests.

Selective Batching: - **Problem**: Batching only works when all sequences have the same dimensionality. - **Solution**: Training: when all sequences of the same length, operate on a B x S x H tensor. But we might have different lengths. - **Attention computation**: Process each sequence separately. - **Non-attention computation**: Concatenate all the sequences together to `[3 + 9 + 5, H]`.

PagedAttention (Kwon+ 2023): - Paper that introduced vLLM. - **Previous status quo**: Request comes in. Allocate section of KV cache for prompt and response (up to a max length). - **Problem**: Fragmentation (what happens to your hard drive). This is wasteful since we might generate fewer tokens (internal fragmentation). Might be extra unused space between sections (external fragmentation). - **Solution**: PagedAttention (remember operating systems). Divide the KV cache

of a sequence into non-contiguous blocks. - **How it works**: Divide KV cache into blocks. Put them wherever there's space. If you have two requests, they might be scattered across memory. If you need to diverge, you copy and reduce the reference count.

[05:40:00] Summary

- **Inference is important** (actual use, evaluation, reinforcement learning).
- **Different characteristics compared to training** (memory-limited, dynamic).
- **Techniques**: New architectures, quantization, pruning/distillation, speculative decoding.
- **Ideas from systems** (speculative execution, paging).
- **New architectures have huge potential for improvement**.

Practical Takeaways

- Inference efficiency is paramount for deploying large language models, impacting user experience and operational costs.
- The sequential nature of token generation makes inference memory-bound, especially for attention layers.
- Optimizing inference involves a multi-pronged approach: architectural innovations, quantization, pruning, and systems-level scheduling.
- Understanding the trade-offs between latency and throughput is crucial for designing efficient inference systems.
- Novel architectures like GQA, MLA, local attention, state-space models, and diffusion models offer significant potential for improving inference speed and memory usage.
- Quantization and pruning are effective for reducing model size and speeding up inference, but require careful handling to preserve accuracy.
- Speculative decoding and continuous batching/PagedAttention are powerful techniques that leverage the asymmetry between checking and generation, and systems-level memory management, respectively.

Open Questions / Things to Remember

- How to balance the trade-off between latency and throughput for different application requirements?

- What are the practical limits of quantization and pruning before accuracy degrades unacceptably?
- How can we further bridge the gap between architectural innovations and their practical deployment in real-world inference systems?
- Will diffusion models or state-space models eventually replace transformers for text generation due to their inference advantages?
- How to effectively manage the dynamic and heterogeneous nature of live inference traffic to maximize GPU utilization and minimize latency?
- The "inference game" is not just about optimizing existing models but also about designing models that are inherently efficient for inference.

← Lec 09

Lec 11 →

Lecture 11: Scaling Laws 2

Stanford CS336 · Spring 2025 · 1:18:13 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



These are comprehensive notes from the lecture.

TL;DR

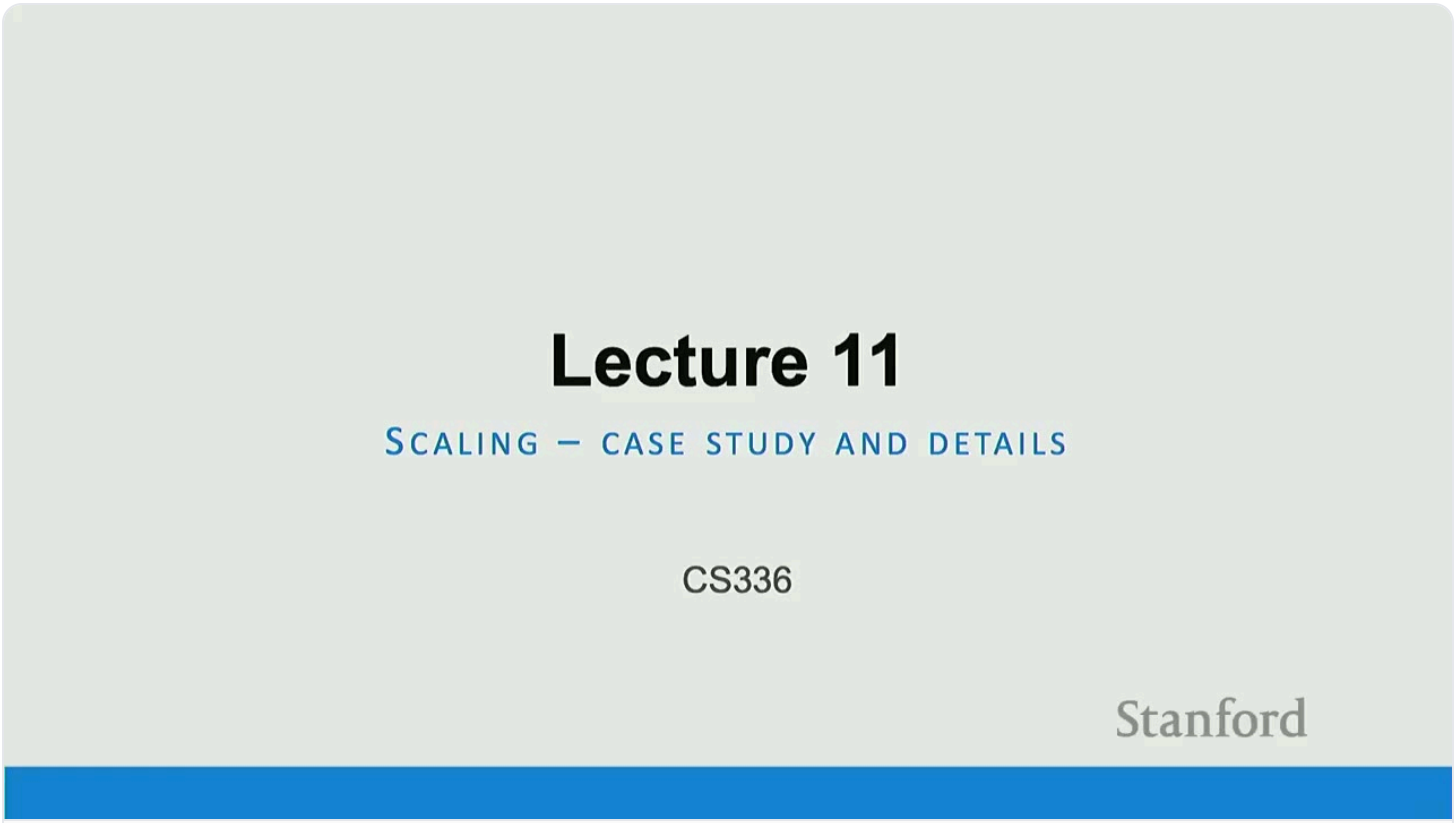
- **Scaling Laws in Practice:** Modern large language model (LLM) builders use scaling laws as a core part of their design process, but the details of these studies are often kept proprietary.
- **MUP for Hyperparameter Stability:** The Maximum Update Parameterization (MUP) aims to keep optimal hyperparameters (like learning rates) stable across different model scales, simplifying the search process.

- **WSD Learning Schedules:** Warm-up Stable Decay (WSD) learning rate schedules, often trapezoidal, allow for efficient data scaling analysis by enabling the reuse of stable training phases for different data budgets.
- **Chinchilla Replications:** Several recent works (Llama 3, Hunyuan-1, Minimax-01) replicate Chinchilla-style scaling laws, often finding different optimal token-to-parameter ratios, suggesting that the original 20:1 ratio is not a universal constant.
- **Challenges in Scaling:** Key challenges in scaling LLMs include setting model architecture hyperparameters (width, depth), optimizing hyperparameters (learning rate, batch size), and managing the compute needed for extensive scaling law sweeps.

Key Concepts

- MUP (Maximum Update Parameterization)
- WSD (Warm-up Stable Decay) Learning Schedules
- Chinchilla Scaling Laws
- IsosFlop Curves
- Data-to-model Ratio
- Hyperparameter Stability
- Activation Norms
- Weight Decay
- Exotic Optimizers

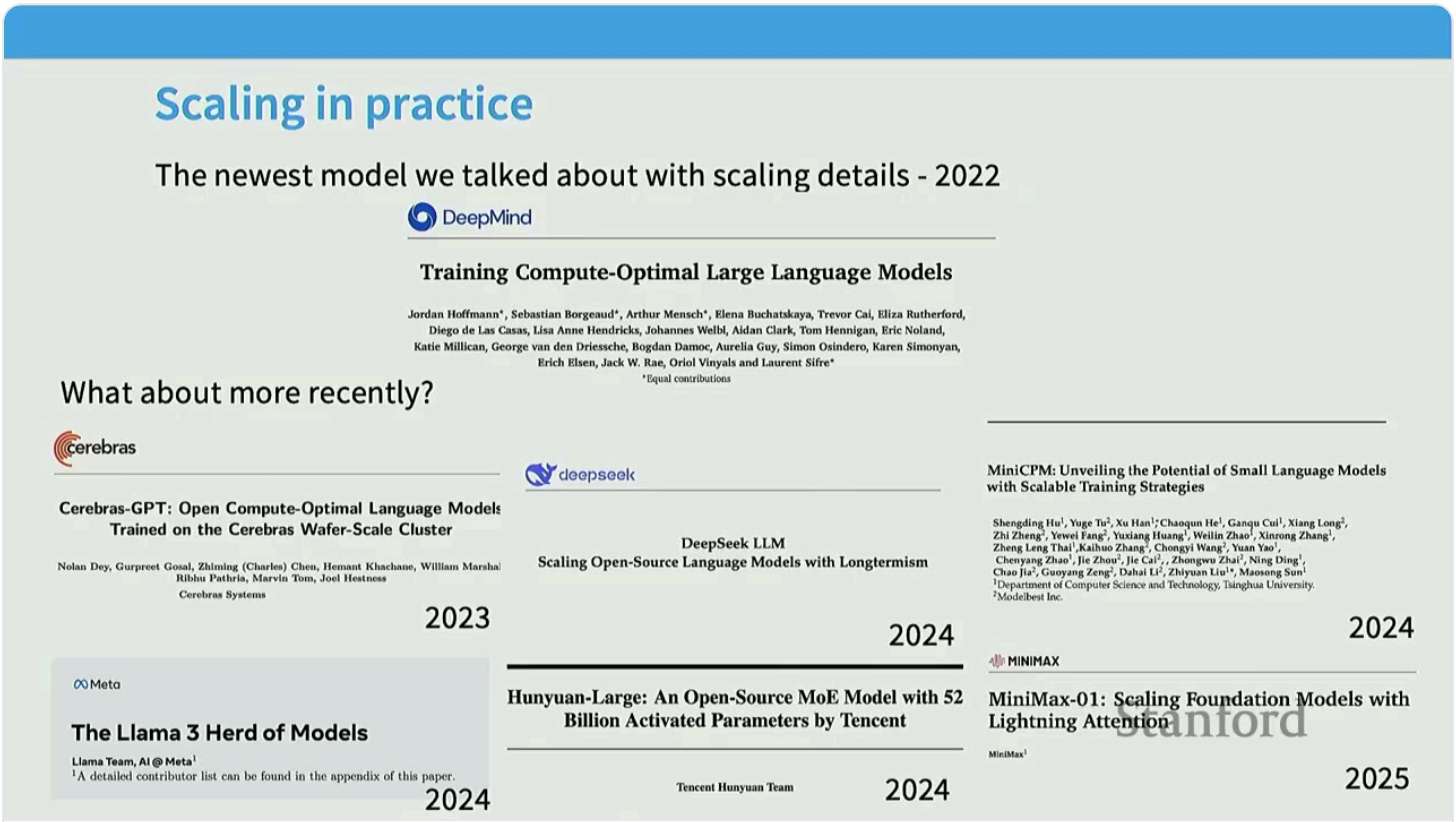
[0:00] Introduction: Scaling Laws in Practice



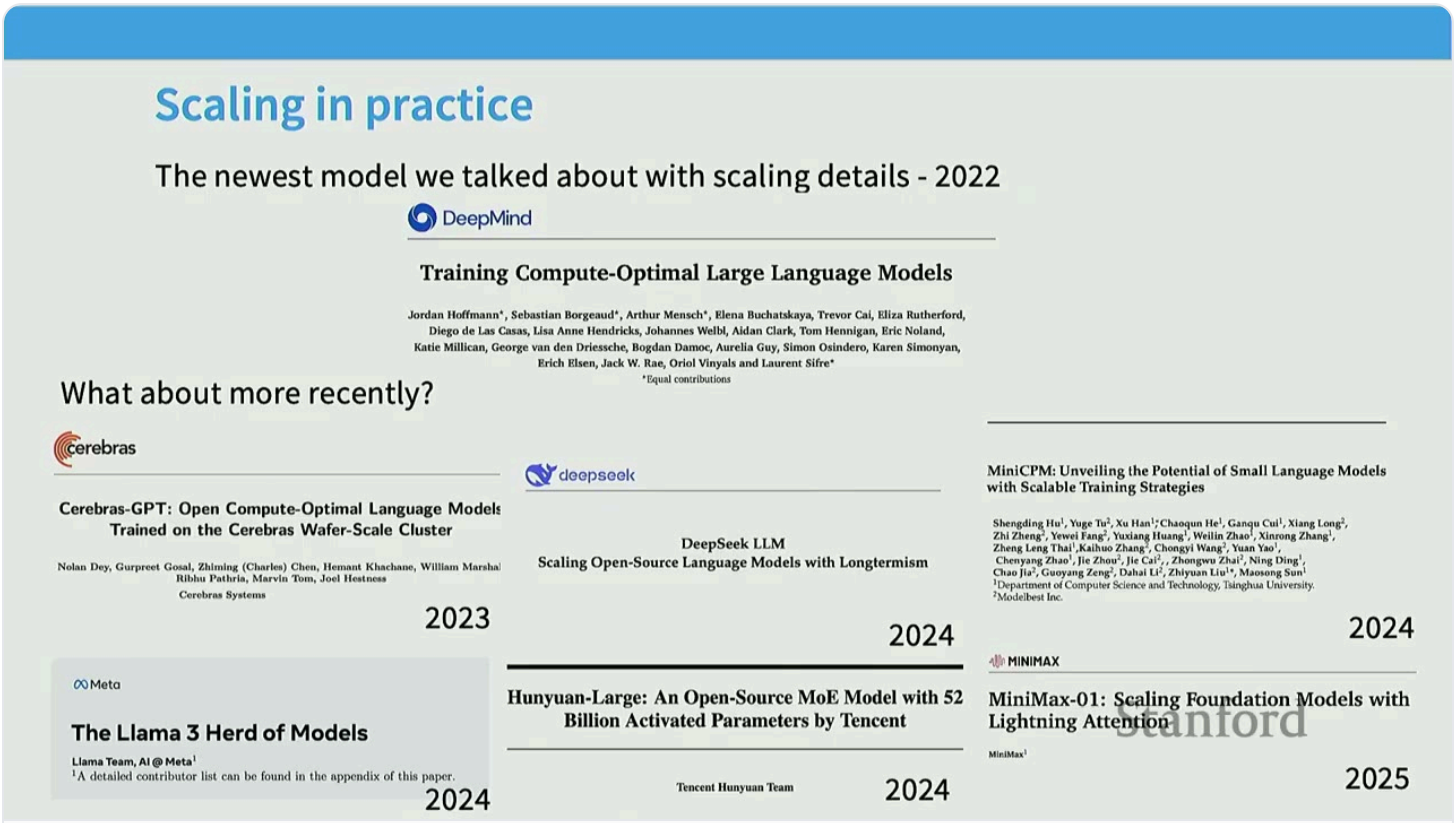
0:21 Lecture 11 title slide: Scaling - Case Study and Details, for Stanford CS336.



0:39 A male lecturer stands at a podium, looking at his laptop.



2:05 Slide titled "Scaling in practice" showing recent language models and their scaling details.



2:15 Slide titled "Scaling in practice" showing recent language models and their scaling details.

Today's lecture is the second and last on scaling laws. It will be a case study and details-oriented lecture, covering two main topics: 1. **Case Studies:** Examining how modern LLM builders use scaling laws in their design process, looking at specific papers. 2. **MUP Deep Dive:** A detailed look into the MUP parameterization method.

The motivation is to understand the best practices for scaling large language models, specifically how to choose hyperparameters and architecture effectively. While Chinchilla introduced scaling laws, there's a healthy skepticism about how robust these findings are in practice.

After the DeepMind Chinchilla paper in 2022, the landscape of LLM development became very secretive, with companies stopping publications on data and scaling. However, some competently executed large-scale models have still shared their scaling insights.

Recent Models with Scaling Insights: - **Cerebras-GPT (2023):** Open Compute-Optimal Language Models - **DeepSeek LLM (2023):** Scaling Open-Source Language Models with Longtermism - **MiniCPM (2024):** Unveiling the Potential of Small Language Models with Scalable Training Strategies - **Llama 3 (2024):** The Llama 3 Herd of Models - **Hunyuan-Large (2024):** An Open-Source MoE Model with 100 Billion Activated Parameters by Tencent Team - **Minimax-01 (2024):** Scaling Foundation Models with Longtermism

The newer models (Llama 3, Hunyuan-Large, Minimax-01) have sparser scaling law insights compared to DeepSeek and MiniCPM, which are considered gold standards for detailed scaling studies.

[3:22] MUP: Maximum Update Parameterization

MUP is an approach designed to address the challenge of hyperparameter tuning when scaling models.

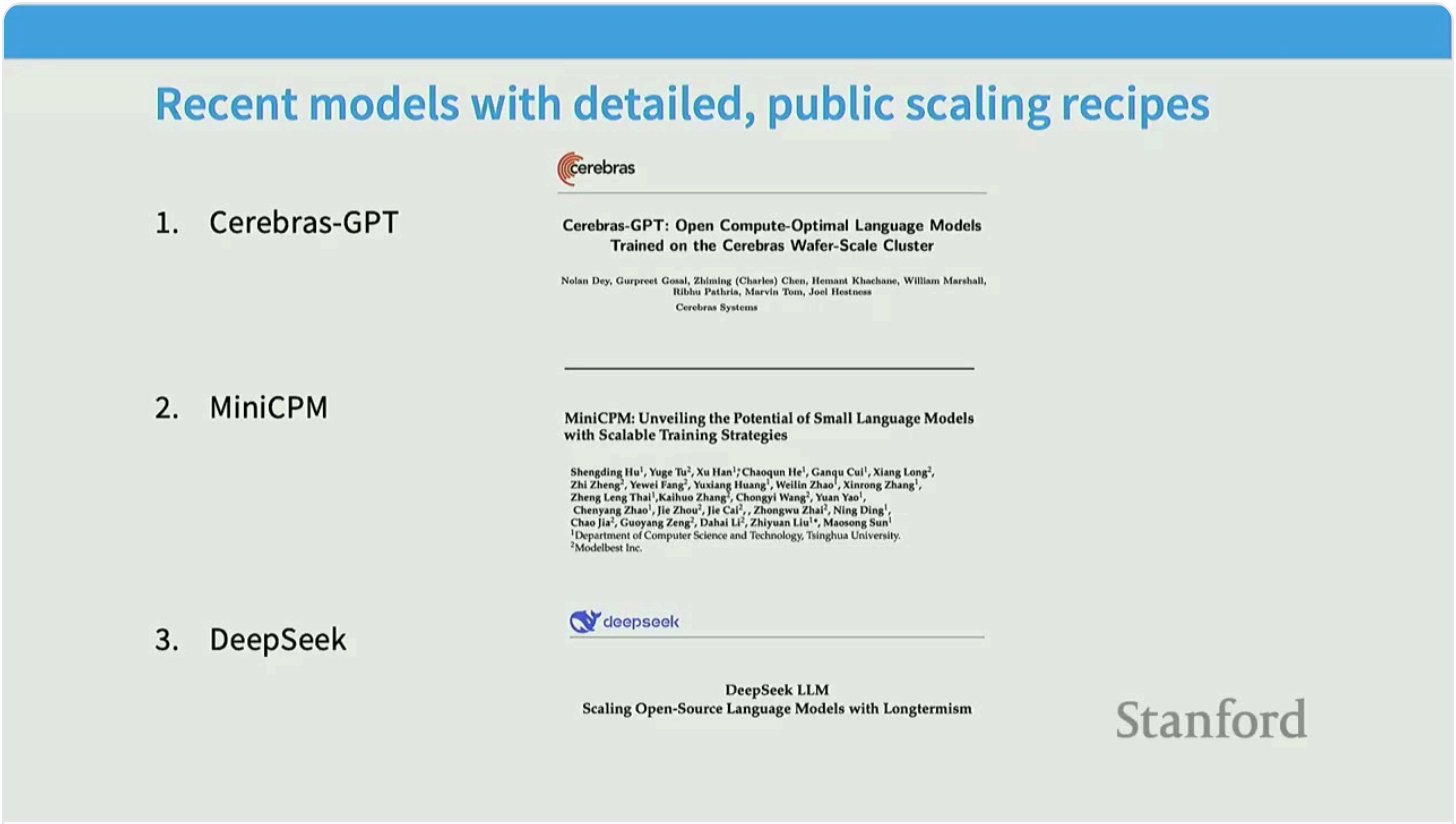
Problem: When training models, as they get larger (e.g., wider MLPs), the optimal learning rate often shifts downwards. This means hyperparameters need to be re-tuned for larger models, which is computationally expensive.

MUP Solution: If models could be parameterized differently such that the optimal learning rate (and other hyperparameters) remains stable across scales, it would greatly simplify the design process. MUP aims to achieve this by ensuring scale-invariance in hyperparameters.

[5:07] Recent Models and Scaling Recipes

The first part of the lecture focuses on three models with detailed scaling recipes: CerebrasGPT, MiniCPM, and DeepSeek.

[5:41] CerebrasGPT: MUP for Stable Scaling



5:22 Slide listing "Recent models with detailed, public scaling recipes" including Cerebras-GPT, MiniCPM, and DeepSeek.

muP parametrization

Appendix contains a very clear set of differences in parametrizing the model for scaling

	Standard Parameterization (SP)	Maximal Update (μP)
Variables		
W	A multiplicative or fully-connected weights tensor	
b	A bias weights tensor	
X, Y	Activation tensors: layer input, output, respectively	
$d_{\text{model,base}}$	Proxy (base) model's layer width	
d_{model}	Width of each layer	
d_{head}	Size of each attention head	
embed	Combined token + position embedding function	
η_{base}	The base learning rate (LR): Maximum in training schedule	
σ_{base}	The base initialization standard deviation	
m_{width}	—	Layer width multiplier: $d_{\text{model}}/d_{\text{model,base}}$
m_{emb}	—	Emdedding output multiplier

	Standard Parameterization (SP)	Maximal Update (μP)
Empirically Tuned Values		
$d_{\text{model,base}}$	—	256
η_{base}	Must tune for each model size	6e-3
σ_{base}	0.02	0.08
m_{emb}	—	10.0
Formulas		
Embedding initializer	$W_{\text{emb}} \sim N_{\text{trunc}}(0, \sigma^2_{\text{base}})$	
Embedding LR	$\eta_{\text{emb}} = \eta_{\text{base}}$	
Embedding output	$Y_{\text{emb}} = \text{embed}(X)$	$Y_{\text{emb}} = m_{\text{emb}} \cdot \text{embed}(X)$
LN initializer	$W_\gamma \sim 1, b_\beta \sim 0$	
LN LR	$\eta_{\text{LN}} = \eta_{\text{base}}$	
Bias initializer	$b \sim 0$	
Bias LR	$\eta_b = \eta_{\text{base}}$	
MHA equation	$\text{softmax}\left(\frac{Q^T K}{\sqrt{d_{\text{head}}}}\right) V$	$\text{softmax}\left(\frac{Q^T K}{d_{\text{head}}}\right) V$
QKV weights initializer	$W_{\text{qkv}} \sim N_{\text{trunc}}(0, \sigma^2_{\text{base}})$	$W_{\text{qkv}} \sim N_{\text{trunc}}(0, \sigma^2_{\text{base}}/m_{\text{width}})$
QKV weights LR	$\eta_{\text{qkv}} = \eta_{\text{base}}$	$\eta_{\text{qkv}} = \eta_{\text{base}}/m_{\text{width}}$
O weights initializer	$W_o \sim N_{\text{trunc}}(0, \frac{\sigma^2_{\text{base}}}{2 \cdot n_{\text{layers}}})$	$W_o \sim N_{\text{trunc}}(0, \frac{\sigma^2_{\text{base}}}{2 m_{\text{width}} \cdot n_{\text{layers}}})$
O weights LR	$\eta_o = \eta_{\text{base}}$	$\eta_o = \eta_{\text{base}}/m_{\text{width}}$
ffn1 weights initializer	$W_{\text{ffn1}} \sim N_{\text{trunc}}(0, \sigma^2_{\text{base}})$	$W_{\text{ffn1}} \sim N_{\text{trunc}}(0, \sigma^2_{\text{base}}/m_{\text{width}})$
ffn1 weights LR	$\eta_{\text{ffn1}} = \eta_{\text{base}}$	$\eta_{\text{ffn1}} = \eta_{\text{base}}/m_{\text{width}}$
ffn2 weights initializer	$W_{\text{ffn2}} \sim N_{\text{trunc}}(0, \frac{\sigma^2_{\text{base}}}{2 \cdot n_{\text{layers}}})$	$W_{\text{ffn2}} \sim N_{\text{trunc}}(0, \frac{\sigma^2_{\text{base}}}{2 m_{\text{width}} \cdot n_{\text{layers}}})$
ffn2 weights LR	$\eta_{\text{ffn2}} = \eta_{\text{base}}$	$\eta_{\text{ffn2}} = \eta_{\text{base}}/m_{\text{width}}$
Output logits multiplier	$Y_{\text{logits}} = W_{unemb} X$	$Y_{\text{logits}} = W_{unemb} X / m_{\text{width}}$

Stanford

9:01 Slide on "mUP parametrization" showing tables of empirically tuned values and formulas for scaling.

MiniCPM

MiniCPM (2024) – new small, high-perf LM from Tsinghua group.

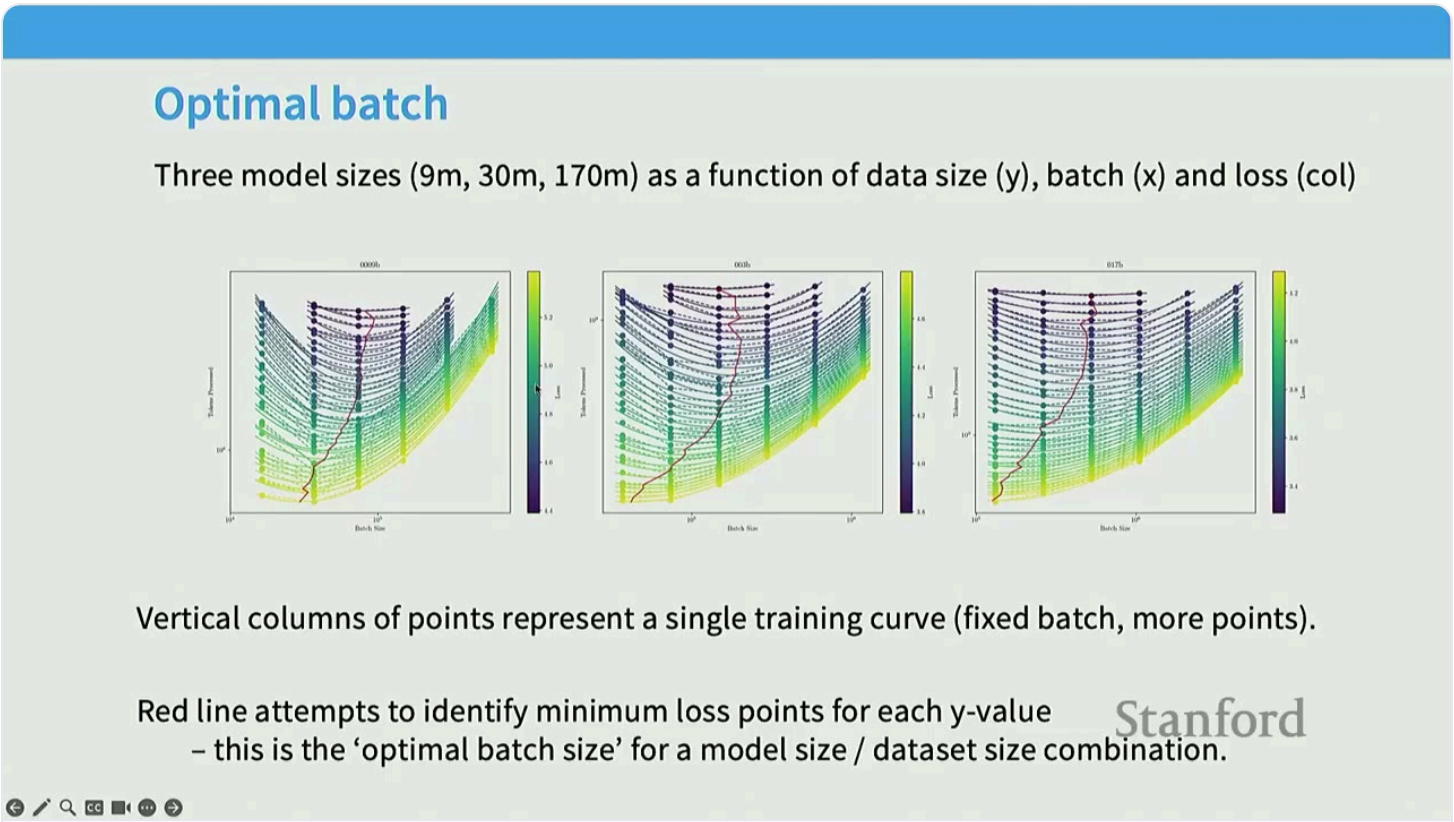
MiniCPM: Unveiling the Potential of Small Language Models with Scalable Training Strategies

Shengding Hu[1], Yuge Tu[2], Xu Han[1], Chaoqun He[1], Ganqu Cui[1], Xiang Long[2], Zhi Zheng[2], Yewei Fang[2], Yuxiang Huang[1], Weilin Zhao[1], Xinrong Zhang[1], Zheng Leng Thai[1],Kaihuo Zhang[2], Chongyi Wang[2], Yuan Yao[1], Chenyang Zhao[1], Jie Zhou[2], Jie Cai[2], , Zhongwu Zhai[2], Ning Ding[1], Chao Jia[2], Guoyang Zeng[2], Dahai Li[2], Zhiyuan Liu[1]*, Maosong Sun[1]
[1]Department of Computer Science and Technology, Tsinghua University.
[2]Modelbest Inc.

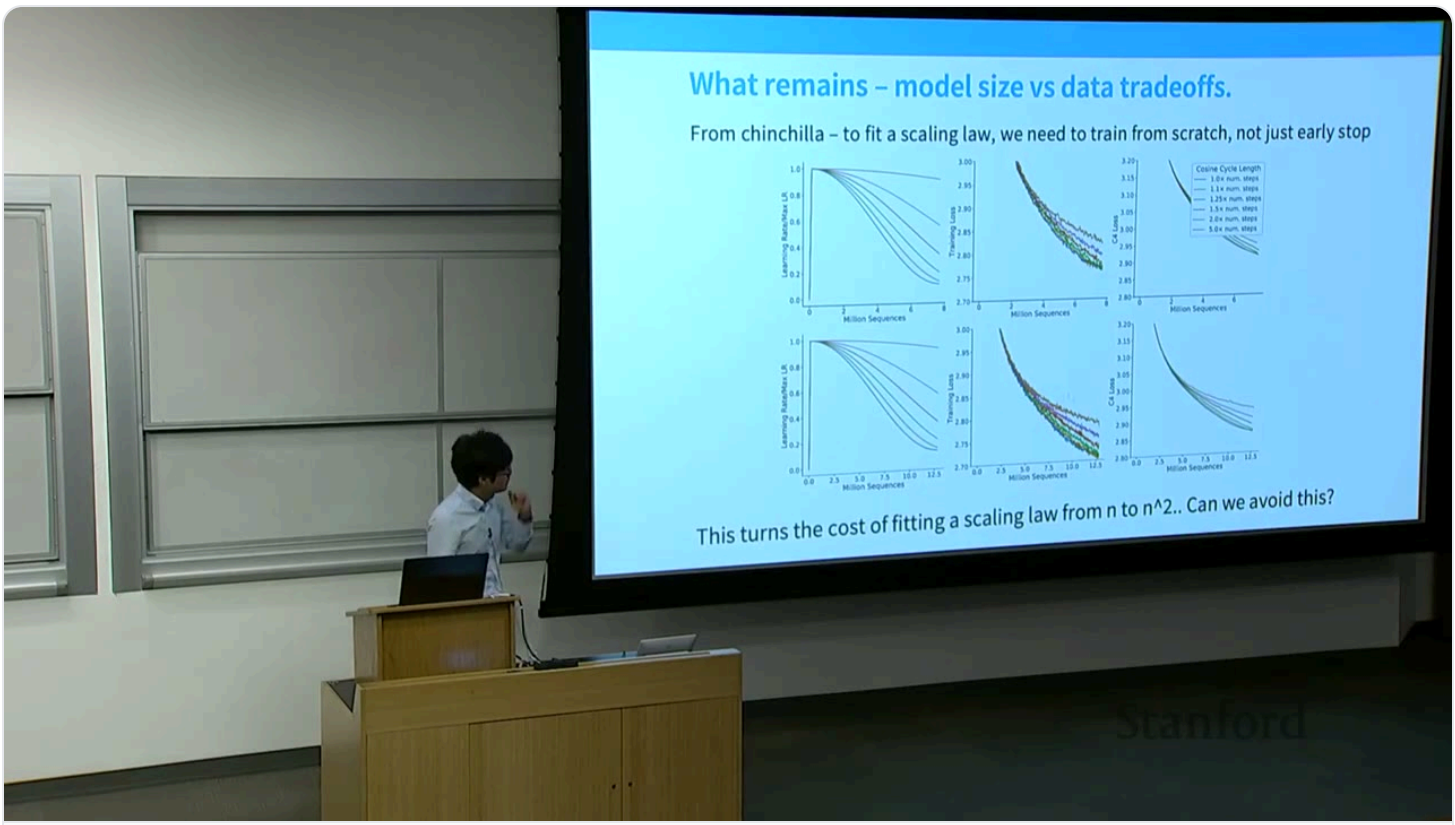
Careful, extensive scaling computations + muP to stabilize and simplify scaling

Stanford

10:47 Slide introducing "MiniCPM (2024)" from Tsinghua group, emphasizing scaling computations and mUP.



15:30 Slide on "Optimal batch" showing three plots of model sizes as a function of data size, batch, and loss.



19:58 A lecturer points at a screen displaying graphs about model size vs. data tradeoffs.

(partial) solution in miniCPM – WSD learning rate

Instead of cosine, split learning rate into warmup, stable, and decay phases.

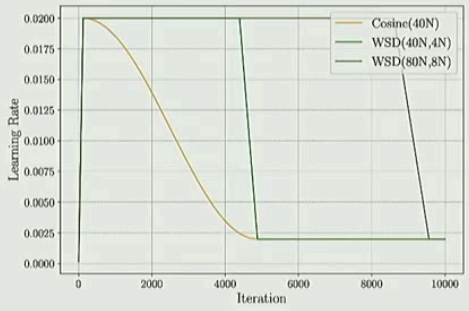


Figure 15: Illustrative comparison between Cosine LRS and WSD LRS. The WSD LRS with different end steps share the same stable training stage.

In this section, we introduce the utilization of the WSD scheduler as an effective approach to explore the scaling law with linear cost ($O(mC)$). Since WSD scheduler has the advantage of arriving at the optimal loss of Cosine LRS after decaying from stable stage's checkpoints of any step, we are now able to precisely measure the optimal scaling properties without re-training the models from scratch to different amount of tokens, thus making the scaling law measurement much more efficient along the data axis.

We measure the scaling law along the data and model axes by training SLMs of 6 sizes ranging from 0.04B to 2B, each with 6 decayed model starting from checkpoint of $10N$ to $60N$ data during the stable training stage. The final loss is evaluated on five held-out evaluation dataset. To potentially compare the loss when the model uses different tokenizer, we take the average of loss by number of bytes instead of number of tokens, following Achiam et al. (2023). The final loss of each pair of data size and model size is shown in the blue lines in Figure 17.

Stanford

For chinchilla-style analysis, can restart the run at the end of the stable phase.

20:44 Slide on "WSD learning rate" in miniCPM, showing a graph comparing Cosine and WSD LRS.

(partial) solution in miniCPM – WSD learning rate

Instead of cosine, split learning rate into warmup, stable, and decay phases.

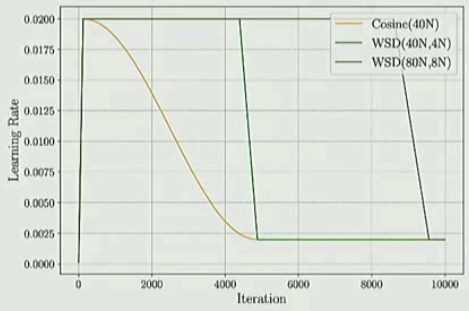


Figure 15: Illustrative comparison between Cosine LRS and WSD LRS. The WSD LRS with different end steps share the same stable training stage.

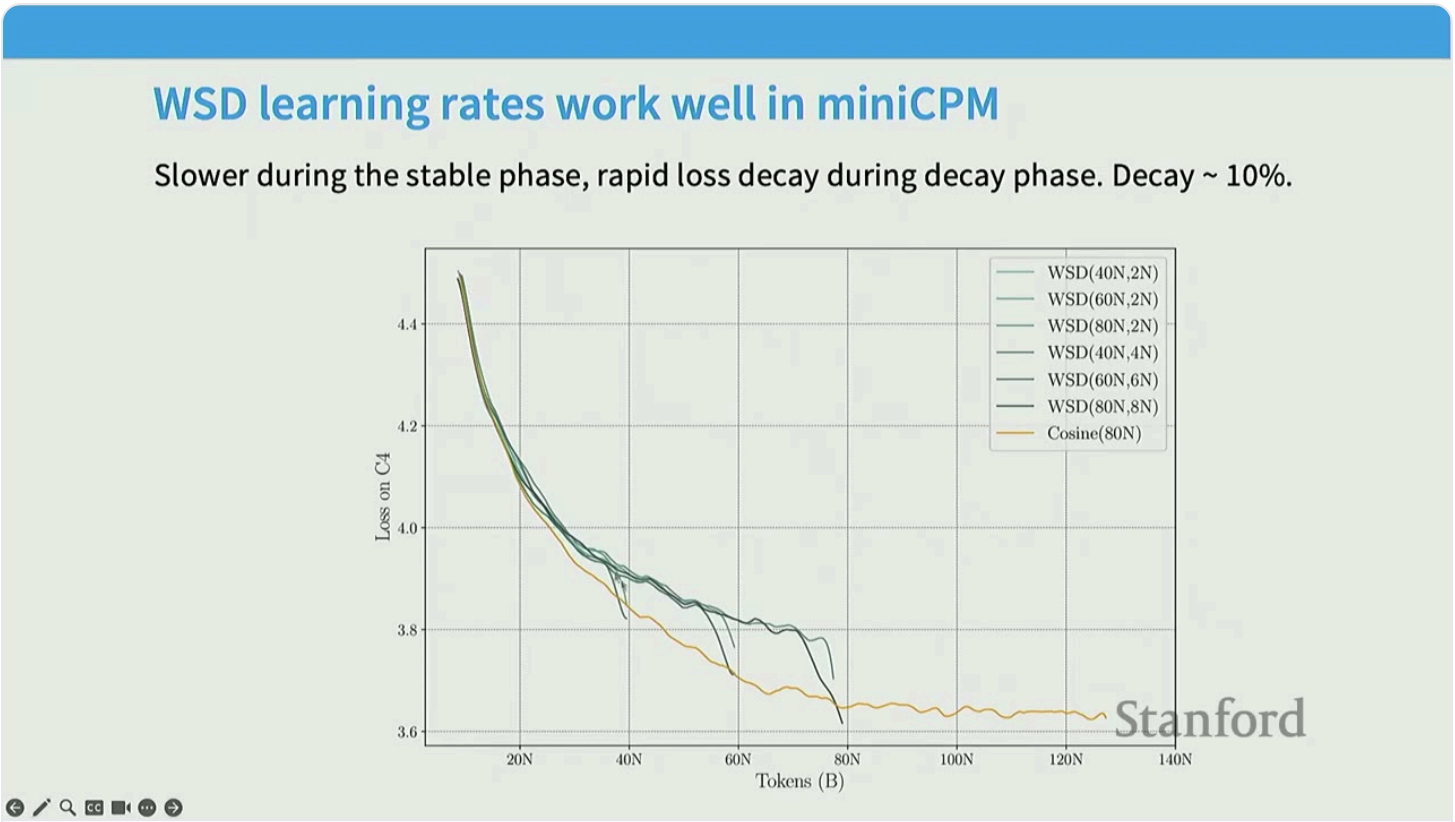
In this section, we introduce the utilization of the WSD scheduler as an effective approach to explore the scaling law with linear cost ($O(mC)$). Since WSD scheduler has the advantage of arriving at the optimal loss of Cosine LRS after decaying from stable stage's checkpoints of any step, we are now able to precisely measure the optimal scaling properties without re-training the models from scratch to different amount of tokens, thus making the scaling law measurement much more efficient along the data axis.

We measure the scaling law along the data and model axes by training SLMs of 6 sizes ranging from 0.04B to 2B, each with 6 decayed model starting from checkpoint of $10N$ to $60N$ data during the stable training stage. The final loss is evaluated on five held-out evaluation dataset. To potentially compare the loss when the model uses different tokenizer, we take the average of loss by number of bytes instead of number of tokens, following Achiam et al. (2023). The final loss of each pair of data size and model size is shown in the blue lines in Figure 17.

Stanford

For chinchilla-style analysis, can restart the run at the end of the stable phase.

21:22 Slide on "WSD learning rate" in miniCPM, showing a graph comparing Cosine and WSD LRS.



22:50 Slide showing a graph titled "WSD learning rates work well in miniCPM" with various WSD and Cosine curves.

CerebrasGPT trained a family of models (0.1 to 13 billion parameters) using the Chinchilla recipe. Their core finding is that using MUP makes scaling much more stable and pleasant to deal with.

Core Finding: MUP parameterization makes scaling more stable. - **Plot Analysis:** The plot shows test loss on the Pile dataset versus training FLOPs. - Blue line (Cerebras-GPT): Standard parameterization. - Orange line (Cerebras-GPT MUP): MUP parameterization. - MUP models (orange) show better scaling behavior, often outperforming or matching Pythia and GPT-J.

Hyperparameter Scaling Strategy: - CerebrasGPT authors found more predictable scaling from MUP parameterization. - **Standard Parameterization:** When using standard parameterization, there are significant oscillations around the predicted scaling law (dashed line), making it hard to achieve the expected performance. This is due to the need to adjust learning rates with scale. - **MUP Parameterization:** With MUP, the orange line (MUP scaling) is much closer to the predicted scaling law, indicating more stable and predictable scaling. - **Importance of MUP:** MUP is crucial because it allows hyperparameters (initialization, per-layer learning rates) to remain stable across scales, reducing the need for expensive re-tuning. This is a key practice in frontier labs (e.g., Llama 4 uses "MetaP," a MUP variant). - **MUP Parameterization Details:** The CerebrasGPT paper provides a detailed appendix (Table 2) outlining the differences between Standard

Parameterization (SP) and MUP. - **Key Difference:** In MUP, every non-embedding parameter is initialized with $1/\text{width}$, and per-layer learning rates are scaled down by $1/\text{width}$. This contrasts with SP, where initialization might use $1/\sqrt{\text{width}}$ and learning rates are global. - **Setting Empirical Values:** MUP is combined with aggressive scaling for hyperparameter search and μ Transfer to optimize hyperparameters. - They scale down experiments to 40 million parameters, perform extensive hyperparameter search on this proxy model, and then scale up using MUP to keep hyperparameters stable. - The plots (Figure 13) show validation loss for different hyperparameters. MUP helps keep hyperparameters generally stable. - This strategy (training small surrogate models and then scaling up) is also seen in MiniCPM and DeepSeek.

[1:00:00] MiniCPM: Unveiling the Potential of Small Language Models

MiniCPM (2024) is a new, high-performance LLM from Tsinghua group. - **Goal:** Train relatively small language models using a lot of compute to achieve high performance. - **Approach:** Careful scaling computations and MUP to stabilize and simplify scaling. - **Performance:** MiniCPM models (1.2B to 2.4B parameters) were remarkably good at the time of their release, beating most 2B models and matching many modern 7B models (Figure from MiniCPM paper). This indicates they did something right at the frontier.

Technique 1: MUP to Stabilize Scaling - MiniCPM uses the same MUP strategy as CerebrasGPT (Table 7). - **Key Idea:** Pick hyperparameters at a small scale, hope they stay stable, and then scale everything up. MUP helps ensure this stability. - **Similarities:** The scaling factors and parameters (e.g., `scale_emb`, `scale_depth`, `init_std`, `lr`) are very similar to CerebrasGPT.

Scaling Recipe / Strategy: - Use MUP for initialization, fix the aspect ratio, and scale up the overall model size. - **Model Parameters (Table from MiniCPM):** Shows how model dimensions (D_m, N_h, D_{ff}, N_l) are scaled. - **Compute Savings:** This approach offers roughly 5x compute savings by allowing efficient scaling from smaller pilot models to larger ones.

Optimal Batch Size: - MiniCPM also analyzes optimal batch sizes, similar to the Kaplan paper. - **Plot Analysis (Figure 12):** Shows training loss for three model sizes (9M, 30M, 170M) as a function of data size (y-axis), batch size (x-axis), and columns (col). - Vertical columns represent single training curves (fixed batch, more points). - Red lines identify minimum loss points for each y-value, representing the "optimal batch size" for a model size/dataset size combination. - **Critical Batch Size:** The critical batch size is the point of diminishing returns. As models get bigger and losses get lower, larger batch sizes can be used. - **Optimal Batch Size Plot (Figure 13):** Shows log(Batch Size) vs. Loss. - A fairly clean trend: polynomially increase the batch size as

loss decreases. - This allows predicting the optimal batch size for a given target loss (which can be derived from scaling laws).

Optimal Learning Rate (LR): - According to MUP, optimal learning rate should be (roughly) stable. - **Plot Analysis (Figure 3):** Shows loss vs. learning rate for different model sizes (0.06B to 2.1B). - The minimum loss (optimal learning rate) remains fixed across relatively large orders of magnitude (from small to big models, the minimum is at 10^{-2}). - **Validation:** This provides evidence that properly scaling model initialization and per-layer learning rates allows avoiding extensive learning rate tuning or fitting scaling laws on learning rates.

What Remains – Model Size vs. Data Tradeoffs: - To fit a Chinchilla scaling law, one needs to train from scratch, not just early stop. This turns the cost of fitting a scaling law from N to N^2. Can this be avoided? - **Problem with Cosine Learning Schedules:** If you use a cosine learning rate schedule, early checkpoints (less data) will have different optimal learning rates than later checkpoints (more data). This means you cannot simply use one long training run and extract checkpoints to analyze data scaling; you need to train separate models for each data budget.

Partial Solution in MiniCPM – WSD Learning Rate: - Instead of cosine, split learning rate into warm-up, stable, and decay phases. - **WSD Scheduler (Figure 15):** A trapezoidal learning rate schedule with a warm-up phase, a stable phase (flat learning rate), and a rapid decay phase. - **Advantage:** The stable phase can be reused. To analyze data scaling, you warm up, run the stable phase to the end, and then rewind checkpoints to perform different decay phases. This allows fitting Chinchilla-style scaling laws with mostly a single training run. - **WSD Learning Rates Work Well (Figure 16):** - Cosine learning rate schedules (yellow) show predictable, smooth decay. - WSD schedules (darker lines) show a slower decay during the stable phase, followed by a rapid drop during the decay phase (typically 10% of total steps). - At every token count, the minimum loss achieved by WSD schedules is comparable to or better than cosine schedules. - WSD provides the advantage of efficient data scaling analysis.

Side Note – Other Ways of Estimating Chinchilla Curves: - Gadre et al. propose other curve-fitting based ways of doing similar things. - **Core Idea:** The "penalty" from overtraining remains stable. This means you can extrapolate the excess token penalty from smaller training runs to estimate Chinchilla curves. - This approach hasn't been widely adopted for large-scale training.

Chinchilla-type Analysis: - Equipped with the WSD learning rate, MiniCPM can now try to find the optimal data-to-model size ratio. - They fit the losses with model size N and data size D (Hoffmann et al. 2022) using `scipy.curve_fit`. - They use method 1 (lower envelope) and method 3 (joint fit) from the Chinchilla paper.

Chinchilla Method 1 (Figure 17): - Fairly clear (though maybe not linear?) trends. - Different colors indicate different models. Their runs suggest relatively low diminishing returns due to data.

Chinchilla Method 3 (Figure 18): - Their primary scaling approach is the joint fit. They find very high data-model ratios. - **Tiny Models with Lots of Data:** The overall data-to-model ratio is very high (192), though they argue Llama architectures should have a higher ratio. - MiniCPM's optimal data-to-model ratio (D_{opt}/N_{opt}) is 192, compared to Hoffmann et al.'s (2022) 20. - Recent models like Llama 3 have significantly higher data-to-model ratios, suggesting that with careful optimization, we might be able to go far beyond the 20x model_size rule of thumb. - **Takeaway:** The Chinchilla analysis isn't a strong constraint; 20x model size is just a starting point. It's possible to significantly increase the token-to-parameter ratio.

Scaling Curve Fits are Generally Good (Figure 19): - Overall fits and predictions of models across a large range of sizes is fairly good. - X-axis: number of tokens in billions.

[1:31:00] DeepSeek: Another LLM with Careful Scaling Analysis

DeepSeek (2024) is another LLM with careful scaling analysis. - **Models:** 7B and 67B parameter models. - **Performance:** Generally high performance compared to other open LLMs. Roughly comparable to Llama 2 models of equivalent size (Table from DeepSeek paper). - **DeepSeek's Approach:** Very open about their experimental approach.

Scaling Strategy – Batch + LR: - DeepSeek does not use MUP. They directly estimate optimal batch size and learning rate. - **Method:** They compute a grid of loss for batch size and learning rate on small-scale experiments (177M FLOPs/token). - **Plot Analysis (Figure 2):** Shows loss landscapes for 1.17e17 FLOPs/token and 2.94e18 FLOPs/token. - The generalization error remains stable across a wide range of choices of batch sizes and learning rates. - This indicates that near-optimal performance can be achieved within a relatively wide parameter space. - **Learning Rate Scaling (Figure 3):** - They train a bunch of models with different non-embedding training FLOPs. - They vary batch size and learning rate across a grid. - They then plot optimal batch size vs. FLOPs and optimal learning rate vs. FLOPs. - **Batch Size Scaling Curve:** Shows a clear scaling law for optimal batch size. - **Learning Rate Scaling Curve:** Shows a scaling law for optimal learning rate, though the fit looks "a bit questionable" (as noted by the lecturer). - **Strategy:** Extrapolate these scaling laws to determine optimal batch size and learning rate for larger models.

Chinchilla Analysis: WSD-Style Learning Rate: - DeepSeek uses WSD-style learning rate (fast warm-up + two decay steps of 10% each). - **Plot Analysis (Figure 4):** Compares training loss for multi-step vs. cosine learning rate decay. - Generally seems to match performance of cosine

learning rates. - The advantage is that WSD allows for efficient Chinchilla-style analysis (data scaling) by reusing the stable phase and re-cooling down from checkpoints. - DeepSeek uses two decay steps (10% + 10%) for their WSD schedule.

Data-size Tradeoff Analysis: Chinchilla Method 2: - Straightforward isoflop-style analysis for selecting model size tradeoffs. - **Plot Analysis (Figure 5):** - **IsosFlop Curve:** Bits-per-byte on validation set vs. non-embedding FLOPs/token. Shows the characteristic U-shaped curves for different compute budgets. - **Optimal Model Scaling:** Non-embedding FLOPs/token vs. training FLOPs. Shows a linear relationship on a log-log plot. - **Optimal Data Scaling:** Tokens vs. training FLOPs. Shows a linear relationship on a log-log plot. - This is a replication of the Chinchilla result, providing a straightforward way to analyze token-to-model size tradeoffs. - DeepSeek's optimal token-to-parameter ratio is 192, which is significantly higher than the 20:1 ratio from Chinchilla. This is an outlier compared to most other literature. - **Takeaway:** The Chinchilla analysis is not a strong constraint. 20x model size is just a starting point. Recent models like Llama 3 have significantly higher data-to-model ratios.

Scaling Predicts Final Model Loss (Figure 6): - The fitted scaling models (generally) accurately predict the final model loss. - DeepSeek used small-scale experiments to predict the compute budget C and optimal generalization error for their 7B and 67B LLMs. - The results indicate that small-scale experiments can accurately predict well-trained models. They extrapolate from 10^{20} to 10^{24} FLOPs and accurately predict the performance of their 7B and 67B models.

[1:59:00] Llama 3 (2024) Scaling Laws

Llama 3 is a recent large model release with interesting scaling bits. - **IsosFlops-style Scaling (Figure 2, left):** They redo the IsosFlops-style scaling analysis. - They find an optimal ratio of 39:1 (token-to-parameter). This is higher than Chinchilla's 20:1. - This suggests that the 20:1 ratio isn't stable and can vary with architectural improvements, data quality, or other factors. - **Compute-to-downstream Scaling (Figure 2, right):** - They correlate compute (FLOPs) to normalized NLL per character. - They then correlate NLL per character to downstream accuracy (e.g., MMLU, Lambada). - This allows them to predict downstream performance based on compute, essentially mapping compute to accuracy. - They fit a sigmoid curve to relate NLL to accuracy, showing that they can accurately predict Llama 3 405B's performance based on smaller models.

[2:08:00] Hunyuan-1 (2024) Large Scaling Laws

Hunyuan-1 is another recent Chinese LLM. - **MoE Scaling:** They are training Mixture-of-Experts (MoE) models. - **IsosFlops-style Scaling (Figure 3):** They redo the Chinchilla-style analysis for MoE models. - They fit quadratics to training loss vs. activated parameters for different compute budgets. - They find an optimal ratio of 96:1 (data-to-active parameter). This is different from Chinchilla due to the MoE architecture. - **Takeaway:** Replications of Chinchilla are common because researchers want to understand how far they can push the token-to-parameter ratio, aiming for higher ratios (more data per parameter) for cheaper inference.

[2:10:00] Minimax-01 (2025) Large Scaling Laws

Minimax-01 is a linear-time or long-context LLM from another Chinese startup. - **Architecture Scaling + Chinchilla Method 1 (Figure 6):** - They compare Softmax Attention (quadratic), Lightning Attention (linear-time), and Hybrid-Lightning Attention. - They use Chinchilla Method 1 (lower envelope of loss curves) to analyze performance. - **Conclusion:** Their linear and hybrid models perform similarly to Softmax Attention, justifying their use for training long-context models. - This type of plot is common in papers introducing new efficient architectures (e.g., Mamba), showing that the new architecture performs comparably to standard attention as a function of compute.

[2:15:00] Recent Scaling Recipes (Recap)

CerebrasGPT: - Uses MUP to make hyperparameters invariant to scale. - Uses the Chinchilla scaling formula.

DeepSeek: - Assumes most transformer hyperparameters are invariant to scale. - Does a scaling analysis on batch size/LR to figure out optimal scaling. - Uses IsosFlop analysis to figure out model sizing. - Uses a piecewise-linear schedule to make Chinchilla scaling cheap.

MiniCPM Recipe: - Uses MUP to make transformer + LR invariant to scale. - Uses a piecewise linear schedule to get sample for Chinchilla method 3 (curve fitting).

Llama 3 / Hunyuan: - Just IsosFlops (no other scaling details). - Recent (late 2024+) but less detailed.

Minimax: - Architecture choice / decision scaling.

General Observations: - Learning rate and batch size are key concerns in scaling. - Fixing aspect ratio and scaling total model size is a common strategy.

[2:22:00] Validating and Understanding MUP

Goal: Scale-invariant hyperparameter tuning would be very useful. - **Plot Analysis (Figure from MUP paper):** Shows training loss vs. log(learning rate). - **Standard Practice (left):** Optimal learning rates shift as model width increases. - **Our Work (right):** With MUP, optimal learning rates remain stable across different model widths. - CerebrasGPT and MiniCPM also use MUP. Is it actually useful? - A preprint by Lucas Dax Lingle, "A Large-Scale Exploration of μ-Transfer," provides extensive ablations on MUP.

[2:29:00] What is MUP, Anyway?

MUP is based on the following assertions (from "A Spectral Condition for Feature Learning" by Yang et al.): - **A1:** The activations at initialization should remain $\Theta(1)$ (constant, not blowing up or vanishing), as a function of the width of the network n_l. - **A2:** After one gradient step, the change in activation should be $\Theta(1)$. - **Norm Note:** If individual activations are $\Theta(1)$, then the norm should be $\Theta(\sqrt{n_l})$.

[2:44:00] Deriving MUP (Condition A1)

Let's derive MUP from these two conditions using a simple example.

Setup: - Consider a deep, simple linear network: $h_l = W_l h_{l-1}$. - Initialize $W_l \sim \mathcal{N}(0, \sigma_l^2/n_{l-1} I_{n_l} \times n_{l-1})$. This is a Gaussian initialization with variance σ_l^2/n_{l-1}. - Basic matrix concentration implies that the operator norm of W_l concentrates around $\sigma_l\sqrt{n_l + n_{l-1}}$.

Goal: We want the norm of h_l at initialization to be $\Theta(\sqrt{n_l})$.

Derivation: 1. **Inductive Assumption:** Assume that for layer $l-1$, the norm of h_{l-1} is $\Theta(\sqrt{n_{l-1}})$. 2. **Inductive Case:** We need to show that the norm of h_l is $\Theta(\sqrt{n_l})$. - The norm of h_l is approximately $||W_l|| \cdot ||h_{l-1}||$. - Substitute the matrix concentration limit for $||W_l||$ and the inductive assumption for $||h_{l-1}||$. - So, $||h_l|| \approx \sigma_l\sqrt{n_l + n_{l-1}} \cdot \Theta(\sqrt{n_{l-1}})$. 3. **Choosing σ_l:** To make $||h_l|| = \Theta(\sqrt{n_l})$, we need to choose σ_l carefully. - Let's pick $\sigma_l = \frac{\sqrt{n_l}}{\sqrt{n_{l-1}}} \cdot \frac{1}{\min(1, \sqrt{n_l/n_{l-1}})}$.

- This simplifies to $\sigma_l = \frac{1}{\sqrt{n_{l-1}}}$ if $n_l \geq n_{l-1}$ (fan-out over fan-in). - This simplifies to $\sigma_l = \frac{1}{\sqrt{n_l}}$ if $n_l < n_{l-1}$ (fan-in over fan-out). 4. **Result:** With this choice of σ_l, plugging it back into the approximation for $\|h_l\|$, we get $\|h_l\| = \Theta(\sqrt{n_l})$, plus some lower-order terms.

Conclusion for Condition A1: For initializations, we should pick $\sigma_l = \frac{1}{\sqrt{n_{l-1}}}$ (plus a small correction factor) to ensure activations do not blow up at initialization. This is roughly $1/\sqrt{\text{fan-in}}$.

[2:59:00] Deriving MUP (Condition A2)

Now we deal with updates. The second part of MUP is about learning rates. - **Goal:** After one gradient step past initialization, the change in activation should be $\Theta(1)$. This means the update size shouldn't blow up or vanish. - **Update Rule:** For SGD on a linear layer, the update to weights ΔW_l is a rank-one outer product: $\Delta W_l = -\eta_l \nabla_l L \cdot h_{l-1}^T$. - **Change in Activation:** The change in activation at layer l is $\Delta h_l = \Delta W_l h_{l-1} + W_l \Delta h_{l-1} + \Delta W_l \Delta h_{l-1}$. - **Simplification:** Assuming that the leading order terms don't cancel, we see that: - $\|W_l \Delta h_{l-1}\| = \Theta(\sqrt{n_l})$ from the inductive assumption and condition A1 argument. - $\|\Delta W_l h_{l-1}\| = \Theta(\sqrt{n_l})$ from above. - $\|\Delta W_l \Delta h_{l-1}\| = O(\sqrt{n_l})$. - **Key Insight:** We need to figure out how big ΔW_l is. This is the product of the previous layer's norm times the operator norm of ΔW_l. - **Loss Change:** The change in loss ΔL should also be $\Theta(1)$ (constant, not blowing up or vanishing). - $\Delta L \approx \langle \nabla_l L, \Delta W_l \rangle$. - We know $\Delta W_l = -\eta_l \nabla_l L \cdot h_{l-1}^T$. - Substituting and solving for η_l, we get $\eta_l = \frac{n_l}{n_{l-1}}$.

Conclusion for Condition A2: For SGD, the learning rate $\eta_l = \frac{n_l}{n_{l-1}}$. This means the learning rate should be scaled by the ratio of fan-out to fan-in.

[3:23:00] MUP Mini Recap

What is MUP about? Controlling activations (and changes) via W and ΔW.

Initialization: Set to $\Theta\left(\frac{1}{\sqrt{\min(n_l, n_{l-1})}}\right)$. - This is $1/\sqrt{\text{fan-in}}$ if fan-out $\geq$ fan-in. - This is $1/\sqrt{\text{fan-out}}$ if fan-out $<$ fan-in.

Learning Rates: - For SGD: Set to $\Theta\left(\frac{n_l}{n_{l-1}}\right)$ (fan-out / fan-in). - For Adam: Set to $\Theta\left(\frac{1}{n_{l-1}}\right)$ (1 / fan-in).

Compared to 'standard' parameterizations: - Initialization: Set to $1/\sqrt{\text{fan-in}}$. - Learning Rates: Set to $\Theta(1)$ (global constant).

Differences: - LR changes for Adam, also init diffs when fanout $n_l <$ fanin n_{l-1}.

[3:26:00] Implementation in CerebrasGPT

The CerebrasGPT paper provides a table (Table 2) detailing their MUP implementation. - **Embedding Layer:** Special case, scaled by $1/\sqrt{n_{emb}}$. - **Other Layers:** - Initialization: Scaled by $1/\sqrt{n_{in}}$ (fan-in). - Learning Rates: Scaled by n_{out}/n_{in} (fan-out / fan-in). - This aligns with the derived MUP rules.

[3:28:00] What is MUP Robust To?

The paper by Lucas Dax Lingle explores the robustness of MUP. - **Question:** Modern LLMs have many components that deviate from MUP's theory. Which of these (if any) break MUP? - **Variations Tested:** - Activations: SwiGLU, squared ReLU. - Batch sizes: Large/small. - Initialization variations: Zero attention, etc. - RMS norm gains. - Exotic optimizers (Lion). - Regularizers.

Robustness to Nonlinearities: - SwiGLU, Squared ReLU have the same optimal LR and both provide minor gains. - The optimal learning rate remains stable across different nonlinearities (SwiGLU, Squared ReLU, ReLU). - This suggests MUP is robust to changes in activation functions.

Robustness to Batch Sizes: - Larger and smaller batches. - The original derivation doesn't handle batch size considerations. - MUP remains robust to varying batch sizes.

Robustness to Initialization: - New initializations are sometimes used. - SP Unembedding: Pre-softmax linear layer. $1/M$ (SP) vs $1/\sqrt{M}$ (MUP). - Zero Query: Set the query matrix to zero (so that all items get uniform attention). - MUP remains robust to these initialization variations.

What MUP is NOT Robust to? RMSnorm gain: - In our arch, RMSnorm has learnable gains. This turns out to break MUP. - If learnable gains are added to RMSnorm, MUP breaks. - However, these gains can be removed with little loss of performance.

What MUP is NOT Robust to? Exotic Optimizers: - There are other, exotic optimizers based on just gradient signs. Do they transfer? - Lion optimizer (which uses the sign of gradients) breaks

MUP. - This is expected, as MUP is designed for specific optimizers like AdamW.

What MUP is NOT Robust to? (Strong) Weight Decay: - Strong weight decay (e.g., 0.1) breaks MUP. - This is one of the few significant MUP failures.

[3:30:00] Recap: Scaling in the Wild

Challenges in scaling 'in practice': 1. Setting model architecture hyperparameters (width, etc.). 2. Setting optimizer hyperparameters (LR, batch). 3. Compute needed to fit the big Chinchilla sweep.

Some solutions: 1. Assume stability (or use MUP). 2. Search for optimal LR/batch in small scale, either keep fixed or predict scaling. 3. Use alternative learning schedules (WSD-like).

Practical Takeaways

- **MUP for Hyperparameter Management:** MUP is a valuable tool for stabilizing hyperparameters across model scales, reducing the computational burden of hyperparameter tuning for large models.
- **WSD for Data Scaling:** WSD learning rate schedules enable efficient data scaling analysis by allowing the reuse of training runs and rewinding checkpoints, making Chinchilla-style studies more feasible.
- **Chinchilla is a Guideline, Not a Law:** While Chinchilla-style analyses are widely used, the optimal token-to-parameter ratios can vary significantly across architectures and datasets. The 20:1 ratio is a starting point, not a strict rule.
- **Deep Dive into Optimizers:** The choice of optimizer (SGD vs. Adam vs. exotic ones) significantly impacts how MUP should be applied, particularly for learning rate scaling.
- **Empirical Validation is Key:** Despite theoretical derivations, real-world scaling involves many complexities (nonlinearities, regularization, exotic optimizers) that require extensive empirical testing to validate the robustness of scaling laws like MUP.

Open Questions / Things to Remember

- **Proprietary Nature of Scaling:** The lack of detailed public information from leading labs makes it challenging to understand the current state-of-the-art in scaling practices.
- **Robustness of MUP:** While MUP offers significant advantages, its robustness to various modern architectural components, optimizers, and regularization strategies needs careful empirical validation.
- **Evolving Optimal Ratios:** The "optimal" token-to-parameter ratio is not fixed and can change with architectural innovations and data quality, suggesting a need for continuous re-evaluation rather than strict adherence to historical benchmarks.
- **Trade-offs in Scaling:** There are inherent trade-offs between computational cost, model size, data size, and hyperparameter stability. Effective scaling involves navigating these trade-offs strategically.
- **Physics Analogy:** The idea of "renormalization" from physics, where one seeks stable properties across different scales, provides a powerful conceptual framework for MUP.

← Lec 10

Lec 12 →

Lecture 12: Evaluation

Stanford CS336 · Spring 2025 · 1:20:48 · Watch on YouTube ↗

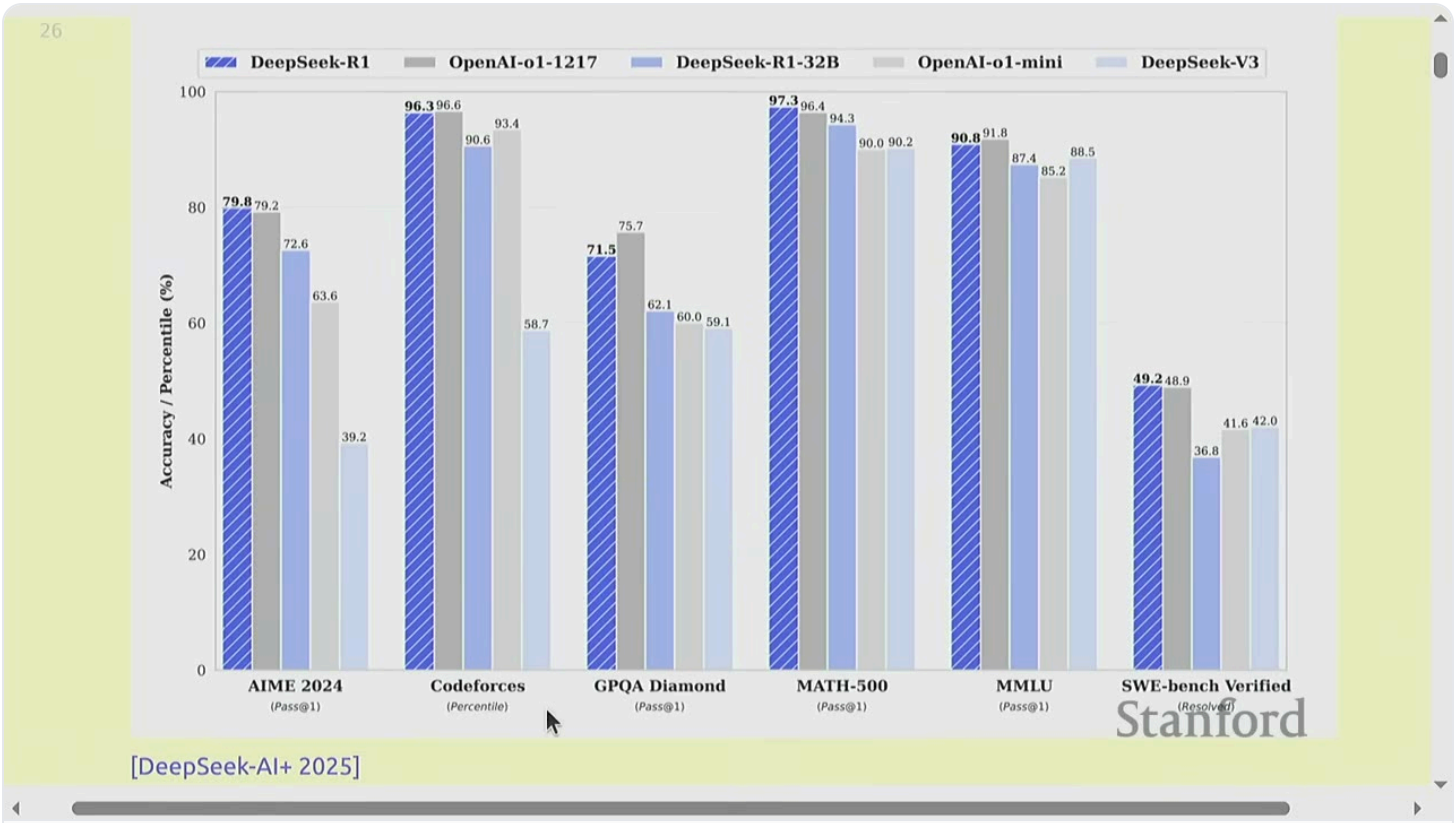
📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



TL;DR:



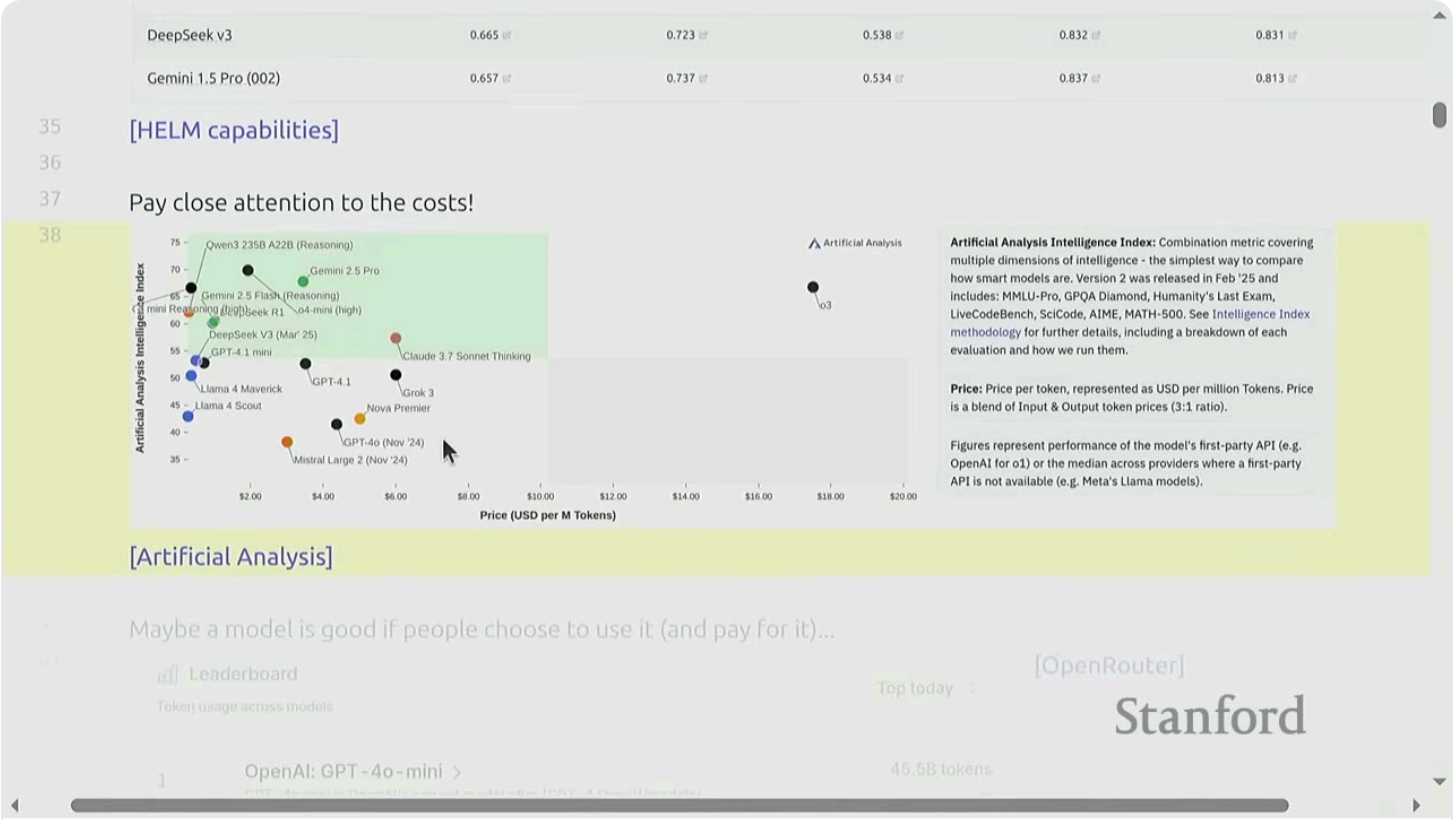
0:29 Code snippet defining "what_you_see" and a bar chart of benchmark scores.



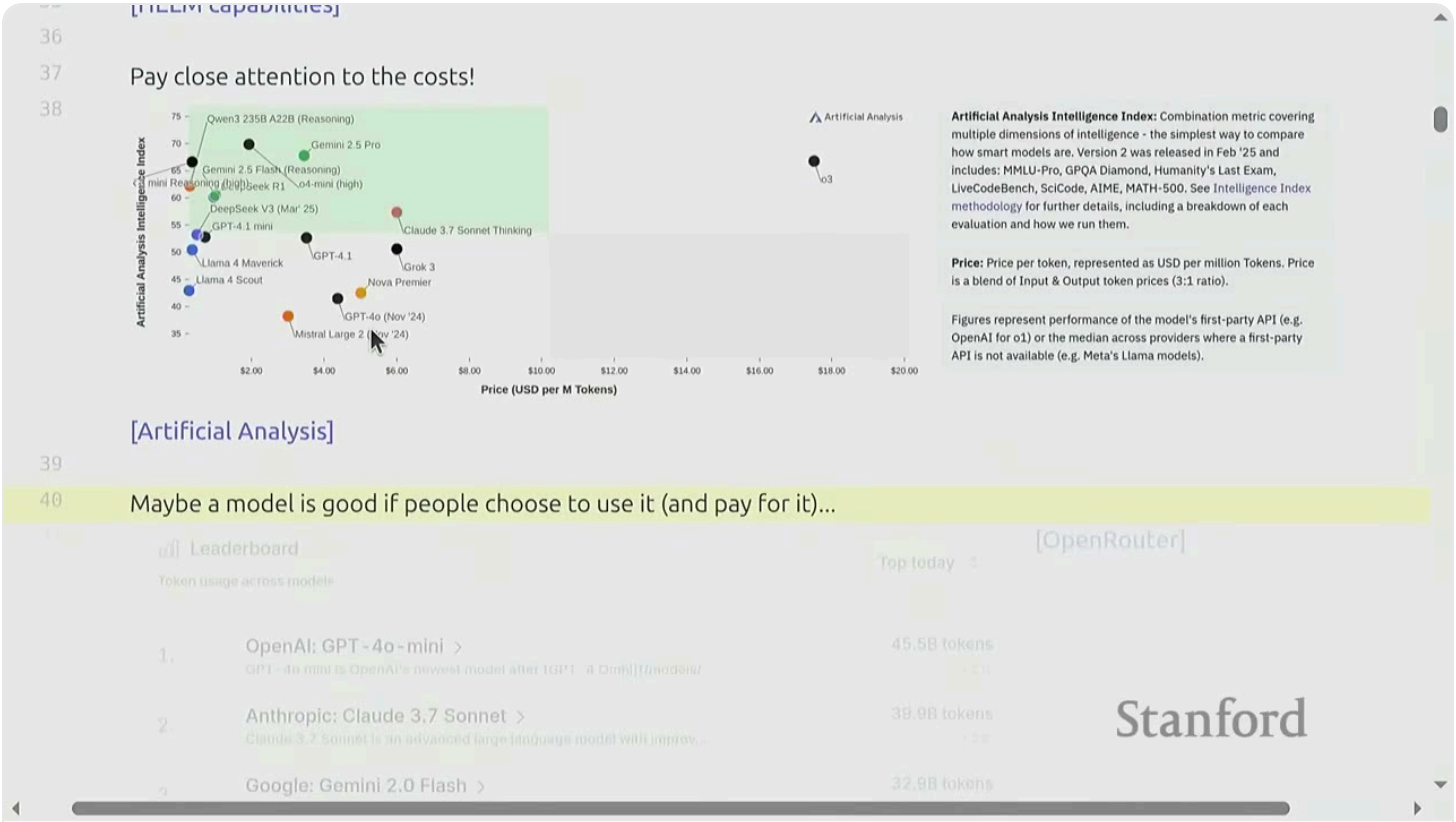
0:42 Bar chart comparing accuracy of different models across various benchmarks.



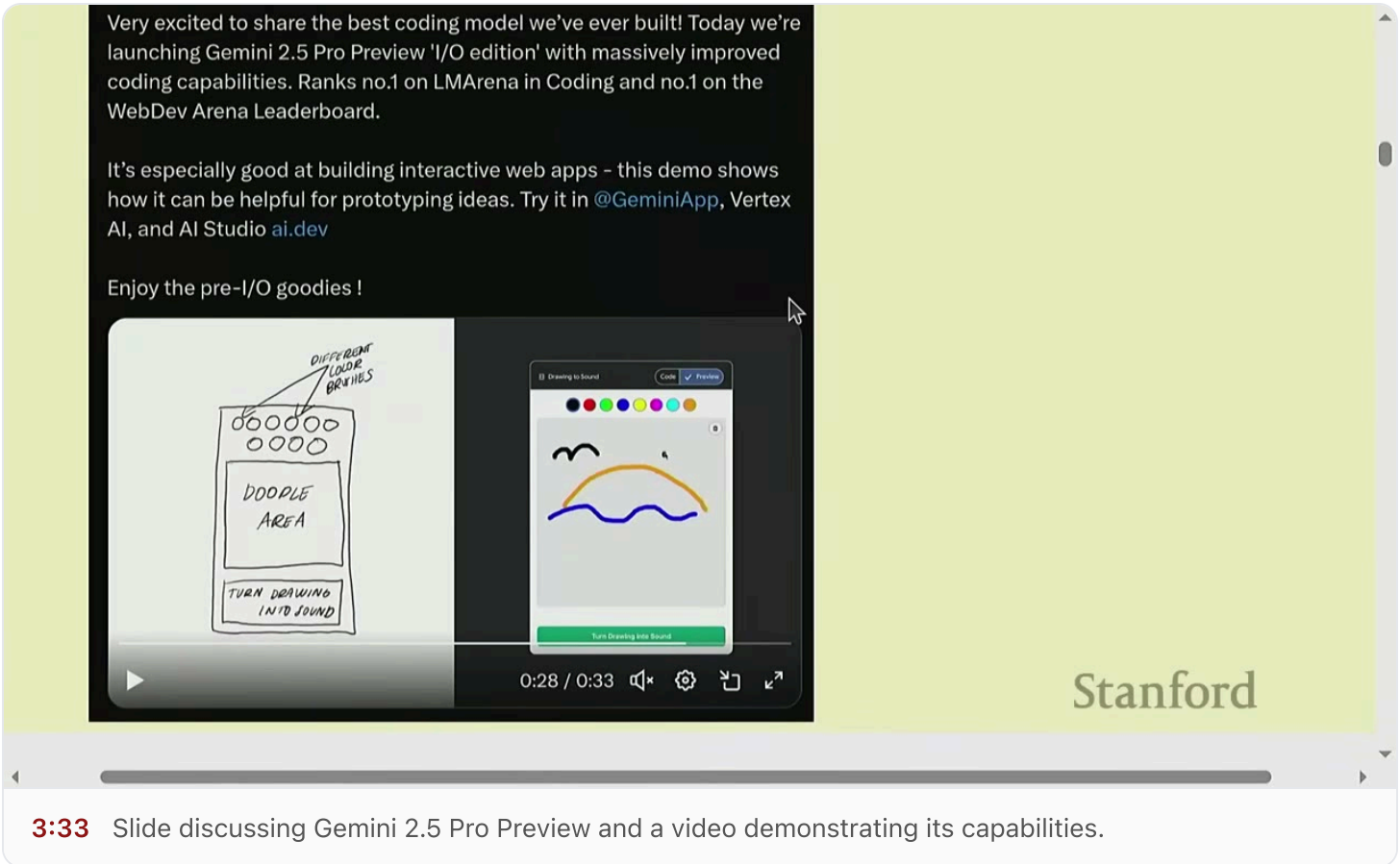
0:56 Table showing benchmark scores for Llama 4, Claude Sonnet, Gemini Pro, and GPT-4.5.



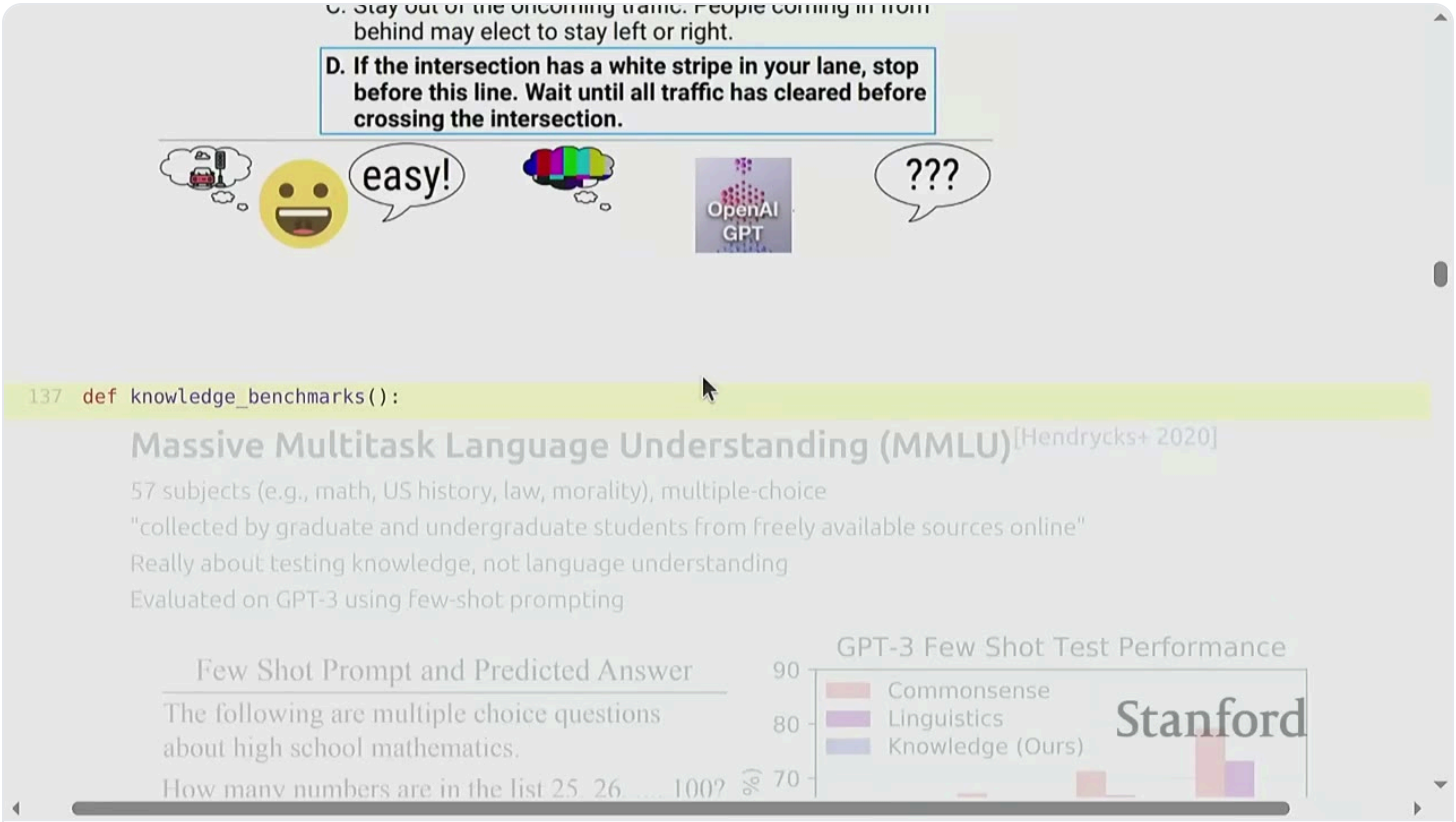
2:02 Scatter plot showing Artificial Analysis Intelligence Index vs. price per M tokens.



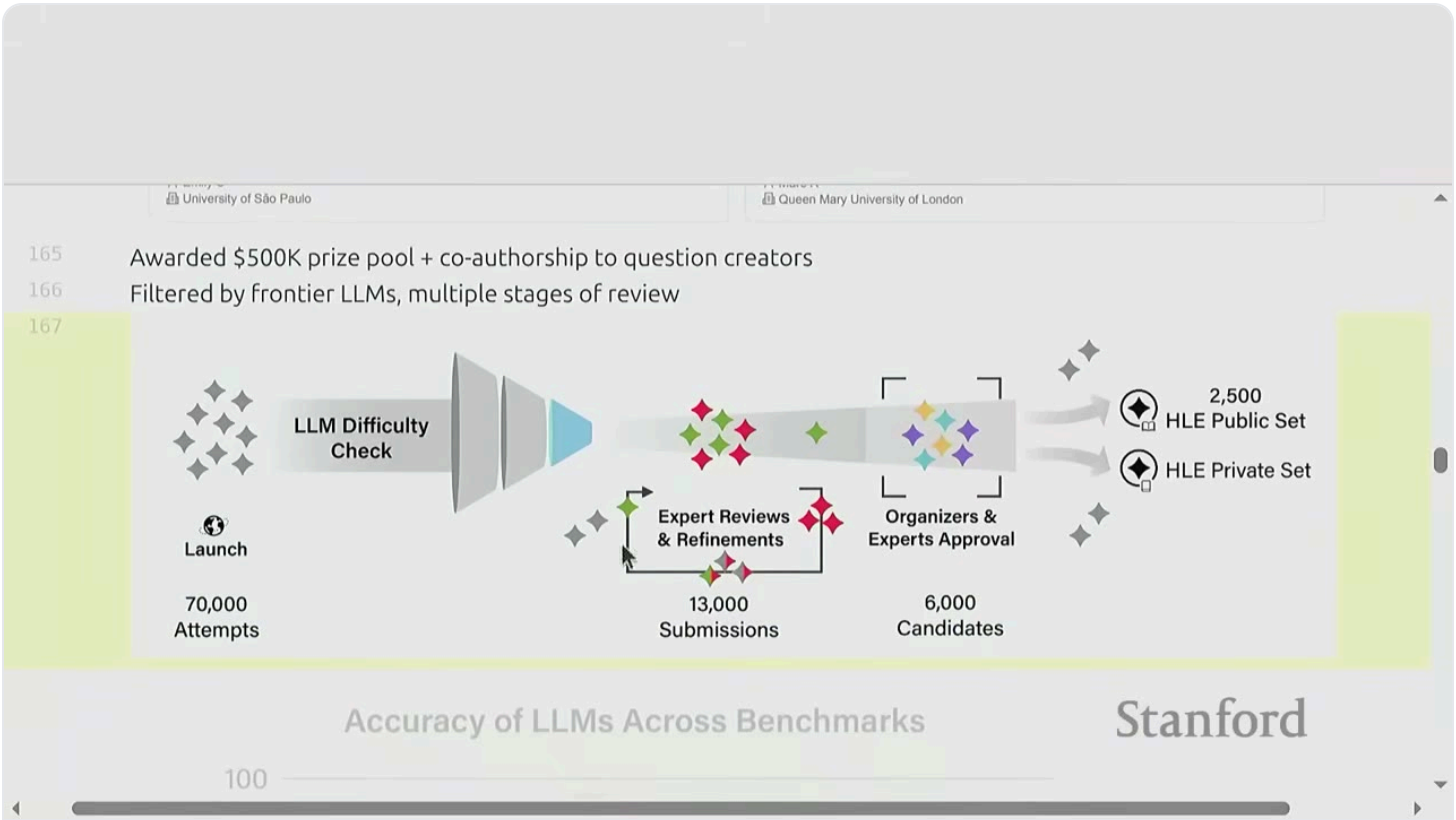
2:21 Scatter plot showing Artificial Analysis Intelligence Index vs. price per M tokens.



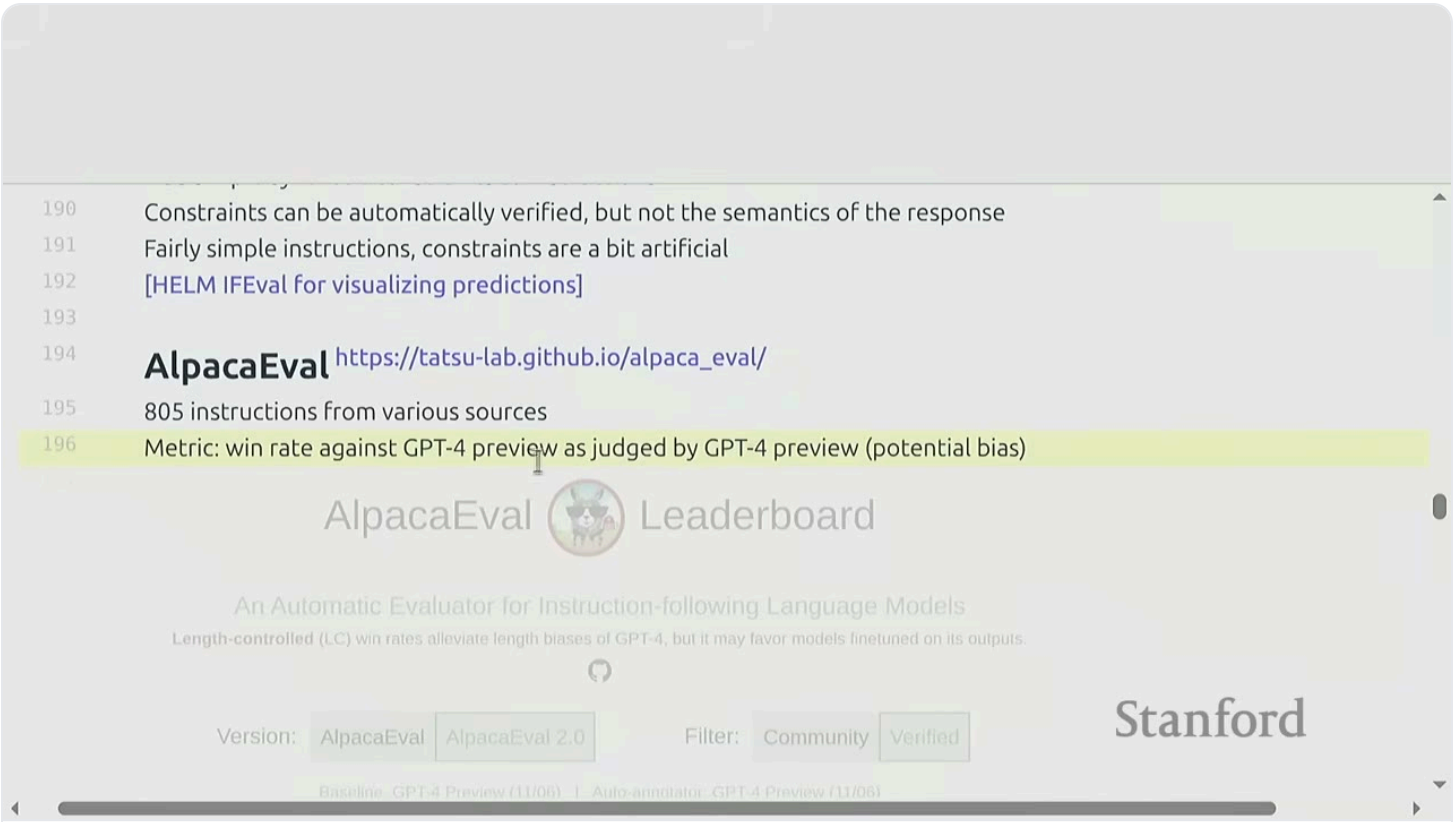
3:33 Slide discussing Gemini 2.5 Pro Preview and a video demonstrating its capabilities.



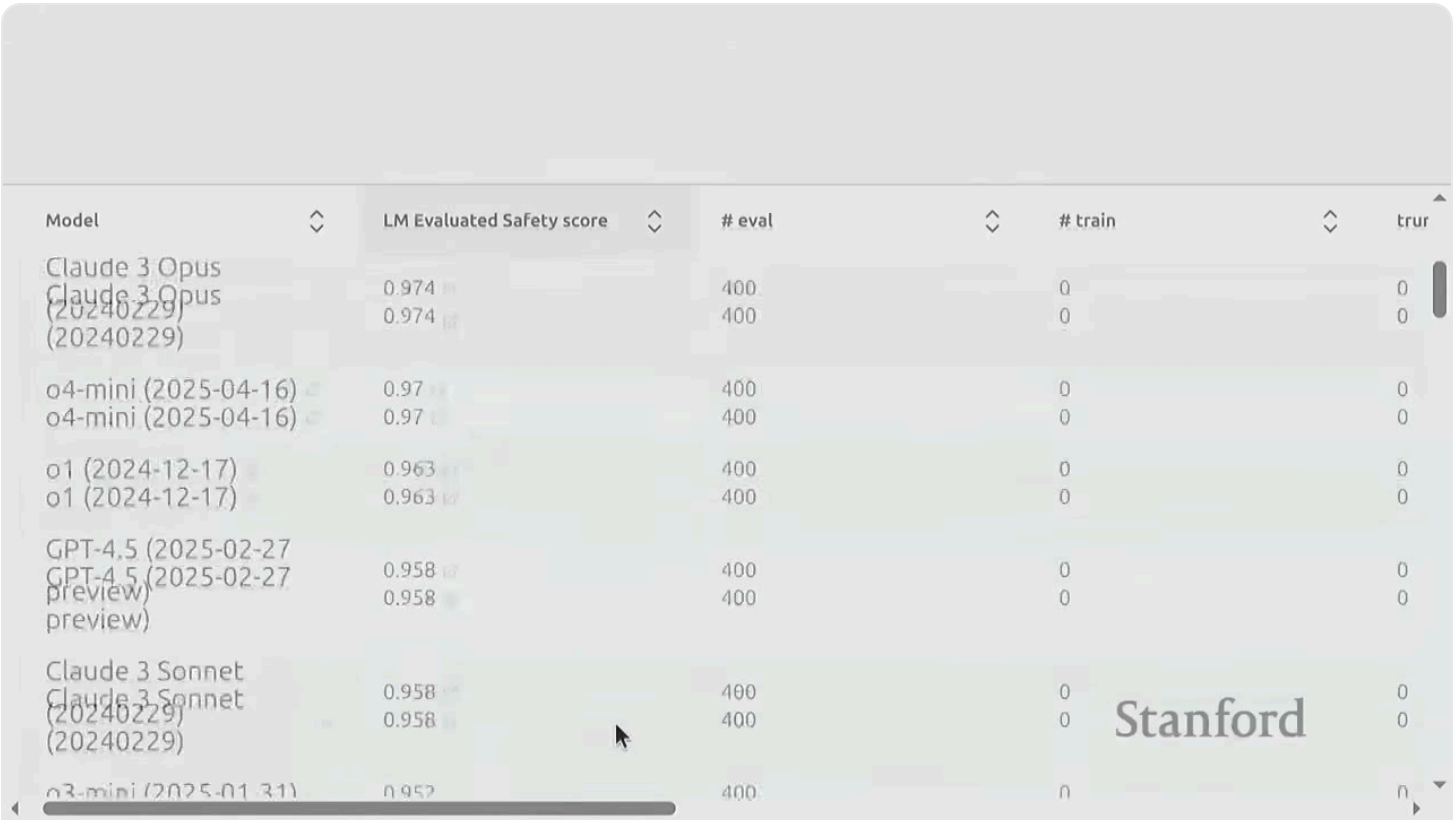
32:22 Slide showing a multiple-choice question and a code snippet for knowledge benchmarks.



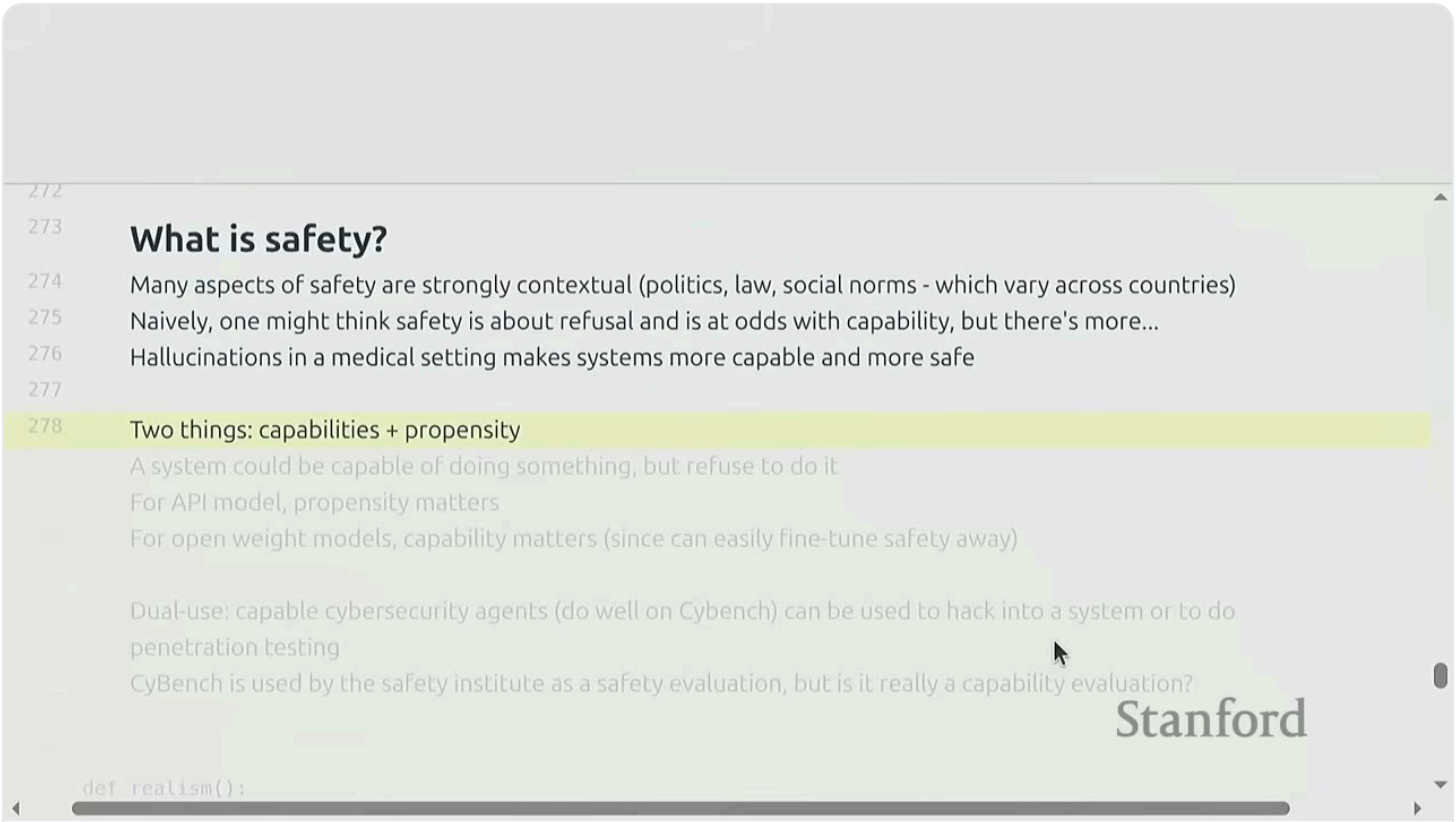
49:58 Diagram illustrating the process of LLM difficulty check and expert reviews.



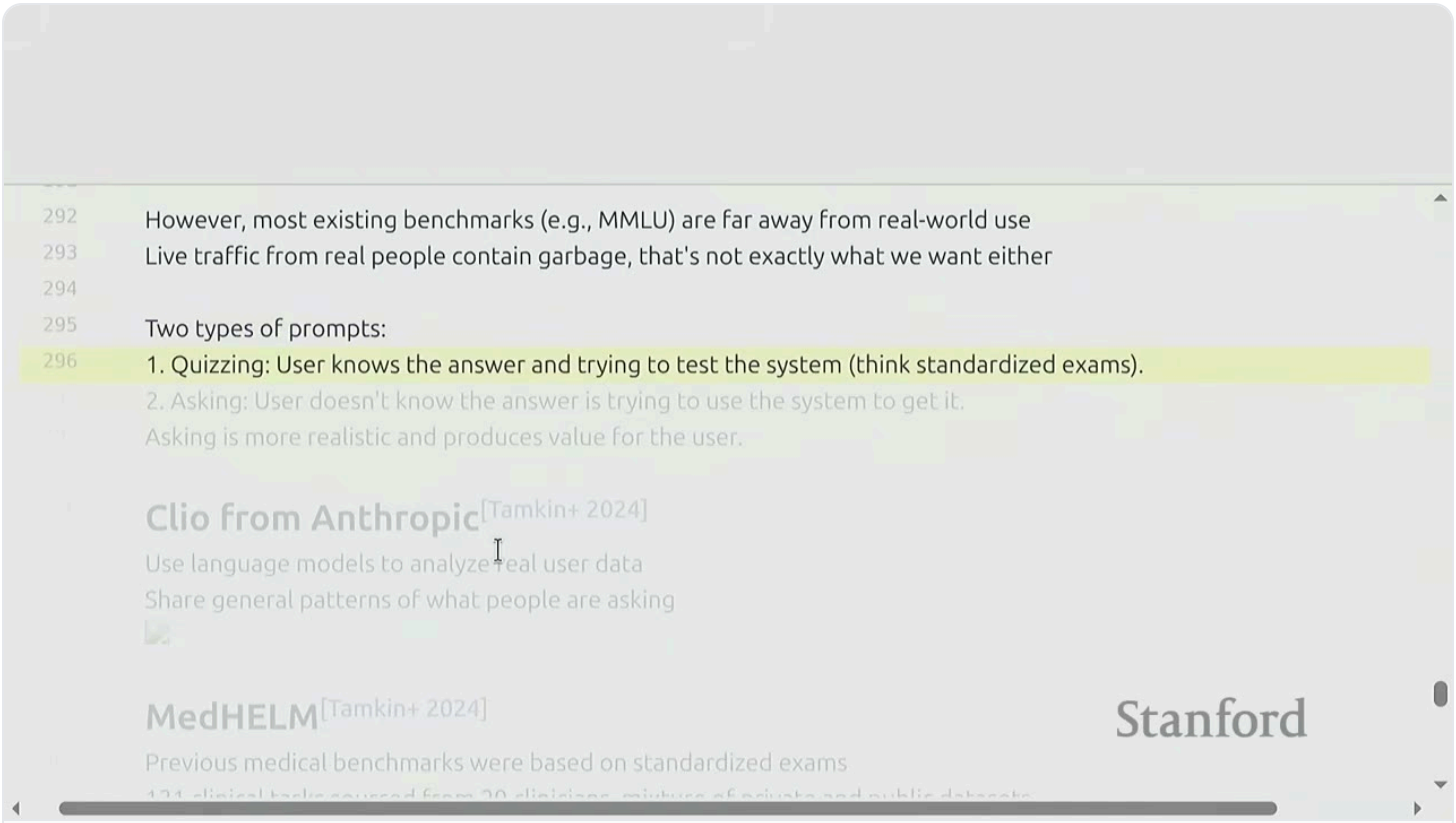
58:20 Slide introducing AlpacaEval, a metric for instruction-following language models.



67:30 Table displaying LM Evaluated Safety scores for various models.



72:22 Slide defining safety in language models, focusing on capabilities and propensity.



74:40 Slide discussing types of prompts: quizzing and asking, and Clio from Anthropic.

- Evaluation is a complex topic that goes beyond simple metrics, influencing how language models are built and used.
- There's no single "true" evaluation; the best approach depends on the specific question being asked (e.g., user decision, research, policy, model development).
- Key aspects of evaluation include inputs (prompts), how the model is called (prompting strategies, agentic systems), how outputs are evaluated (metrics, cost, error types), and how results are interpreted (generalization, model vs. method).
- Perplexity, while a foundational metric, has limitations and is being supplemented by task-specific and human-preference benchmarks.
- Challenges include train-test overlap, data quality, and defining safety in AI, highlighting the need for robust and transparent evaluation protocols.

Key concepts: - Perplexity - MMLU (Massive Multitask Language Understanding) - GPQA (Graduate-Level Google-Proof Q&A) - Humanity's Last Exam (HLE) - Instruction-following benchmarks (Chatbot Arena, IFEval, AlpacaEval, WildBench) - Agent benchmarks (SWEBench, CyBench, MLEBench) - Pure reasoning benchmarks (ARC-AGI) - Safety benchmarks (HarmBench, AIR-Bench, Jailbreaking) - Pre-deployment testing - Realism and Validity in evaluation - Train-test overlap and contamination

---

[00:00] **Introduction to Evaluation** The lecture begins by introducing evaluation as a seemingly simple but profoundly complex topic in language modeling. Mechanically, it's about assessing a fixed model's performance. However, the implications of how we evaluate models significantly influence their development and future.

[00:25] **What You See: Current Landscape of Evaluation** The current landscape of language model evaluation is characterized by a plethora of benchmarks and metrics.

- **Benchmark Scores**: Papers often report scores on various benchmarks like MMLU, AIME, Codeforces, Math 500, GPQA, GSM8K, etc. (e.g., DeepSeek, Llama 4, OLMo).
- **Cost-Benefit Analysis**: Websites like Artificial Analysis evaluate models based on a combination of an "intelligence index" (derived from multiple benchmarks) and the price per token. This helps visualize Pareto frontiers for models, showing trade-offs between performance and cost.
- **User Preference/Usage**: Platforms like OpenRouter track the number of tokens sent to different models, implicitly suggesting that models with higher usage are "better" because users choose them.

- **Human Preference/Elo Ratings**: Chatbot Arena allows users to interact with models and express pairwise preferences, which are then used to calculate Elo-like ratings, ranking models by perceived quality.
- **"Vibes"**: Informal demonstrations on social media (e.g., X) showcasing impressive model capabilities.

[03:40] **The Evaluation Crisis** Andre Karpathy's tweet highlights a growing "evaluation crisis": - Traditional benchmarks like MMLU may be saturated or "gamed." - Chatbot Arena, while popular, has its own issues (e.g., reliance on API requests, prompt bombing, explicit use of rankings as training supervision). - There's a proliferation of benchmarks, but a lack of comprehensive, unbiased, and low-sample-size evaluations.

The overall sentiment is that the current state of evaluation is messy, and it's unclear which methods are truly reliable.

[04:40] **How to Think About Evaluation** Evaluation is not a mere mechanical process; it's a profound topic that shapes the future of language models. The choice of evaluation methods directly influences how models are built, as developers optimize for metrics.

[05:55] **What's the Point of Evaluation?** There is no single "true" evaluation. The purpose of evaluation depends entirely on the question one is trying to answer. 1. **User/Company**: To make a purchase decision (e.g., choosing a customer service chatbot). 2. **Researcher**: To measure the raw capabilities of a model and understand scientific progress in AI. 3. **Policymaker/Business**: To objectively understand the benefits and harms of a model for regulatory or business reasons. 4. **Model Developer**: To get feedback for improving the model during its development cycle.

In each case, there's an abstract goal that needs to be translated into a concrete evaluation.

[08:05] **Framework for Evaluation** A simple framework for thinking about evaluation involves four key questions: 1. **What are the inputs?** (The prompts) 2. **How do you call the language model?** 3. **How do you evaluate the outputs?** 4. **How to interpret the results?**

Let's break these down:

1. What are the inputs? - **Use cases covered**: Do the prompts represent a broad range of real-world use cases, or are they narrow and specific? - **Representation of difficult inputs**: Do the inputs challenge the model's limits, or are they easy, "vanilla" cases? - **Inputs adapted to the model**: In multi-turn (chatbot) settings, inputs are inherently adapted. For red-teaming, adapting inputs can be more efficient. However, adapting inputs makes cross-model comparison difficult.

2. How do you call the language model? - **Prompting strategy**: Few-shot, zero-shot, chain-of-thought, tool use, RAG, etc. Different strategies can significantly impact performance. Language models are very sensitive to prompts. - **Evaluating the language model vs. agentic system**: Model developers might focus on the raw language model, while users care about the overall system performance (including scaffolding, tools, etc.).

3. How do you evaluate the outputs? - **Reference outputs**: Are the reference outputs used for evaluation error-free? (Often, they are not.) - **Metrics used**: Which metrics are appropriate (e.g., `pass@k` for code generation)? - **Cost factor**: How is inference/training cost factored in? (Pareto frontiers are often useful here.) - **Asymmetric errors**: Are all errors equally bad? How are critical errors (e.g., hallucinations in medical settings) handled? - **Open-ended generation**: How do you evaluate responses without a clear ground truth (e.g., "Write me a compelling story")?

4. How to interpret the metrics? - **Interpreting a number**: What does a score (e.g., 91%) truly mean? Is it sufficient for deployment? - **Assessing generalization**: How do we know if the model has truly learned or just memorized? This involves confronting train-test overlap. - **Evaluating the final model vs. the method**: In research, the goal is often to evaluate a new method, not just a specific model instance. This requires careful controls.

[12:20] **Perplexity** Perplexity is a fundamental metric in language modeling. - **Definition**: A language model defines a probability distribution $p(x)$ over sequences of tokens. Perplexity, $PPL(D) = (1/\prod_{x \in D} p(x))^{1/|D|}$, measures how well the model assigns high probability to a given dataset D. - **Pre-training**: During pre-training, models minimize perplexity on the training set. - **Evaluation**: The obvious way to evaluate a language model is to measure its perplexity on a held-out test set, typically with an i.i.d. split.

[17:00] **Perplexity in the 2010s** In the 2010s, language modeling research heavily relied on perplexity. - **Standard datasets**: Penn Treebank (WSJ), WikiText-103 (Wikipedia), 1 Billion Word Benchmark (from machine translation data like Europarl, UN news). - **Research approach**: Researchers would train models on a designated training split and evaluate on a designated test split, reporting perplexity. - **Advancements**: Papers like Jozefowicz et al. (2016) showed dramatic reductions in perplexity (e.g., from 51.3 to 30.0) using CNNs + LSTMs on the 1 Billion Word Benchmark. This was a significant achievement for advancing the field.

[19:00] **GPT-2 and the Shift in Evaluation** GPT-2 (Radford et al., 2019) marked a turning point. - **Training data**: GPT-2 was trained on a massive 40GB web text dataset (websites linked from Reddit). - **Evaluation**: It was evaluated directly on standard perplexity benchmarks without fine-tuning (zero-shot). - **Results**: GPT-2 achieved state-of-the-art perplexity on many smaller

benchmarks (like Penn Treebank) without specific training on them, demonstrating strong generalization. For larger datasets (like 1 Billion Word Benchmark), direct training was still superior.

[20:40] **Discussion on Train-Test Overlap** The question arises: if GPT-2 was trained on web data, how do we know it wasn't implicitly trained on the test sets (e.g., Penn Treebank)? - **Decontamination**: Typically, researchers perform decontamination by removing any test set data (documents, paragraphs, n-grams) that overlap with the training data. - **Subtleties**: This process is complex, as near-duplicates or paraphrases might still exist, leading to potential contamination. - **Conservative approach**: Given the vastness of web data, if a model performs well on a test set even after careful decontamination, it's generally considered a valid result.

[22:20] **Perplexity vs. Downstream Task Accuracy** Since GPT-2 and GPT-3, language modeling papers have shifted focus towards downstream task accuracy. However, perplexity remains useful for several reasons: - **Smoother for scaling laws**: Perplexity provides a smoother signal than downstream task accuracy, making it easier to fit scaling curves. - **Universal measure**: Perplexity is "universal" in the sense that it pays attention to every token in a dataset, potentially capturing nuances that task-specific accuracy might miss. It's harder to game because it requires the model to accurately predict probabilities across the entire vocabulary. - **Conditional perplexity**: Perplexity can also be measured conditionally on downstream tasks (e.g., probability of the correct answer given the prompt), which is being explored in recent scaling law papers (Bhagias et al., 2024).

[25:25] **Caveats of Perplexity Evaluation** - **Trust in model provider**: When running a leaderboard for perplexity, the evaluator needs to trust that the model provider is correctly computing probabilities that sum to one. If the model has a bug or intentionally misreports probabilities, the perplexity score can be misleading. - **Perplexity maximalist view**: - If the true data distribution is T and the model is P, the best possible perplexity is the entropy of T, achieved when $P = T$. - If a model achieves $P = T$, it has effectively "solved" all tasks and reached AGI. - **Caveat**: Pushing down perplexity might not be the most efficient way to reach AGI, as it might involve optimizing for parts of the distribution that don't matter for specific tasks.

[30:00] **Things That Are Spiritually Perplexity** Several tasks resemble perplexity but are not identical: - **Cloze tasks (LAMBADA)**: Given a context, the model predicts a missing word. These tasks often require long-range context and have been largely "saturated" by modern language models. - **Common sense reasoning (HellaSwag)**: Given a sentence, the model chooses the

most plausible completion from several options. This is essentially measuring the likelihood of different continuations.

These tasks are effectively solved by models that are good at perplexity, as they fundamentally test the model's ability to predict the next token or sequence.

[31:15] **Train-Test Overlap in "Spiritually Perplexity" Tasks** A common issue in these tasks is train-test overlap. For example, HellaSwag data was mined from ActivityNet and WikiHow. If the training data for a model includes content from WikiHow, then the model might perform well on HellaSwag not due to true reasoning but due to memorization. Researchers typically try to decontaminate datasets, but near-duplicates or paraphrases can still exist.

[32:00] **Knowledge Benchmarks** These benchmarks assess a model's factual knowledge and understanding across various domains.

[32:50] **MMLU (Massive Multitask Language Understanding)** - **Description**: A multiple-choice question answering benchmark covering 57 subjects (e.g., math, US history, law, morality). - **Origin**: Introduced in 2020 (Hendrycks et al.) shortly after GPT-3's release. - **Data**: Questions collected from graduate/undergraduate students and freely available online sources. - **Purpose**: Primarily tests knowledge, not language understanding (as the lecturer argues, based on personal experience). - **Evaluation**: Originally evaluated GPT-3 using few-shot prompting (providing examples of questions and answers in the prompt). - **Few-shot prompting**: A simple instruction, followed by examples demonstrating the desired format, then the final question. - **Impact**: This was crucial before instruction tuning, as models would often generate irrelevant responses without examples. - **Helm MMLU**: The Helm framework provides a leaderboard for MMLU, allowing users to inspect individual predictions and model performance across subjects. - **Prompting strategy**: Helm uses a standardized prompt strategy (e.g., 5-shot) which might yield lower scores than highly optimized prompts (e.g., using chain-of-thought or ensembling). - **Interpretation**: A high MMLU score from a base model (trained without specific instruction tuning) suggests strong general intelligence, as it indicates the model has learned a vast amount of world knowledge without explicit instruction.

[47:00] **MMLU-Pro** - **Description**: A more robust and challenging version of MMLU (Wang et al., 2024). - **Improvements**: - Removed noisy/trivial questions from the original MMLU. - Expanded from 4 choices to 10 choices, making guessing harder. - Evaluated using chain-of-thought (CoT) prompting, which gives models more "thinking" time and often improves accuracy. - **Impact**:

MMLU-Pro scores are generally lower than MMLU, indicating that models are not yet saturated on this harder benchmark. It's becoming a new standard for evaluating frontier models.

[54:00] **GPQA (Graduate-Level Google-Proof Q&A)** - **Description**: A challenging benchmark of multiple-choice questions written by PhD contractors from Upwork (Rein et al., 2023). - **Difficulty**: Questions are at a PhD level in biology, physics, and chemistry. - **Validation Process**: An elaborate process involving question writers, expert validators (who provide feedback and explanations), and non-expert validators (who spend significant time trying to answer, even with Google). - **Results**: Experts achieve ~65% accuracy. Non-experts (even with Google) achieve ~30% accuracy. GPT-4 achieved 39% accuracy at the time of release. - **Interpretation**: The "Google-proof" aspect means that even with extensive web search, non-experts struggle to answer, indicating the questions require deep understanding, not just information retrieval. - **Helm GPQA**: The Helm leaderboard shows that current frontier models (e.g., 03) achieve ~75% accuracy with CoT, demonstrating significant progress since GPQA's release.

[1:01:00] **Humanity's Last Exam (HLE)** - **Description**: A multimodal, multi-subject, multiple-choice + short-answer benchmark (Phan et al., 2025). - **Scale**: 2500 questions across 100 subjects. - **Purpose**: Designed to be the "final closed-ended academic benchmark at the frontier of human knowledge," aiming to push the limits of language models. - **Data Creation**: Involved a complex process: 1. Awarded $500K prize pool + co-authorship to question creators. 2. LLMs were used to filter out "too easy" questions. 3. Multiple stages of expert review and refinement. - **Results**: Current frontier models (e.g., 03) achieve ~20% accuracy on HLE, indicating it's still a very challenging benchmark.

[1:04:40] **Instruction-Following Benchmarks** These benchmarks evaluate a model's ability to follow instructions, a capability popularized by ChatGPT. The challenge is to evaluate open-ended responses.

[1:05:00] **Chatbot Arena** - **Description**: An open platform for evaluating LLMs by human preference (Chiang et al., 2024). - **How it works**: 1. Random internet users type a prompt. 2. They receive responses from two anonymous models. 3. Users rate which response is better. 4. Elo scores are computed based on pairwise comparisons, ranking models. - **Features**: Live (not static) inputs, can accommodate new models dynamically. - **Challenges**: - **Protocol issues**: The "leaderboard illusion" (paper by Zheng et al., 2023) highlights that some model providers might game the system (e.g., multiple submissions, privileged access). - **Input distribution**: The

prompts come from random internet users, which might not represent a desired or controlled distribution. - **Human bias**: Human preferences can be subjective and inconsistent.

[1:08:00] **IFEval (Instruction-Following Eval)** - **Description**: A benchmark that narrowly tests a model's ability to follow constraints (Zhou et al., 2023). - **How it works**: 1. Define various constraints (e.g., keywords to include/exclude, length limits, formatting rules like number of paragraphs or JSON). 2. Add these synthetic constraints to instructions for open-ended generation tasks. 3. Automatically verify if the model's response adheres to the constraints using a simple script. - **Purpose**: Tests the model's "compliance" rather than the semantic quality of the response. - **Challenges**: - **Partial evaluation**: It only checks constraint adherence, not the overall quality or meaning of the generated text. - **Gameable**: Models can be optimized to strictly follow constraints without necessarily improving semantic quality.

[1:10:00] **AlpacaEval** - **Description**: An automatic evaluator for instruction-following language models (Dubois et al., 2023). - **How it works**: 1. Uses 805 instructions from various sources. 2. Metric: Win rate against GPT-4 preview, judged by GPT-4 itself. - **Challenges**: - **Potential bias**: Using an LLM (GPT-4) as a judge introduces potential bias towards responses that GPT-4 itself would prefer. - **Gaming**: Early versions were gamed by models generating longer responses, which GPT-4 tended to prefer. This was corrected with a "length-corrected" win rate. - **Correlation**: AlpacaEval's win rates correlate well with Chatbot Arena, suggesting both provide similar information about model performance.

[1:11:00] **WildBench** - **Description**: Another benchmark using LLMs as judges (Lin et al., 2024). - **How it works**: 1. Sources 1024 examples from 1M human-chatbot conversations. 2. Uses GPT-4 Turbo as a judge with a checklist (like CoT for judging) + GPT-4 as a judge. 3. Correlates well (0.95) with Chatbot Arena.

[1:12:00] **Agent Benchmarks** These benchmarks evaluate models that require tool use (e.g., running code, accessing the internet) and involve iterating over a period of time. An agent consists of a language model and an "agent scaffolding" (programmatic logic for deciding how the LM is used).

[1:12:40] **SWEBench** - **Description**: Evaluates agents on software engineering tasks (Jimenez et al., 2023). - **How it works**: 1. Given 2294 tasks across 12 Python repositories. 2. Given a codebase and a GitHub issue description. 3. The agent must submit a Pull Request (PR) that resolves the issue. 4. **Evaluation metric**: Unit tests must pass. - **Results**: Current agent accuracies are still relatively low (e.g., sub-20% for passing any medal threshold), but improving.

[1:13:20] **CyBench** - **Description**: A framework for evaluating cybersecurity capabilities (Chan et al., 2024). - **How it works**: 1. Agents participate in Capture The Flag (CTF) competitions. 2. The agent has access to a server and must hack into it to retrieve a secret key. 3. **Agent architecture**: A standard "Act-Execute-Update" loop where the LM thinks, generates commands, executes them in an environment, observes the results, and updates its memory. - **Results**: Accuracies are still low (e.g., ~20% for 03-mini), but models are showing progress in solving tasks that took humans significant time (e.g., 42 minutes for 03-mini).

[1:14:00] **MLEBench** - **Description**: Evaluates agents on Kaggle-like competitions (Chan et al., 2024). - **How it works**: 1. 75 Kaggle competitions. 2. The agent is given a competition description and a dataset. 3. The agent must write code, train a model, debug, tune hyperparameters, and submit a solution. 4. **Evaluation metric**: Kaggle's scoring system (e.g., accuracy, medals). - **Results**: Accuracies are still low (e.g., sub-20% for achieving any medal), but improving.

[1:14:30] **Pure Reasoning Benchmarks** These benchmarks aim to isolate and measure pure reasoning ability, distinct from linguistic understanding or world knowledge.

[1:14:50] **ARC-AGI (Abstract Reasoning Corpus - AGI)** - **Description**: A challenge introduced by François Chollet (2019) to capture a "more pure form of intelligence" that isn't just memorizing facts. - **How it works**: 1. Models are given visual patterns (grids of colored cells) and must infer the underlying rule to complete new patterns. 2. No language description is provided. - **Purpose**: Designed to be easy for humans but hard for current AI, testing abstract reasoning and generalization. - **Results**: Traditionally, LLMs performed very poorly on these tasks (e.g., GPT-4o at ~0% accuracy). However, recent models like 03 are starting to show significant progress, achieving ~80% accuracy on some tasks. - **Cost**: Solving these tasks can be computationally expensive (e.g., hundreds of dollars per task for 03).

[1:15:50] **Safety Benchmarks** These benchmarks assess the safety and ethical behavior of language models.

[1:16:00] **What does safety mean for AI?** - **Contextual**: Safety is strongly contextual, depending on politics, law, social norms, and varies across countries. - **Beyond refusal**: Safety is broader than just refusal (e.g., reducing hallucinations in medical settings makes systems safer). - **Capabilities vs. Propensity**: - **Capabilities**: The ability of a system to do something (e.g., cause harm). - **Propensity**: The likelihood of a system to do something (e.g., refuse to cause harm). - For API models, propensity matters (if it refuses, it's fine). - For open-weight models, capabilities

matter more, as fine-tuning can easily bypass safety mechanisms. - **Dual-use issues**: Cybersecurity agents (like CyBench) can be used to hack systems (malicious use) or to perform penetration testing (beneficial use).

[1:16:40] **HarmBench** - **Description**: A benchmark for evaluating harmful behaviors that violate laws or norms (Mazeika et al., 2024). - **How it works**: 1. Identifies 510 harmful behaviors. 2. Prompts LLMs to perform these behaviors. 3. Evaluates if the model complies or refuses. - **Results**: Some models (e.g., 03) show high refusal rates (~90%), while others (e.g., DeepSeek) are more compliant.

[1:17:20] **AIR-Bench** - **Description**: A benchmark based on regulatory frameworks and company policies (Zeng et al., 2024). - **How it works**: 1. Builds a taxonomy of 314 risk categories (e.g., legal, social, technical risks). 2. Evaluates LLMs against prompts related to these risks. - **Purpose**: Aims to ground safety evaluation in real-world regulatory and policy contexts.

[1:18:00] **Jailbreaking** - **Description**: The process of bypassing a model's safety mechanisms to elicit harmful responses. - **Challenge**: LLMs are trained to refuse harmful instructions, but clever prompting techniques can bypass these safeguards. - **Research**: Papers like Zou et al. (2023) developed "Greedy Coordinate Gradient" (GCG) to automatically optimize prompts to bypass safety. This technique works on open-weight models (Llama) and transfers to closed models (GPT-4). - **Implications**: The ability to jailbreak models highlights the ongoing cat-and-mouse game between model developers and malicious actors, and the need for robust safety interventions.

[1:19:00] **Pre-deployment Testing** - **Collaborative effort**: US Safety Institute + UK AI Safety Institute + other countries are working together. - **Protocol**: Companies (e.g., Anthropic, OpenAI) voluntarily give early access to their models (pre-release) to safety institutes. - **Process**: Institutes run safety evaluations, generate reports, and provide feedback to the companies to inform deployment. - **Status**: This is a voluntary, non-binding protocol for now.

[1:19:30] **Realism** - **Challenge**: Most existing benchmarks (e.g., MMLU) are far from real-world use cases. - **Live traffic**: While real-life traffic provides realistic inputs, it often contains "garbage" (spammy, non-serious utterances) that are not the target distribution for evaluation. - **Types of prompts**: - **Quizzing**: User knows the answer and is testing the system (standardized exams). - **Asking**: User doesn't know the answer and is trying to use the system to get it. - **Asking prompts**: More realistic and produces value for the user. Standardized exams are not realistic but can be helpful for certain types of evaluation.

[1:20:00] **Clio from Anthropic** - **Description**: Uses language models to analyze real user data (Tamkin et al., 2024). - **How it works**: 1. Takes real user conversations with LLMs. 2. Uses LLMs to cluster these conversations and identify patterns of use. - **Purpose**: To understand how people are actually using LLMs in the wild, including common use cases (e.g., coding, education) and safety-related issues.

[1:20:20] **MedHelm** - **Description**: A project focused on medical benchmarks (Tamkin et al., 2024). - **How it works**: 1. Identifies 121 clinical tasks sourced from 29 clinicians. 2. Creates a wide suite of benchmarks based on private and public medical datasets. - **Challenge**: Realism and privacy are often at odds, as realistic medical data is sensitive and cannot be publicly hosted.

[1:20:50] **Validity** - **Core question**: How do we know our evaluations are valid? - **Train-test overlap**: - **Machine Learning 101**: Don't train on your test set. - **Pre-foundation models**: Benchmarks like ImageNet and SQuAD had well-defined train-test splits. - **Nowadays**: Models are trained on the internet, and developers don't always disclose their training data, making it difficult to verify train-test separation. - **Route 1: Infer train-test overlap**: Researchers are developing methods to infer if a model was trained on a test set by observing its behavior (e.g., exploiting exchangeability in the benchmark, Oren et al., 2023). - **Route 2: Encourage reporting norms**: Language model developers should report train-test overlap (Zhang et al., 2024), including confidence intervals.

- **Dataset quality**: Many benchmarks have quality issues. For example, SWEBench had errors, and many math benchmarks contain noise or mislabeled data. This can lead to misleading performance metrics.

[1:21:40] **What Are We Even Evaluating?** - **Fundamental question**: What are the rules of the game? - **Shift in focus**: Previously, we evaluated *methods* (e.g., new architectures, learning algorithms) using standardized train-test splits. - **Today**: We are evaluating *systems* (e.g., LLMs with various prompting strategies, tool use, agentic scaffolding) where "anything goes." - **Exceptions**: Some competitions (e.g., nanoGPT speedrun, DataComp) still focus on specific, fixed rules to encourage algorithmic innovation. - **Conclusion**: It's crucial to define the rules of the game for any evaluation and to clearly understand the purpose of the evaluation.

Practical takeaways: - **Context is King**: The "goodness" of a model is relative to the specific use case and question being asked. Avoid one-size-fits-all evaluation. - **Beyond Single Metrics**: Relying solely on perplexity or simple accuracy can be misleading. Consider a broader set of

metrics, including cost, error types, and human preferences. - **Beware of Train-Test Contamination**: With models trained on vast internet data, ensuring true generalization is challenging. Always question the source and overlap of evaluation data. - **Transparency and Reproducibility**: Clear protocols for evaluation, including prompt strategies, model access, and data handling, are essential for valid comparisons. - **Embrace Diverse Evaluation Methods**: Combine traditional benchmarks with human preference, agentic tasks, and safety evaluations to get a holistic view of model capabilities and risks.

Open questions / things to remember: - How to design robust benchmarks that are not easily gameable, especially for open-ended generation. - How to effectively evaluate agentic systems that involve complex interactions with tools and environments. - How to define and measure "safety" in AI in a way that is both comprehensive and actionable, considering its contextual nature. - The tension between realism (evaluating on real-world data) and privacy (protecting sensitive user information). - The ongoing debate about whether current evaluation methods truly capture "intelligence" or merely test memorization and pattern recognition.

← Lec 11

Lec 13 →

Lecture 13: Data 1

Stanford CS336 · Spring 2025 · 1:19:06 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



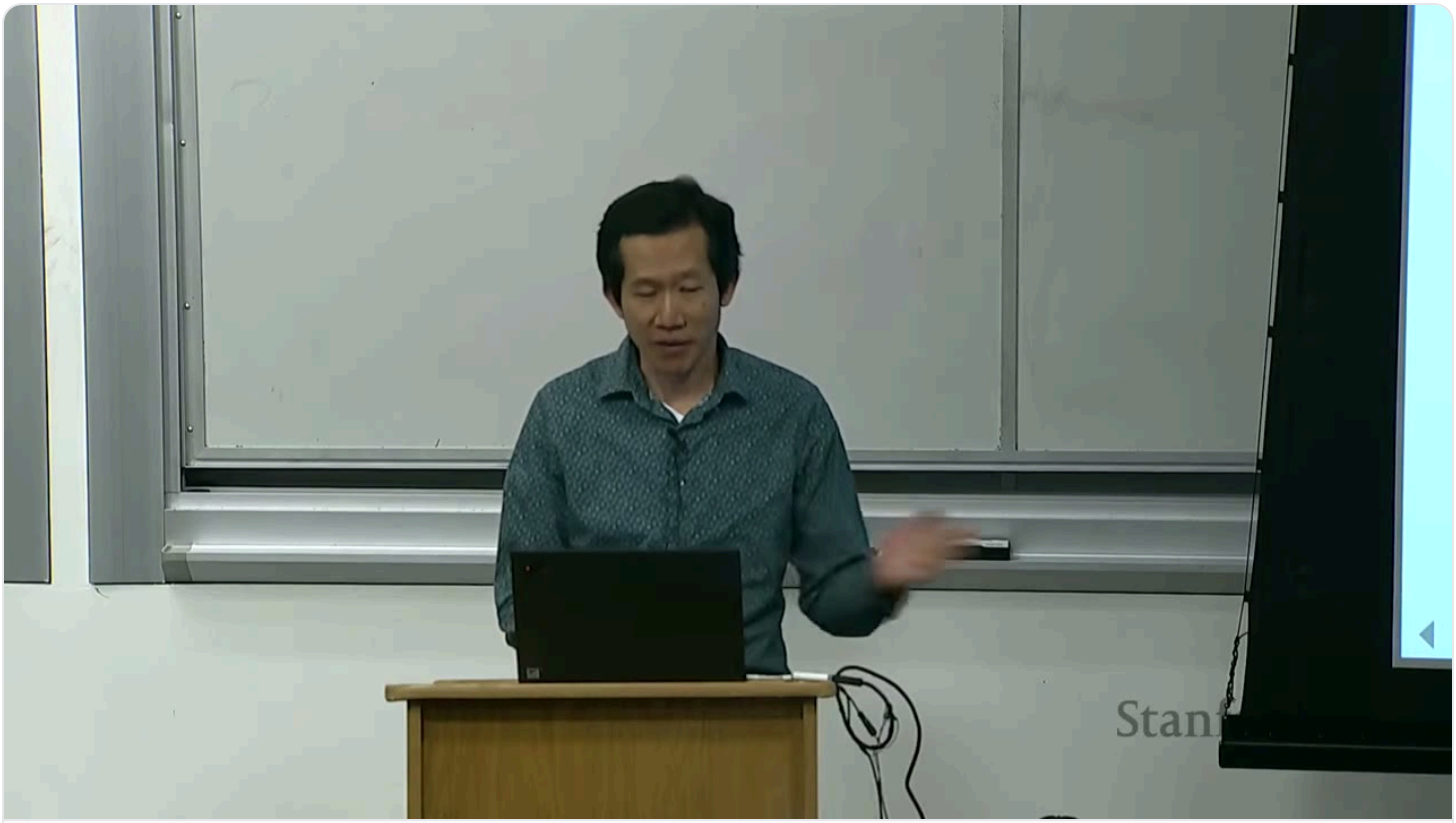
TL;DR * Data is the most critical component for successful language models, often more so than architecture or scaling laws. * Data collection and processing is a complex, multi-stage pipeline involving raw data acquisition, conversion, filtering, and deduplication. * Companies are often secretive about their data sources and processing techniques due to competitive dynamics and legal/copyright concerns. * The field of data curation is highly heuristic, with various approaches (rule-based, model-based, synthetic generation) constantly evolving. * Copyright law and terms of service significantly impact what data can be legally and practically used for training models.

Key concepts * Pre-training, Mid-training, Post-training * Data Sources: BooksCorpus, Wikipedia, WebText, Common Crawl, C4, The Pile, GitHub, StackExchange, RefinedWeb, FineWeb, Dolma, LLaMA, NemoTron-CC * Data Processing: Filtering (rule-based vs. model-based), Deduplication,

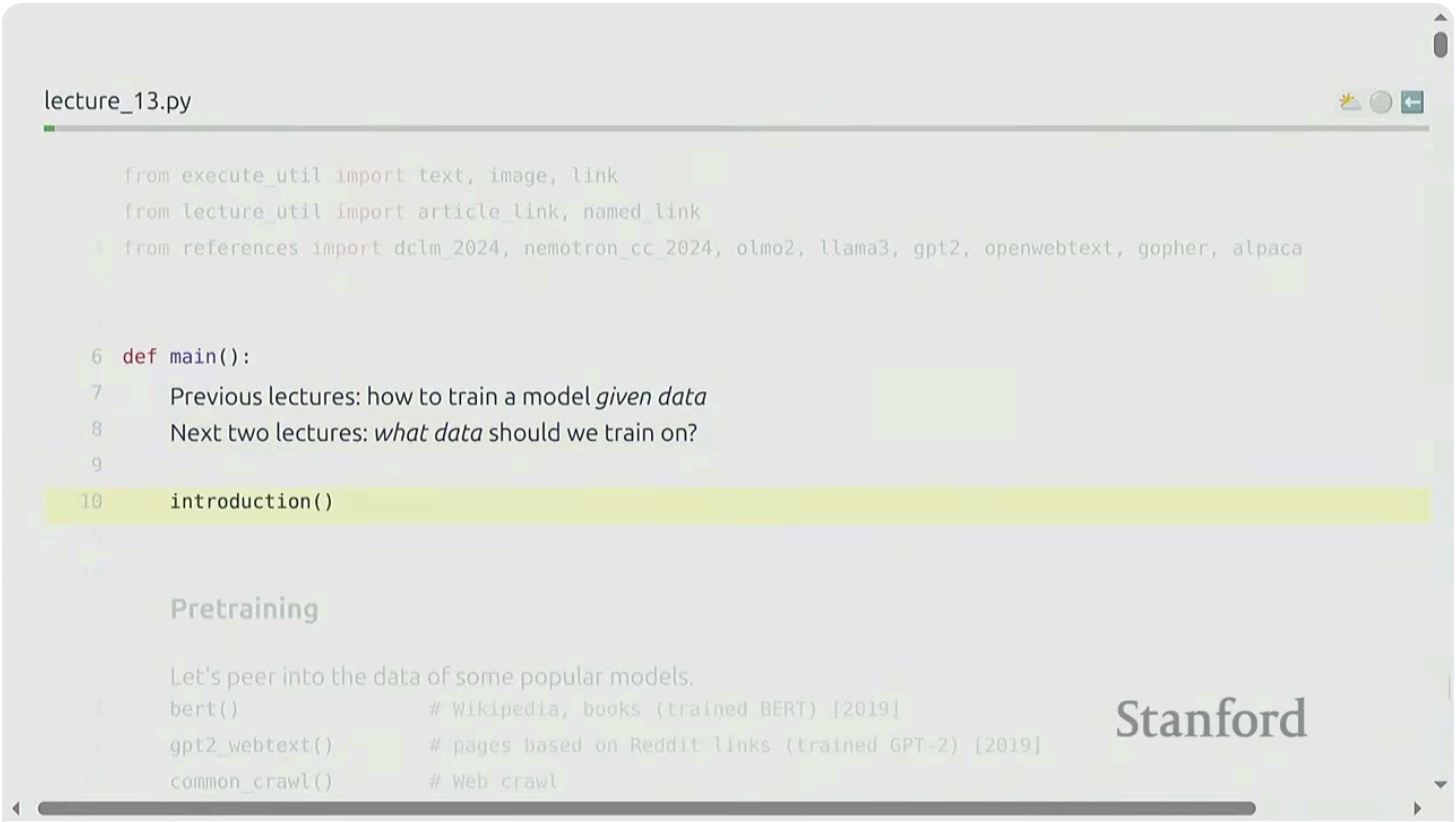
Language Identification, Quality Filtering, Toxicity Filtering, Anonymization, Synthetic Data Generation, Self-Instruct, RLHF * Copyright Law: Intellectual Property, Fair Use (purpose, nature, amount, market effect), Licenses (Creative Commons), Terms of Service * Long Context * Instruction Following / Chat Data

---

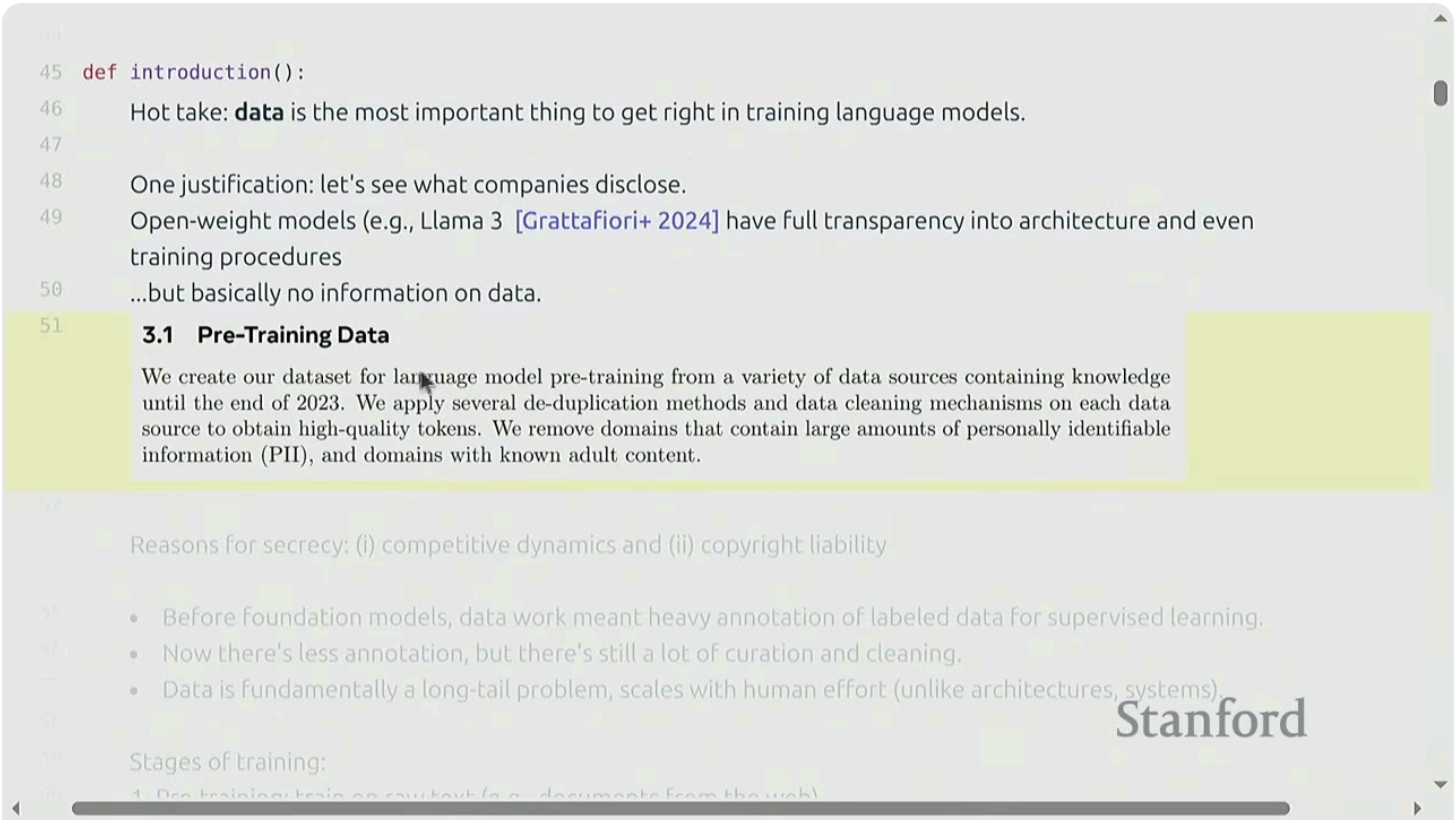
[0:00] Introduction: The Importance of Data



0:10 A lecturer stands behind a podium with a laptop, gesturing while speaking.



0:27 Python code for lecture_13.py, outlining topics on training models and data selection.



1:10 Slide section "Pre-Training Data" detailing dataset creation and data cleaning methods.

The focus of this lecture is on **data** for language models.

Hot Take: Data is the most important thing in getting language models right. * Some might disagree, arguing that scaling laws are more important. * **Justification:** Companies are often transparent about model architecture and training procedures (e.g., LLaMA 3, DeepSeek fully disclose these), but they are typically very secretive about their data. This secrecy suggests data is a key differentiator. * **Reasons for Secrecy:** 1. **Competitive Dynamics:** Data pipelines and curation methods are proprietary and give companies an edge. 2. **Copyright Liability:** Companies want to avoid lawsuits related to data usage.

Before foundation models, data annotation for supervised learning was clearly recognized as important. Now, even with less annotation, significant curation and cleaning are still required.

Data is fundamentally a "long-tail problem." * It's highly scalable: you can hire hundreds of people to work on different aspects of data (multilinguality, code, images, etc.). * In contrast, architecture design is often handled by a small, focused team.

[2:52] Stages of Training

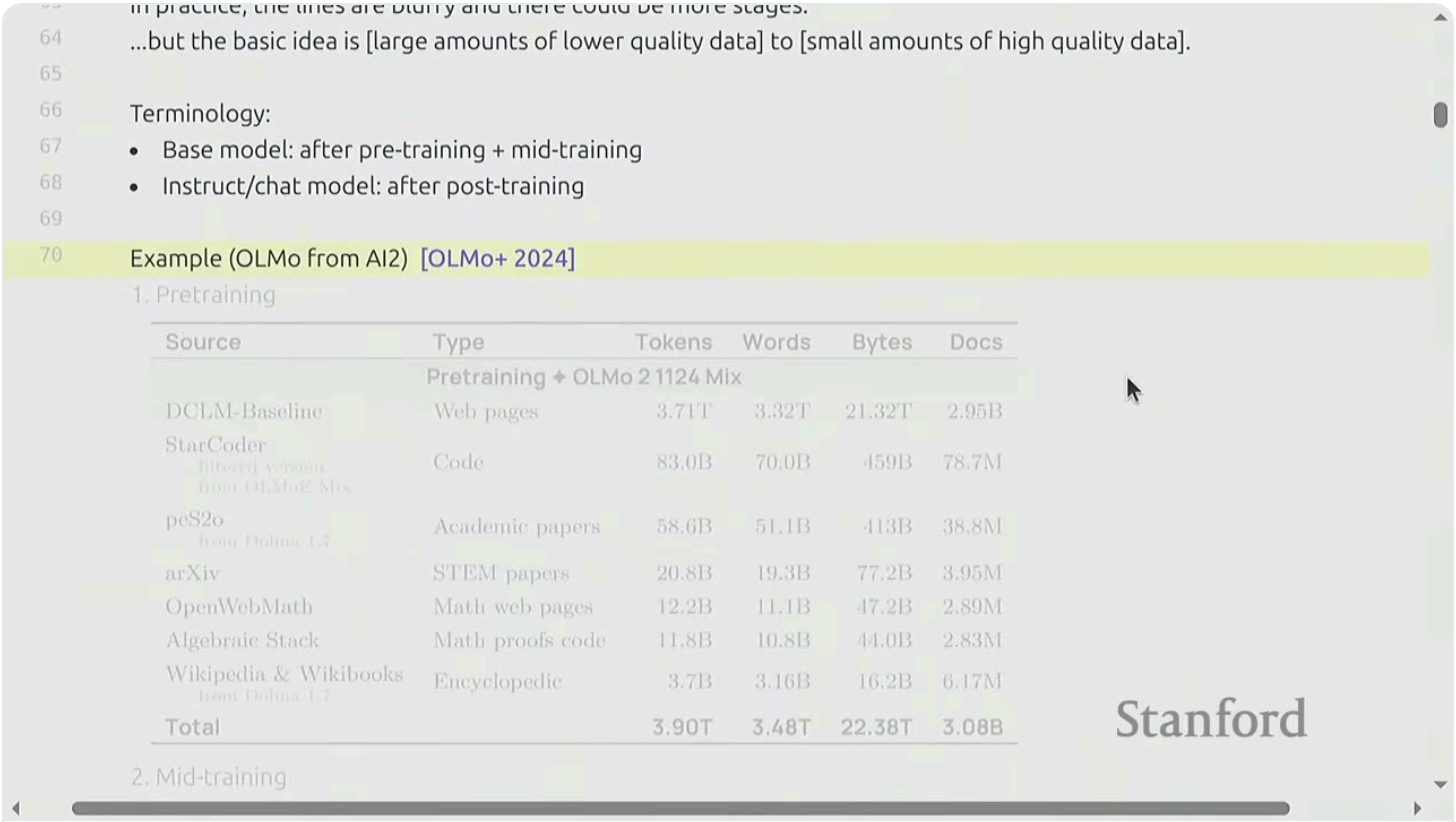
Language model training typically involves multiple stages:

1. **Pre-training**: Training on raw, large-scale data (e.g., from the web). This is the primary focus of this lecture.
2. **Mid-training**: Curating smaller sets of high-quality data (documents) to target particular capabilities (e.g., math, code, long context).
3. **Post-training**: Fine-tuning on instruction-following data, chat data, or using reinforcement learning (RLHF) to align the model with human preferences and ensure safety.

In practice, the lines between these stages can be blurry, and more stages might be involved in recent models. The basic idea is to start with large amounts of low-quality data and progressively refine training on smaller amounts of high-quality data.

Terminology: * **Base model**: Refers to the checkpoint after pre-training and mid-training. * **Instruct/Chat model**: Refers to the model after post-training.

[4:09] Example: OLMo (from AI2)



4:12 Table showing pretraining data statistics for OLMo+ 2024, including tokens, words, bytes, and docs.

Let's look at the data mix for OLMo, an open-source model from AI2 (OLMo+ 2024 paper).

1. Pre-training Data Mix:

Source	Type	Tokens (billions)	Words (billions)	Bytes (billions)	Docs (millions)
DCLM-Baseline	Web pages	3.71	3.32	21.32	2.95
StarCoder	Code	83.0	70.0	413	78.8
peS2o	Academic papers	58.6	51.1	413	38.8
arXiv	Math papers	20.8	19.3	77.2	2.89
OpenWebMath	Math web pages	12.2	11.1	47.2	3.08
Algebraic Stack	Math proofs code	1.8	1.7	6.2	0.17

Source	Type	Tokens (billions)	Words (billions)	Bytes (billions)	Docs (millions)
Wikipedia & WikiBooks	Encyclopedia	1.7	1.6	6.2	0.47
Total		**3.90T**	**3.48T**	**22.38T**	**3.08B**

- **DCLM-Baseline**: A web crawl dataset.
- **StarCoder**: Code from GitHub.
- **peS2o**: Academic papers, likely from Semantic Scholar.
- **arXiv**: Preprints, often math-heavy.
- **OpenWebMath, Algebraic Stack**: Math-specific datasets.
- **Wikipedia & WikiBooks**: Encyclopedic content.

Total: ~3.90 Trillion tokens.

2. Mid-training Data Mix: This stage focuses on higher quality data, often a subset of pre-training data or synthetically generated data.

Source	Type	Tokens (billions)	Words (billions)	Bytes (billions)	Docs (millions)
DCLM-Baseline	High quality web	752	670	4.56T	606
FLAN	Instruction data	17.0	14.4	98.2	57.3
peS2o	Academic papers	58.6	51.1	413	38.8
Wikipedia & WikiBooks	Encyclopedia	3.7	3.1	16.2	2.48
Stack Exchange	Q&A	1.26	1.14	7.72	6.17
High quality total		**832.6B**	**739.8B**	**5.09T**	**710.8M**
TuluMath	Synthetic math	230	222	1.03T	220K
TinyGSM-Mind	Synthetic math	28.7	28.1	163	17K

Source	Type	Tokens (billions)	Words (billions)	Bytes (billions)	Docs (millions)
DolminoSynthMath	Synthetic math	6.48	5.68	25.5	125
MathCoder2	Synthetic math	3.87	3.71	18.4	2.83M
Metamath	Math	84.2	76.6	37.4	7.8K
CodeSearchNet	Code	1.78	1.41	29.8	7.2K
GSM8K	Math	2.74	2.59	9.73	21.37M
Train split		**10.7B**	**9.73B**	**45.9B**	**21.37M**

- Notice the DCLM-Baseline is filtered down from 3.7T to 752B tokens.
- FLAN: Instruction data.
- Stack Exchange: Q&A data.
- Synthetic math data (TuluMath, TinyGSM-Mind, DolminoSynthMath, MathCoder2).
- Metamath, CodeSearchNet, GSM8K: Math and code specific datasets.

3. Post-training Data Mix: This stage focuses on instruction following and safety.

Category	Prompt Dataset	Count	# Prompts used in SFT	# Prompts used in DPO	# References
General	Tulu 3 Hardcoded	88	24	7	24
	OpenAssistant 1.2.1	24	7,130	7,130	24
	No Robots	9,500	9,500	9,500	24
	WildChat (GPT-4 subset)	89,892	89,892	0	24
	Ultrafeedback v2	41,635	41,635	0	24
	FLAN v2.1	89,982	89,982	0	24
	SciRIFF	15,000	15,000	0	24
Recall	Tulu 3 Persona Math	149,960	149,960	0	24
Table QA	Tulu 3 Persona Math	149,960	149,960	0	24

Category	Prompt Dataset	Count	# Prompts used in SFT	# Prompts used in DPO	# References
Reasoning	Tulu 3 GSM	19,980	19,980	0	24
	Tulu 3 Persona Algebra	19,980	19,980	0	24
	OpenMathInstruct-2	64,312	64,312	0	24
Coding	NumaMath-IR	20,000	20,000	0	24
	Tulu 3 CodeCoCo	10,000	10,000	0	24
Safety-Compliance	Tulu 3 WildJailbreak	50,000	50,000	0	24
& Non-Compliance	Tulu 3 WildGuardMix	50,000	50,000	0	24
Multilingual	Aya	29,980	29,980	0	24
Precise IF	Tulu 3 Persona IF	19,980	19,980	0	24
Augmented IF	Tulu 3 IF-augmented	65,530	65,530	0	24
Total		**939,927**	**939,927**	**425,147**	

This stage involves various chat-like datasets and synthetically generated data for specific capabilities.

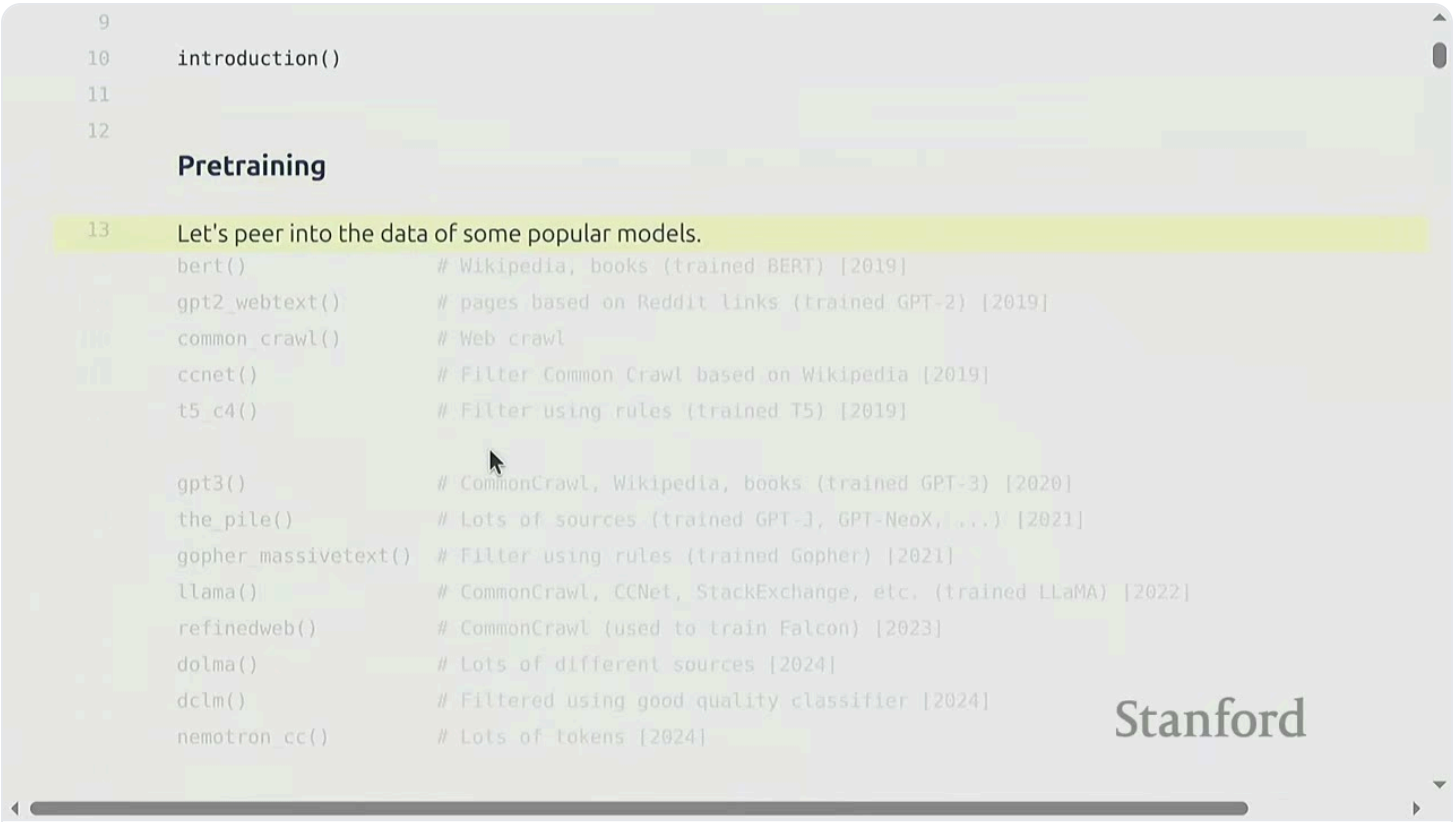
[5:40] What are these datasets? How are they chosen and processed?

There isn't a formal, principled way to decide these things. It's often heuristic.

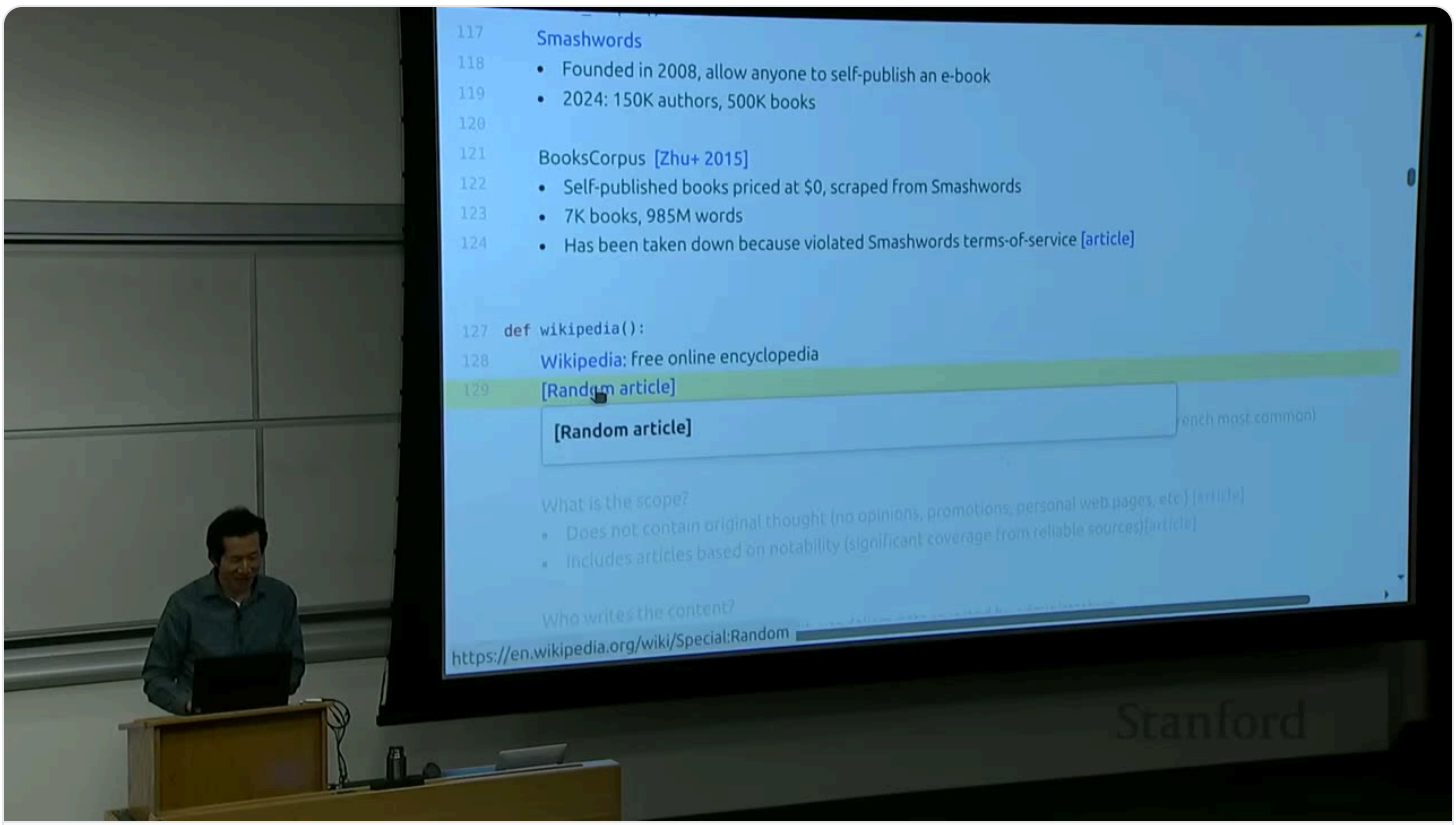
[6:15] Pre-training: Data of Popular Models

Let's peer into the data of some popular models, starting from earlier models and moving to more recent ones.

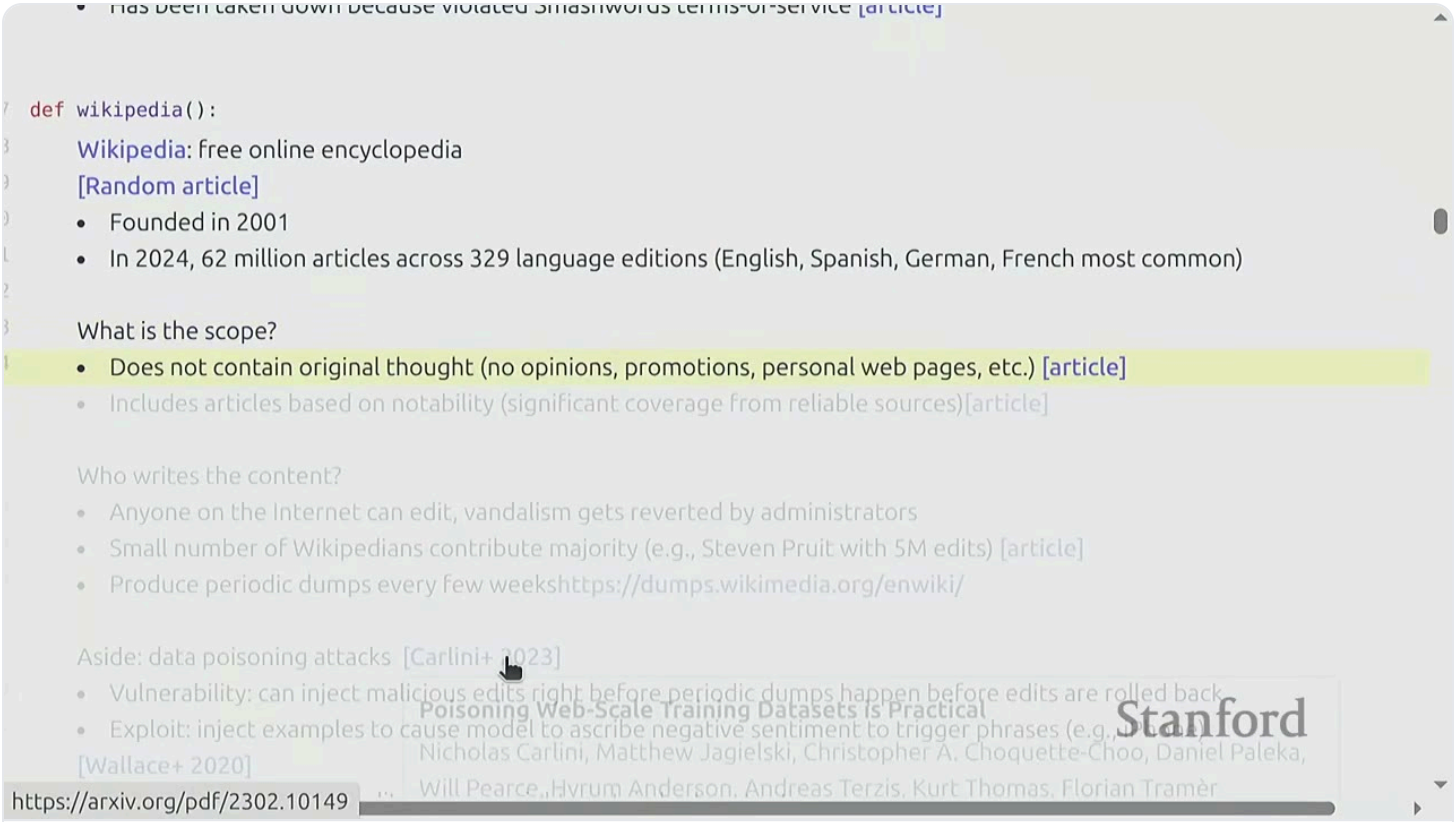
[6:50] BERT (Devlin+ 2018)



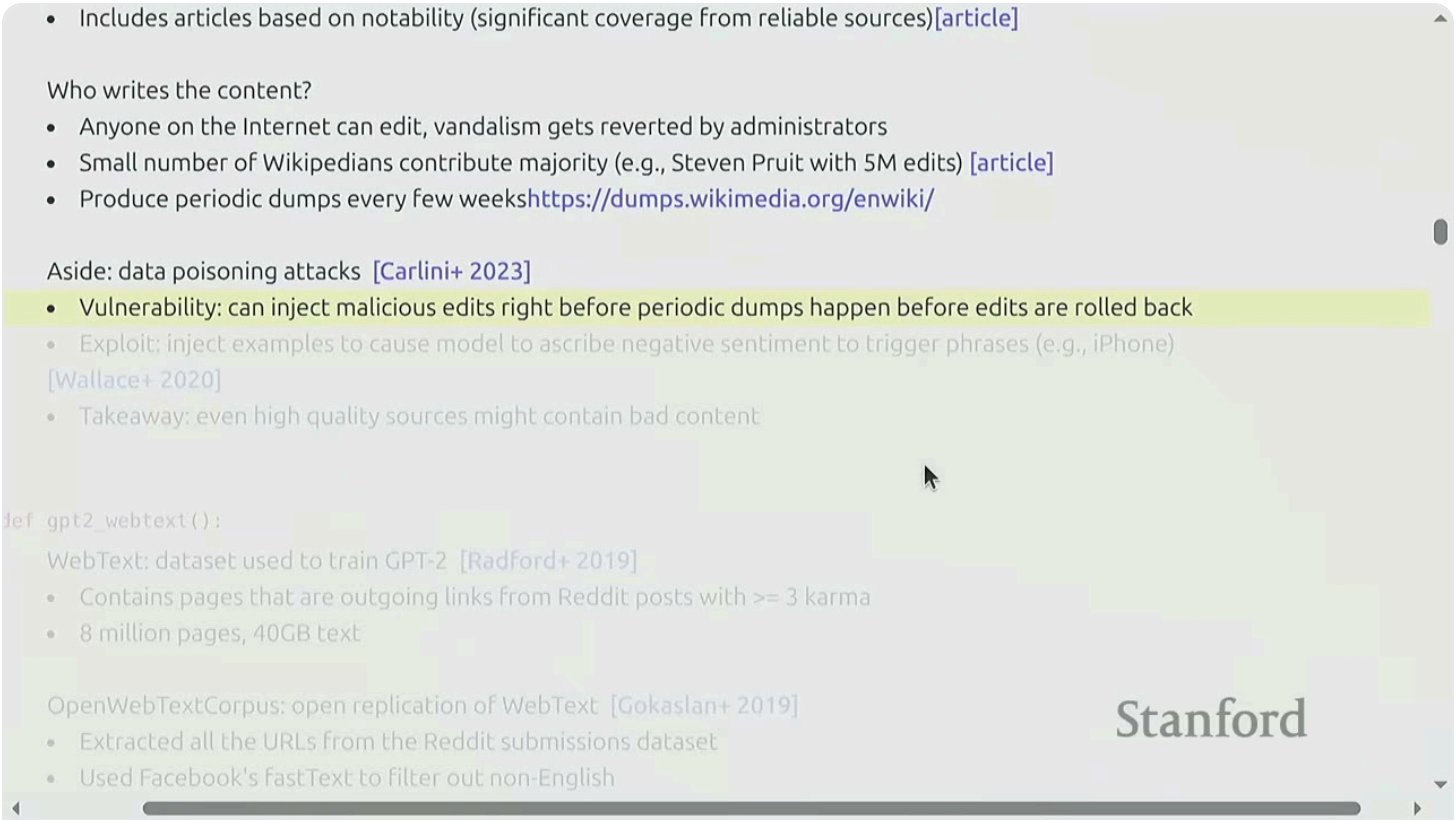
6:50 Python code snippet listing popular models and their data sources for pretraining.



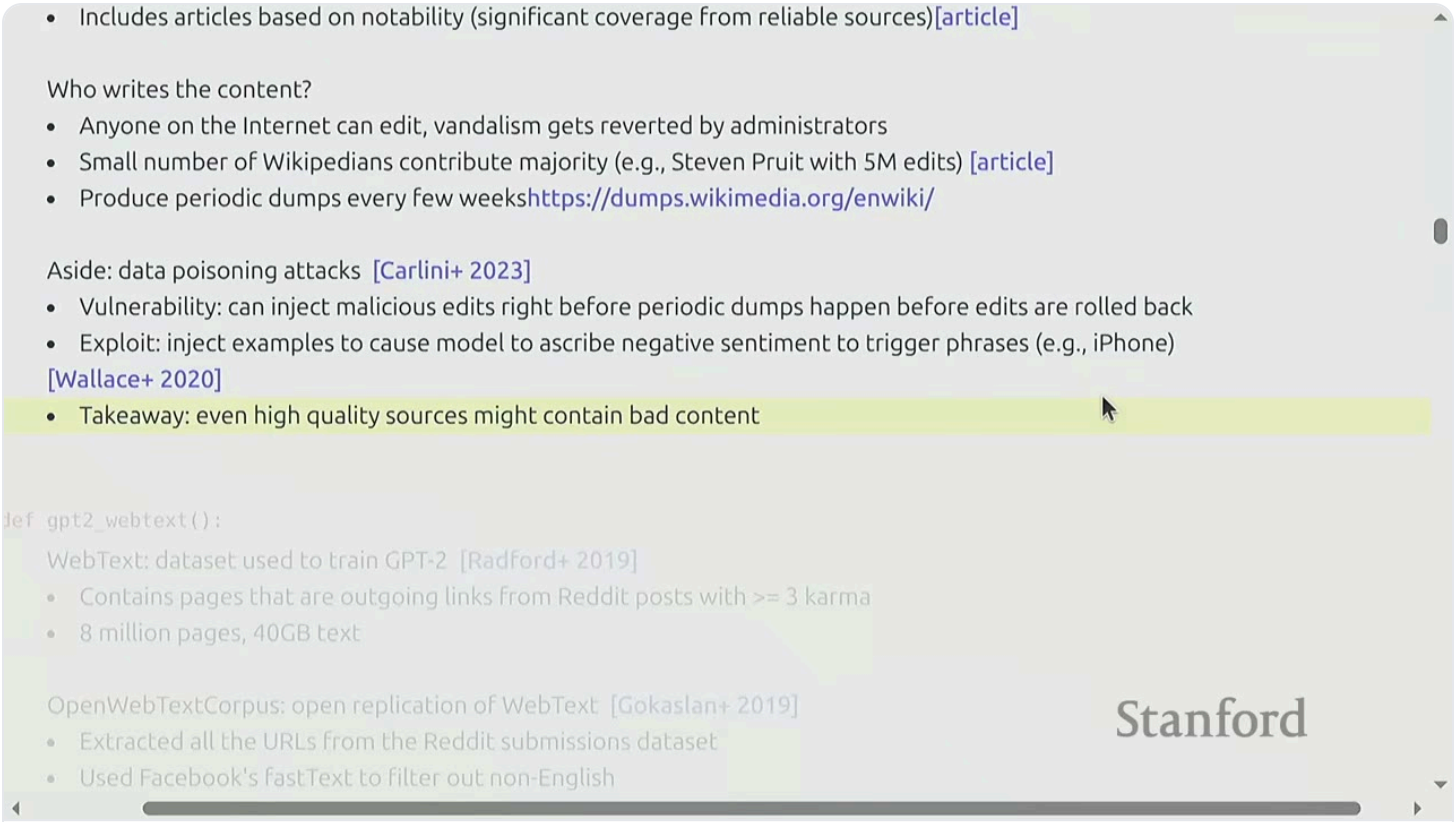
8:36 Lecturer at podium, with a slide showing details about Smashwords, BooksCorpus, and Wikipedia.



9:01 Slide detailing Wikipedia's scope, content, and data poisoning attacks.



10:47 Slide continuing discussion on data poisoning attacks, specifically vulnerability and exploit examples.



11:33 Slide concluding data poisoning attacks with a takeaway: even high-quality sources might contain bad content.

BERT was trained on: * `books_corpus()` * `wikipedia()`

`books_corpus()` * **Smashwords**: Founded in 2008, allowed anyone to self-publish an e-book. By 2024, there were ~150,000 authors and 500,000 books. * **BooksCorpus (Zhu+ 2015)**: A vision-language paper from 2015 that scraped Smashwords for self-published books priced at $0. It contained 7,000 books and 985 million words. * **Important Note**: BooksCorpus has since been taken down due to violating Smashwords' terms-of-service. In 2015, AI copyright wasn't a major concern.

`wikipedia()` * Free online encyclopedia, founded in 2001. * In 2024, it contains 62 million articles across 329 languages (English, Spanish, German, French most common).

What is the scope of Wikipedia? * Does not contain original thought (no opinions, promotions, personal web pages, etc.). Everything is cited from primary sources. * Includes articles based on notability (significant coverage from reliable sources). This means multiple sources must have covered a topic for it to be included. * **Who writes the content?** Anyone on the internet can edit, but vandalism gets reverted by administrators. A small number of Wikipedians contribute the majority (e.g., Steven Pruitt with 5M edits). * **Periodic Dumps**: Wikipedia produces periodic

dumps every few weeks (e.g., `https://dumps.wikimedia.org/enwiki/`). You can download these dumps as zip files.

[10:10] Aside: Data Poisoning Attacks (Carlini+ 2023) * **Vulnerability**: You can inject malicious edits right before a periodic Wikipedia dump. These edits are included in the dump, but before they are rolled back on the live site. * **Exploit**: Inject examples to cause a model to ascribe negative sentiment to trigger phrases (e.g., "iPhone"). * **Takeaway**: Even high-quality sources might contain bad content. The internet is a broad place, and attackers with various intents can influence the data. It's hard to have oversight.

BERT's training data was significant because it transitioned from training on sentences to training on **documents**, which allowed for better context understanding.

[12:19] GPT-2 WebText (Radford+ 2019)

- **WebText**: Dataset used to train GPT-2.
- **Insight**: Reddit posts often link out to external pages. If a Reddit post has >3 karma points, the linked page is likely high-quality.
- This method yielded 8 million pages, 40 GB of text.
- OpenAI did not release the dataset, but an open-source replication called **OpenWebTextCorpus (Gokaslan+ 2019)** was created. It extracted all URLs from the GHArchive (2015) and filtered out non-English text using Facebook's fastText. Duplicates were removed.

[13:29] Common Crawl (Web Crawl)

- Used Facebook's fastText to filter out non-English
- Removed near duplicates

```
def common_crawl():
```

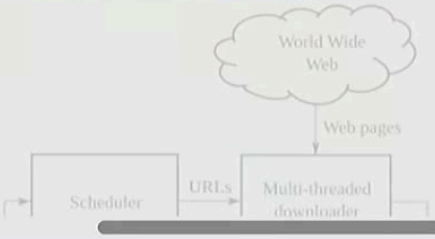
Common Crawl is a non-profit organization founded in 2007

Statistics

- Every ~month, run a web crawl
- So far, there have been ~100 crawls from 2008-2025
- In 2016, crawl takes 10-12 days on 100 machines [article]
- Latest crawl: April 2025https://commoncrawl.org/blog/april-2025-crawl-archive-now-available
- Crawls have some overlap but try to diversify

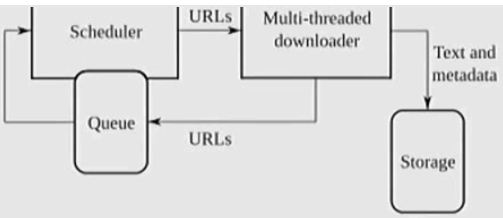
Crawling

Uses Apache Nutch [article]



Stanford

15:06 Slide on Common Crawl statistics, including frequency, duration, and overlap of crawls.



- Starts with a set of seed URLs (at least hundreds of millions) [article]
- Download pages in a queue and add hyperlinks to queue

Policies [article]

- Selection policy: which pages to download?
- Politeness policy: respect robots.txt, don't overload server
- Re-visit policy: how often to check if pages change
- Challenge: URLs are dynamic, many URLs lead to basically same content

Two formats

- WARC: raw HTTP response (e.g., HTML)
- WET: converted to text (lossy process)

HTML to text

- Tools to convert HTML to text: trafilatura, resiliparse
- DCLM paper shows that the conversion matters for downstream task accuracy: [Li+ 2024]

Stanford

19:16 Diagram of a web crawler's architecture, showing scheduler, queue, downloader, and storage components.

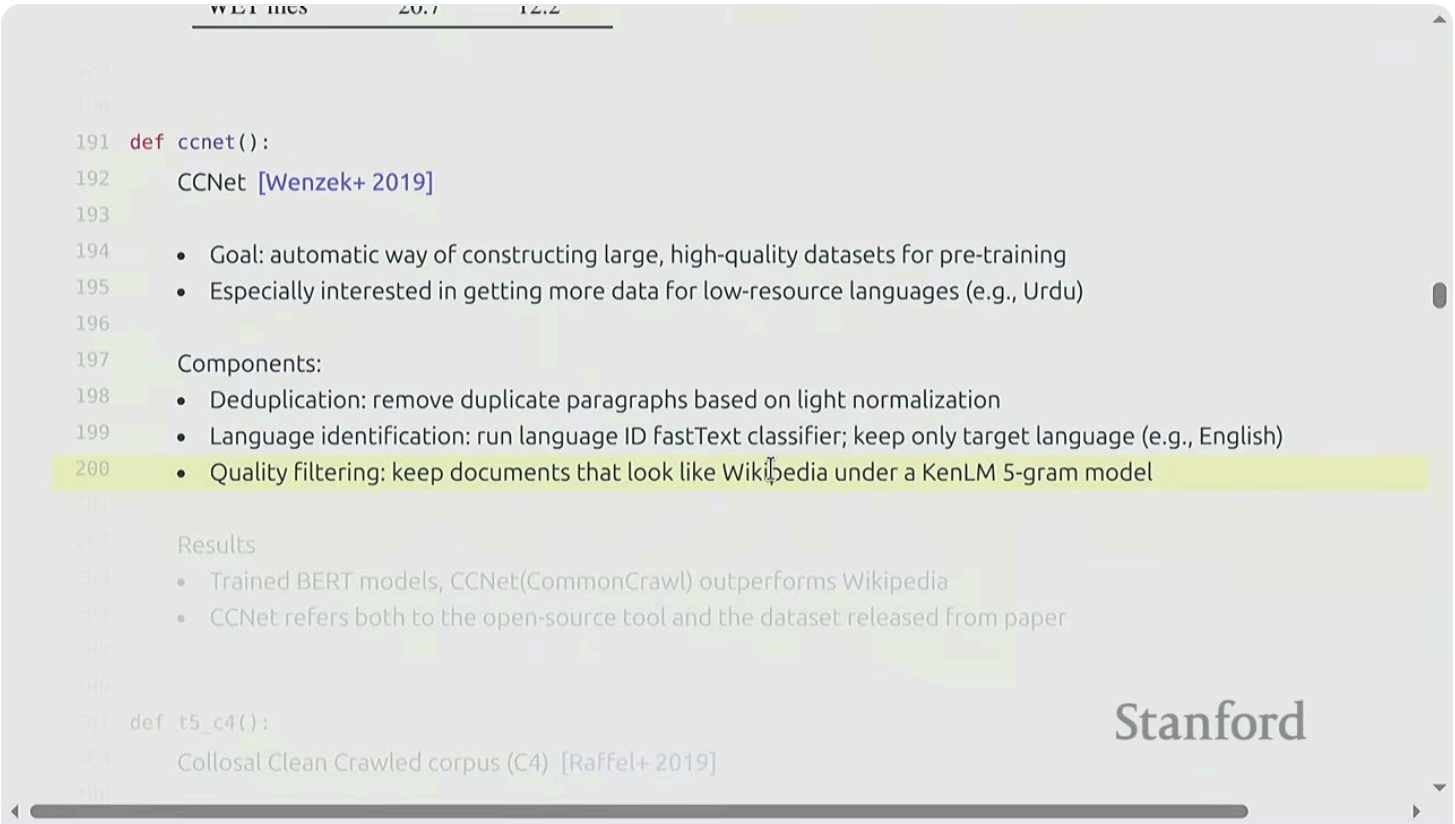
Common Crawl is an academic approximation of the internet. * **Statistics**: * Non-profit organization founded in 2007. * Runs a web crawl every month. * Over 100 crawls from 2008-2025. * In 2016, a crawl took 10-12 days on 100 machines. * Latest crawl: April 2025 (available at `https://commoncrawl.org/blog/april-2025-crawl-archive-now-available`). * Each crawl has some overlap, but attempts are made to diversify. * The April 2025 crawl captured 2.74 billion web pages (468 TiB of uncompressed content).

- **Crawling (Uses Apache Nutch)**:
    - Starts with a set of seed URLs (at least hundreds of millions).
    - Downloads pages in a queue and adds hyperlinks to the queue (BFS-like).
    - **Policies**:
        - **Selection policy**: Which pages to download?
        - **Politeness policy**: Respect `robots.txt`, don't overload the server.
        - **Re-visit policy**: How often to check if pages change.
        - **Challenge**: URLs are dynamic, many URLs lead to basically same content (duplication).
- **[18:10] Q: Does Common Crawl filter offensive content?**
    - A: By default, Common Crawl is very permissive. The idea of "offensive" is a high-level semantic decision. There's likely a lot of offensive/harmful content. Some illegal sites might be blacklisted, but generally, it's broad.
- **[19:00] Q: Can a website opt-out of Common Crawl?**
    - A: Yes, by including a `robots.txt` file, which specifies which crawlers are allowed/disallowed. However, `robots.txt` is a guideline, not a formal enforcement mechanism. Some crawlers might not respect it.
    - Most major LLM developers use their own crawlers because Common Crawl, despite its size, is quite sparse in terms of coverage.
- **[20:40] Q: How are images/media handled?**
    - A: Common Crawl technically gets the raw HTTP response. Sometimes it's text, sometimes it's images. Most of Common Crawl is biased towards text, but it can contain other media. Specialized crawlers can be developed for specific media types.
- **[21:15] Q: What fraction of Common Crawl is copyrighted?**

    - A: Most of it. This is a complex topic that will be discussed later.

- **Two formats for Common Crawl data**:
    - **WARC**: Raw HTTP response (e.g., HTML).
    - **WET**: Converted to text (lossy process).
- **HTML to text**: Tools like `trafilatura` and `resiliparse` convert HTML to text. The choice of tool matters for downstream task accuracy. Using `trafilatura` can yield significantly better results than using pre-converted WET files.

[21:50] CCNET (Wenzek+ 2019)



23:09 Slide on CCNet, its goal of constructing high-quality datasets, and components like quality filtering.

CCNet is an automatic way of constructing large, high-quality datasets for pre-training, especially for low-resource languages (e.g., Urdu).

Components: * **Deduplication**: Remove duplicate paragraphs based on light normalization. * **Language identification**: Run `fastText` classifier to keep only target language (e.g., English). * **Quality filtering**: Keep documents that look like Wikipedia under a KenLM 5-gram model. * This is a key insight: Wikipedia serves as a surrogate for high-quality data. * This approach helps filter

for well-structured, factual content. However, Wikipedia doesn't cover everything (e.g., opinions, recipes).

Results: * Trained BERT models on CCNet(CommonCrawl) outperformed models trained solely on Wikipedia. * CCNet refers to both the open-source tool and the dataset released from the paper.

[23:45] T5 C4 (Raffel+ 2019)

C4 stands for Colossal Clean Crawled Corpus. This paper is more famous for Text-to-Text Transfer Transformer (T5), but C4 was a major contribution.

Observation: Common Crawl is mostly not useful natural language.

Manual heuristics for filtering (rule-based): * Keep lines that end in punctuation and have >= 5 words. * Remove pages with fewer than 3 sentences. * Remove pages that contain any "bad words" (list not shown). * Remove pages containing `{` (no code), `lorem ipsum`, `terms of use`, etc. (This removes a lot of code, which is interesting). * Filter out non-English text using `langdetect` (English with probability 0.99).

End result: 806 GB of text (156 billion tokens).

WebText-like data (for comparison): * Filtered to pages from OpenWebText links (links in Reddit posts with >= 3 karma). * Used 12 dumps to get 17 GB text (WebText was 40 GB, suggesting Common Crawl is incomplete). * This improved on various NLP benchmarks (GLUE, SQuAD, etc.).

[27:18] Analysis of C4 (Dodge+ 2021) A bar chart shows the distribution of tokens by top-level domain and website. * `.com` and `.org` are prominent top-level domains. * `patents.google.com`, `en.wikipedia.org`, `news.google.com`, `books.google.com` are among the top websites. * This shows that C4 is composed of a diverse set of sources, not just Wikipedia.

[27:42] GPT-3 Dataset (Brown+ 2020)

The GPT-3 dataset was trained on: * `Common Crawl (processed)` * `WebText2` (expanded with more links) * `Mysterious Internet-based books corpora (Books1, Books2)` * `Wikipedia`

Result: 570 GB (400 billion tokens).

Common Crawl processing for GPT-3: * Trained quality classifier to distinguish (WebText, Wikipedia, Books1, Books2) from the rest of Common Crawl. * Fuzzy deduplication of documents (including WebText and benchmarks). * **Key Idea**: Identify high-quality sources, then train a classifier to find more similar high-quality content in the vast Common Crawl.

[29:05] The Pile (Gao+ 2020)

The Pile was created in reaction to GPT-3's closed nature, as part of an effort to produce open-source language models. It was a grassroots effort with many volunteers contributing/coordinating on Discord.

Curated 22 high-quality domains: A treemap shows the composition of The Pile by category (Academic, Internet, Prose, Dialogue, Misc). * **Academic**: PubMed Central, ArXiv, USPTO, NIH, PMA, FreeLaw. * **Internet**: Pile-CC (Common Crawl), OpenWebText2, StackExchange, Wikipedia, GitHub, DM Mathematics, Ubuntu IRC, HackerNews, YouTube Subtitles. * **Prose**: Books3, PG-19, BookCorpus2, EuroParl, NIH Exporter, Enron Emails. * **Dialogue**: OpenSubtitles, MusicBrainz. * **Misc**: Bibliotik, BC2.

A table provides more statistics:

Component	Raw Size (GiB)	Weight (%)	Epochs	Effective Size (GiB)	Mean Document Size (KiB)
Pile-CC	227.12	18.11	1.0	227.12	4.33
PubMed Central	90.27	14.40	2.0	180.55	30.5
Books3	100.96	12.07	1.5	151.44	3.68
OpenWebText2	62.77	10.01	2.0	125.54	6.39
ArXiv	95.16	7.59	1.0	95.16	8.96
GitHub	56.21	8.96	1.5	84.31	7.29
FreeLaw	51.15	8.16	1.5	76.73	6.87
StackExchange	32.20	5.13	2.0	64.40	8.88
USPTO Backgrounds	19.26	3.07	2.0	38.53	1.97
PubMed Abstracts	22.90	3.65	1.5	34.35	1.89

Component	Raw Size (GiB)	Weight (%)	Epochs	Effective Size (GiB)	Mean Document Size (KiB)
Gutenberg (PG-19)	18.63	2.97	1.5	27.94	1.62
OpenSubtitles	17.08	2.72	1.5	25.62	1.11
DM Mathematics	6.38	1.55	3.0	19.13	0.89
Ubuntu IRC	7.75	1.24	2.0	15.50	5.91
BookCorpus2	6.30	0.75	1.5	9.45	68.9
EuroParl	4.59	0.73	1.5	6.88	6.30
HackerNews	3.90	0.62	2.0	7.80	3.73
YouTube Subtitles	3.73	0.60	2.0	7.46	1.89
NIH Exporter	2.38	0.38	2.0	4.76	1.44
Enron Emails	0.88	0.14	2.0	1.76	1.78
The Pile	**825.18**			**1254.20**	**5.91**

Result: 825 GB of text (~275B tokens).

- **Pile-CC**: Common Crawl, use WARC, `jusText` to convert into text (WET).
- **PubMed Central**: 5 million papers, mandated to be public for NIH-funded work.
- **ArXiv**: Preprints for research papers since 1991 (use LaTeX).
- **Enron Emails**: 500k emails from Enron senior management, released during Enron investigation (2002).
    - **Reasoning**: Email datasets are rare due to privacy. Enron emails provide a unique source of real-world communication. This might introduce bias in the model's understanding of email.
- **Project Gutenberg**: Started in 1971 by Michael Hart, wanted to increase access to literature. ~75k books, mostly English. Only includes books that have received copyright clearance (mostly in the public domain). PG-19 is a subset of these books.
- **Books3 (Presser, 2020)**: 196k books from the shadow library `Bibliotik`. Contained books from authors like Stephen King, Min Jin Lee, Zadie Smith. Has been taken down due to copyright infringement/lawsuits.

- **Shadow Libraries**: Examples include Library Genesis (LibGen), Z-Library, Anna's Archive, Sci-Hub. These disregard copyright and bypass paywalls. They are often hosted on servers in various countries to circumvent legal controls. Proponents argue they make freely available what should be free. LibGen has ~4M books (2019), Sci-Hub has ~88M papers (2022). Meta has trained models on LibGen.
- **StackExchange**: Collection of sites of user-contributed questions and answers. Started with StackOverflow in 2008, grew to other topics (e.g., math, literature). Uses reputation points and badges to incentivize participation.
    - **Q&A format**: This data is close to instruction tuning/real applications.
    - **Metadata**: Users, votes, comments, badges, tags for filtering are available.
    - **Data dumps**: In XML (anonymized, include metadata).
- **GitHub**: Code is helpful for programming tasks, but also for reasoning (folklore). GitHub started in 2008, acquired by Microsoft in 2018.
    - **Random repository**: A random GitHub repository might not be representative of high-quality code.
    - **Contents of a repository**: A directory, not all is code. Metadata (users, issues, commit history, pull request comments, etc.) is available. Lots of duplicates (e.g., copied code, forks, etc.).
    - **GH Archive**: Hourly snapshots of GitHub events (commits, forks, tickets, commenting). Available on Google BigQuery.
- **The Stack (Kocetkov+ 2022)**: Took repository names from GHArchive (2015-2022). Git cloned 137M repositories, 51B files (5B unique!). Kept only permissively licensed (MIT, Apache) using `go-license-detector`. Removed near-duplicates using `minhash` and `Jaccard similarity`. Result: 3.1 TB of code.
    - **Key Insight**: Code often has clearer licenses than web pages.
    - **Takeaway**: When someone says they trained on "GitHub," ask for specifics about their preprocessing steps.

[4:26] LLaMA (Touvron+ 2023)

Dataset for LLaMA: * `CommonCrawl` processed with `CCNet`, classify references of Wikipedia or not. * `C4` (more diverse; recall: rule-based filtering). * `GitHub`: Kept permissive licenses, filtering based on manual rules. * `Wikipedia`: June-August 2022, 20 languages, manual filtering. * `Project Gutenberg` and `Books3` (from The Pile). * `ArXiv`: Removed comments, inline expanded macros, bibliography. * `Stack Exchange`: 28 largest websites, sorted answers by score.

Result: 1.2 Trillion tokens. * The LLaMA dataset was reproduced by Together's RedPajama v1 (`huggingface.co/datasets/togethercomputer/RedPajama-Data-1T`). * Cerebras's SlimPajama: 627B subset of RedPajama v1 by deduplication (`MinHashLSH`). * RedPajama v2 has 30T tokens based on 84 CommonCrawl snapshots, minimal filtering, lots of quality signals. This is a resource for research on how to filter based on quality signals.

[4:38] RefinedWeb (Penedo+ 2023)

Point: Web data is all you need. * **Thesis**: If you do a good enough job filtering web data, it's sufficient. * **Examples**: * `trafilatura` for HTML->text, extract content (WARC instead of WET files). * Filtering: Gopher rules, avoid ML-based filtering to avoid biases. * Fuzzy deduplication using `MinHash` over 5-grams. * Release 600B (out of 5T) tokens.

[4:46] FineWeb (HuggingFaceFW/fineweb)

- Started as a replication of RefinedWeb, but improved it.
- Used 95 Common Crawl dumps.
- URL filtering, language ID (keep if p(en) > 0.65).
- Filtering: Gopher, C4, more manual rules.
- Fuzzy deduplication using `MinHash` .
- Anonymize email and public IP addresses (PII).

Result: 15 Trillion tokens. * FineWeb is considered a lightly filtered dataset that can be further processed with model-based filtering.

[4:52] Dolma (Soldaini+ 2024)

AI2's initial OLMo model was trained on the Dolma dataset.

Source	Doc Type	UTF-8 bytes (GB)	Documents (millions)	Unicode words (billions)	Llama tokens (billions)
Common Crawl	web pages	9,022	3,370	1,775	2,760
The Stack	code	1,043	210	260	411
C4	web pages	790	364	153	228
Reddit	social media	339	77	72	89

Source	Doc Type	UTF-8 bytes (GB)	Documents (millions)	Unicode words (billions)	Llama tokens (billions)
PeS2o	STEM papers	268	30	68	70
Project Gutenberg	books	20.4	0.056	3.7	4.3
Wikipedia, Wikibooks	encyclopedia	16.2	6.2	3.7	4.3
Total		**11,519**	**4,367**	**2,318**	**3,059**

- **Reddit**: From the Pushshift project (2005-2023), includes submissions and comments separately. This project is now defunct.
- **PeS2o**: 40M academic papers from Semantic Scholar.
- **Common Crawl processing**:
  - Language identification (`fastText` classifier), keep English.
  - Quality filtering (Gopher, C4 rules), avoid model-based filtering.
  - Toxicity filtering using rules and Jigsaw classifier.
  - Deduplication using Bloom filters.

Result: 3 Trillion tokens.

[5:09] DCLM (Li+ 2024)

DataComp-LM: In search of the next generation of training sets for language models. * **Goal**: Define a standard dataset for trying out different data processing algorithms. * Processed CommonCrawl to produce DCLM-pool (240T tokens). * DCLM-baseline: filtered down DCLM-pool using quality classifier.

[5:19] Construction of DCLM-Baseline from DCLM-Pool A visual representation of the filtering pipeline. * **Heuristic cleaning (sections 4.1 & 4.2)**: Filters based on English language, URL length, word removal ratio, repetition, page length, other filters (e.g., word-length, ellipsis count, stop words). This reduces the data significantly. * **Deduplication (4.3)**: Further reduces data. * **Model-based filtering (4.4)**: Uses `fastText` classifier to filter data. This is the most aggressive step, reducing the data to 1.4% of its original size.

Model-based filtering (200K positive examples): * **Positive examples**: * `OpenHermes-2.5` : mostly GPT-4 generated instruction data (examples). * `ELI5` : subreddit with curiosity questions and answers (examples). * `RefinedWeb` . * **Negative examples**: `RefinedWeb` . * **Result**: 3.8 Trillion tokens.

Quality filtering comparison (1B-1x scale): A table compares various filtering methods based on model-based quality filtering. Training a `fastText` classifier for filtering performs best.

Filter	CORE	EXTENDED
RefinedWeb reproduction	27.5	14.6
Top 20% by Pagerank	26.1	13.9
Classifier on BGE features	27.2	14.0
Perplexity filtering	28.6	15.0
AskLLM (146)	29.0	14.3
SemDedup (1)	29.2	14.7
`fastText` (87) OH-2.5 +ELI5	**30.2**	**15.4**

- This shows that using a `fastText` classifier (model-based filtering) outperforms other filtering methods.
- After this paper, AI2's OLMo 2 started training on DCLM-baseline. This indicates a shift towards model-based filtering for quality. The idea of avoiding model-based biases has largely been replaced by the pursuit of better benchmark scores.

[5:11] NemoTron-CC (Su+ 2024)

NemoTron-CC: Transforming Common Crawl into a Refined Long-Horizon Pretraining Dataset. This came out of NVIDIA.

Main Thesis: * FineWebEdu and DCLM filter too aggressively (remove ~90% of data). * Need more tokens (but keep quality up).

Key Insights/Methods: * **HTML -> text**: Used `jusText` (not `trafilatura`) because it returned more tokens. The goal is to maximize tokens while maintaining quality. * **Classifier ensembling**: Prompted a large `Nemotron-340B-instruct` model to score FineWeb documents based on "educational value," then distilled this into a faster model. This is combined with a DCLM

classifier. * **Synthetic data rephrasing**: * For low-quality data, use an LLM to rephrase it into higher-quality data. * For high-quality data, use an LLM to generate tasks (QA pairs, extract key information, etc.).

Result: 6.3 Trillion tokens (HQ subset is 1.1T). * For reference, LLaMA 3 trained on 15T, Qwen 3 trained on 36T. * 6.3T tokens is a substantial amount for open-source models. * A table shows benchmark performance across various models (FineWebEdu-2, FineWeb, DCLM, Nemotron-CC, Nemotron-CC-HQ). Nemotron-CC-HQ achieves the highest average score (60.1).

[5:41] Copyright

[5:45] Lots of lawsuits around generative AI, mostly around copyright.

Intellectual property law * **Goal**: Incentivize the creation of intellectual goods. * **Types**: Copyright, patents, trademarks, trade secrets.

Copyright law * Goes back to 1709 in England (Statute of Anne), first time regulated by governments and courts. * In United States, most recent: Copyright Act of 1976. * **Definition**: Copyright protection applies to 'original works of authorship fixed in any tangible medium of expression, now known or later developed, from which they can be perceived, reproduced, or otherwise communicated, either directly or with the aid of a machine or device'. * **Original works**: Collections are not copyrightable (e.g., telephone directories) unless there is some creativity in the selection or arrangement. * **Applies to expression, not ideas**: You can't copyright an idea (e.g., quicksort algorithm), but you can copyright its specific expression (e.g., the code). * **Expanded scope**: From 'published' (1909) to 'fixed' (1976). This means copyright applies as soon as a work is fixed in a tangible medium, regardless of publication. * **Registration not required**: Unlike patents, copyright is automatic upon creation. * **Threshold for copyright is extremely low**: Your website is copyrighted as soon as it's created. * **Registration is required before creator can sue**: Costs $65 to register. * **Duration**: Lasts for 75 years, then the copyright expires and it becomes part of the public domain (e.g., works of Shakespeare, Beethoven, most of Project Gutenberg).

Summary: Most things on the internet are actually copyrighted.

How to use a copyrighted work: 1. Get a license for it. 2. Appeal to the fair use clause.

Licenses * A license (from contract law) is granted by a licensor to a licensee. Effectively, 'a license is a promise not to sue'. * **Creative Commons license**: Enables free distribution of

copyrighted work. * Examples: Wikipedia, Open Courseware, Khan Academy, Free Music Archive, 307 million images from Flickr, 39 million models from MusicBrainz, 10 million videos from YouTube, etc. * Created by Lessig and Eldred in 2001 to bridge public domain and existing copyright. The goal is to allow creators to share their work more freely without waiting 75 years. * **Many model developers license data for training foundation models**: Google and Reddit, OpenAI and Shutterstock, OpenAI and StackExchange.

Fair use (section 107) Four factors to determine whether fair use applies: 1. **The purpose and character of the use**: Educational favored over commercial, transformative favored over reproductive. * **Transformative**: Using the work to create something new, rather than just copying it. 2. **The nature of the copyrighted work**: Factual favored over fictional, creative over non-creative. 3. **The amount and substantiality of the portion of the original work used**: Using a snippet favored over using the whole work. 4. **The effect of the use upon the market (or potential market) for the original work**.

Examples of fair use: * Watching a movie and writing a summary of it. * Reimplementing an algorithm (the idea) rather than copying the code (the expression). * Google Books index and show snippets (Authors Guild v. Google 2002-2013): Ruled in favor of Google.

Copyright is not about verbatim memorization. * Plots and characters (e.g., Harry Potter) can be copyrightable. * Parody is likely fair use. * Copyright is about semantics (and economics).

Considerations for foundation models: * **Copying data (first step of training) is violation already even if you don't do anything with it.** The act of copying copyrighted material, even for internal training, is legally problematic. * **Training an ML model is transformative (far from just copy/pasting).** This is a key argument for fair use. * **ML system is interested in idea (e.g., stop sign), not in the concrete expression (e.g., exact artistic choices of a particular image of a stop sign).** This is another argument for fair use. * **Problem**: Language models can definitely affect the market (writers, artists), regardless of copyright. This is a major concern for creators.

Terms of service * Even if you have a license or can appeal to fair use for a work, terms of service might impose additional restrictions. * **Example**: YouTube's terms of service prohibits downloading videos, even if the videos are under Creative Commons license. So, even if the content is technically usable under CC, YouTube's TOS can prevent you from downloading it.

[1:10:03] Mid-training + Post-training

This section focuses on particular capabilities, rather than general high quality. The boundary between pre-training and these stages is often blurry.

- **Long context**: Models now have very long context windows (e.g., DeepSeek v3 has 128K tokens, Claude 3.5 Sonnet has 200K tokens, Gemini 1.5 Pro has 1.5M tokens).
    - Transformers scale quadratically with sequence length, so training on long context is expensive.
    - It's not efficient to pre-train on long contexts from the start. Instead, long context is often added later during mid-training.
    - **LongLora (Chen+ 2023)**: Extends context length of LLaMA 2 7B from 4K to 100K tokens. Uses shifted sparse attention and positional interpolation.
    - **Trained on long documents**: PG-19 (books) and Proof-Pile (math) are used for long context training.
- **Tasks (based on standard datasets)**:
    - `TL;DR`: convert lots of existing NLP datasets into prompts.
    - **Super-Natural Instructions (Wang+ 2022)**: Dataset of 1.6K+ tasks. Fine-tune T5 on k-shot learning. Tasks contributed by community (via GitHub). Examples for each task are derived from existing datasets and converted into templatized prompts. Outperforms `InstructGPT` despite being much smaller.
    - **Flan 2022 (Longpre+ 2023)**: Dataset of 1.8K+ tasks. Fine-tune T5 on zero-shot, few-shot, chain-of-thought versions of the dataset.
- **Instruction_chat (instruction following and chat)**:
    - `TL;DR`: more open-ended instructions, heavy use of synthetic data.
    - **Alpaca (Taori+ 2023)**: Dataset of 52K examples from `text-davinci-003` using `self-instruct` (Wang+ 2022). Fine-tune LLaMA 7B on this dataset.
    - **Self-Instruct**: Prompting an LLM (e.g., `text-davinci-003`) to generate instructions and examples, then filtering for quality. This is a common way to create synthetic instruction data.
    - **Vicuna**: Fine-tuned LLaMA on 70K conversations from ShareGPT (users sharing their ChatGPT conversations; deprecated now).
    - **Baize (Xu+ 2023)**: Generate dataset (111.5K examples) from GPT-3.5 using self-chat (seeded with Quora and StackOverflow questions).

- **WizardLM (Xu+ 2023)**: `Evol-Instruct` dataset ('evolve' questions to increase breadth/difficulty). Fine-tuned LLaMA on this dataset.
- **MAMmoTH2 (Yue+ 2024)**: Curated WebInstruct, 10M instructions from Common Crawl. Filter: train `fastText` classifier on quiz sites. Extract: use GPT-4 and Mistral to extract QA pairs. Fine-tune Mistral 7B on this data. Boosts math performance.
- **OpenHermes 2.5**: Agglomeration of many datasets. Fine-tune Mistral 7B on 1M examples from GPT-4 (model).
- **Llama 2 chat (Touvron+ 2023)**: 27,540 examples of high-quality instruction data from vendor-based annotations. Said was better than using the millions of examples from open datasets. Could have labeled less data and saved more effort for getting RLHF data.
- **Llama-Nemotron (NVIDIA, 2024)**: Prompts: public datasets (e.g., WildChat) or synthetically-generated, then filtered. Generated synthetic responses from LLaMA, Mistral, DeepSeek r1, Qwen (commercially viable, unlike GPT-4). Included reasoning traces.

[1:17:26] Summary

- **Key lesson**: Data does not fall from the sky. You have to work to get it.
- **Live service => raw data => processed data**: This pipeline involves conversion, filtering, and deduplication.
- **Data is the key ingredient that differentiates language models**: Architectures are converging, so data quality and composition are crucial.
- **Legal and ethical issues**: Copyright, privacy, etc., are significant challenges.
- **This pipeline is heuristic**: It's not a science, but an art. This means there are many opportunities to improve!

Practical Takeaways

- The quality and composition of your training data are paramount for language model performance.
- Raw web data is often too noisy and unstructured; significant preprocessing is required.
- Model-based filtering, while potentially introducing biases, can be highly effective for curating high-quality datasets.

- Synthetic data generation is becoming increasingly important for creating specialized instruction-following datasets.
- Be aware of copyright and terms of service when sourcing data; not all publicly available data can be freely used for commercial model training.
- Long context and instruction following are key capabilities that require specific data strategies during mid- and post-training.

Open Questions / Things to Remember

- How to systematically evaluate the "quality" of data beyond simple heuristics or proxies like Wikipedia.
- The trade-off between data quantity and quality: how much low-quality data is equivalent to how much high-quality data?
- The ethical implications of using copyrighted or potentially biased data for training.
- How to ensure diversity in synthetically generated data to avoid overfitting to the generating model's biases.
- The long-term impact of data scarcity and proprietary datasets on open-source LLM development.
- The evolving legal landscape of AI and copyright will continue to shape data practices.

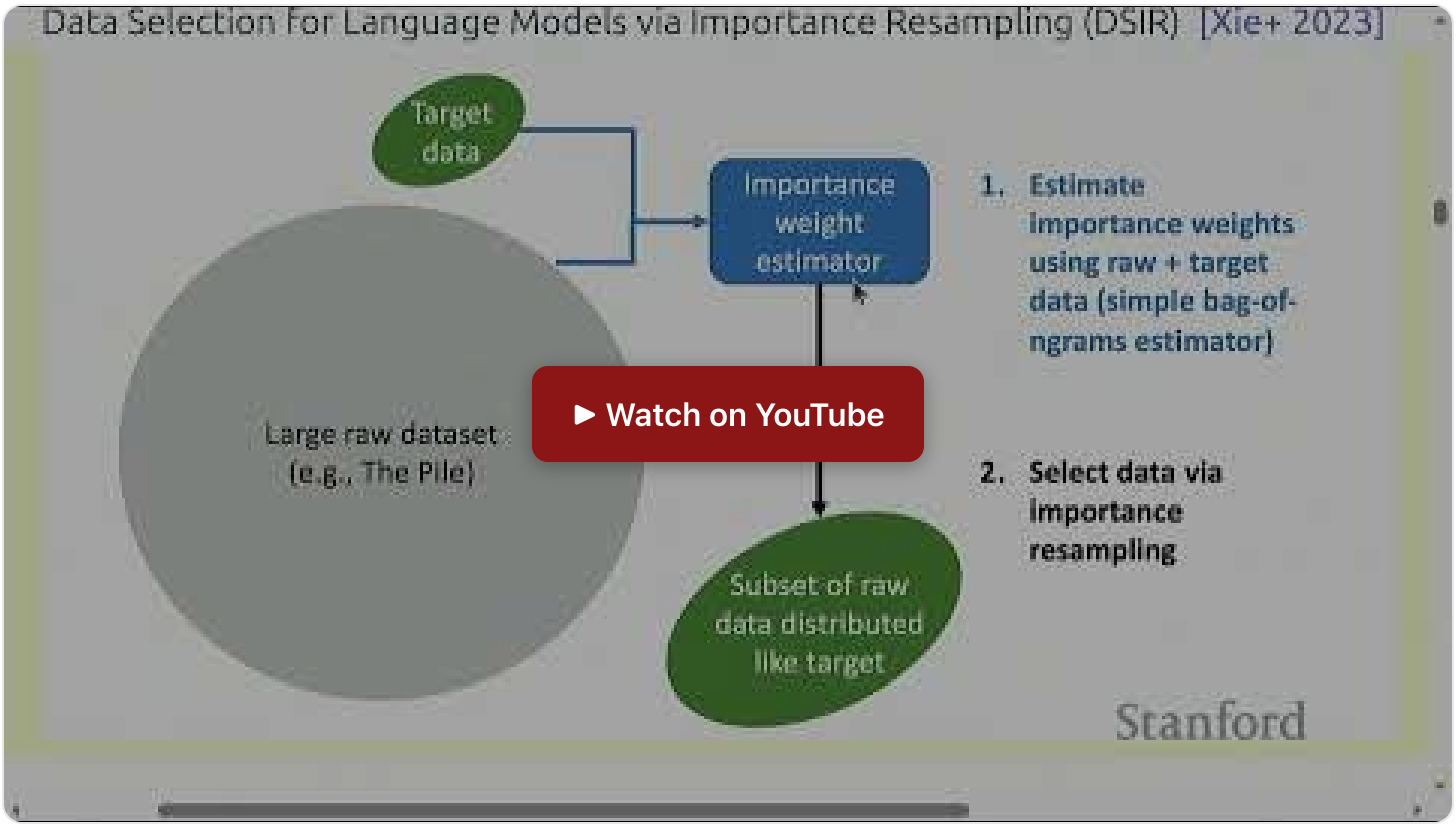
← Lec 12

Lec 14 →

Lecture 14: Data 2

Stanford CS336 · Spring 2025 · 1:19:12 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



TL;DR: * Data preparation for large language models involves filtering and deduplication. * Filtering algorithms aim to select high-quality data similar to a target distribution from a vast raw corpus. * Common filtering techniques include n-gram language models (KenLM), fastText classifiers, and importance resampling (DSIR). * Deduplication removes redundant or near-redundant data to improve training efficiency and mitigate memorization. * Hashing is a fundamental tool for both exact deduplication (e.g., Bloom filters) and approximate deduplication (e.g., MinHash, LSH).

Key Concepts: * **Data Filtering Pipeline:** Live service -> dump/crawl -> processed data (HTML to text, filtering, deduplication). * **Filtering Goal:** Given target data T (small, high-quality) and raw data R (large), find subset T' of R similar to T. * **Filtering Desiderata:** Generalize from T (want

T' to be different from T), extremely fast (to run on R). * **N-gram Language Models (KenLM):** * Maximum Likelihood Estimation (MLE) of $P(w_n|w_{n-1}\dots w_1)$. * Sparse counts problem. * Kneser-Ney smoothing for unseen n-grams. * Perplexity as a scoring function. * Application: CCNET (filter paragraphs by perplexity). * **FastText Classifiers:** * Bag-of-words embeddings with hashing trick for n-grams. * Linear classifier (no non-linearity). * Parallelized, asynchronous SGD. * Application: Language identification, quality filtering. * **Data Selection for Language Models via Importance Resampling (DSIR):** * Estimate importance weights using raw + target data (e.g., simple bag-of-n-grams estimator). * Select data via importance resampling. * Problem: Target data D_P is too small to estimate a good model. * Solution: Use hashed n-grams. * **General Filtering Framework:** 1. Estimate some model based on T and R and derive a scoring function. 2. Keep examples in R based on their score (stochastically). * **Deduplication:** * **Exact Duplicates:** Identical copies (e.g., mirror sites, GitHub forks, Project Gutenberg mirrors). * **Near Duplicates:** Text differing by a few tokens (e.g., terms of service, licenses, formulaic writing, minor formatting). * **Benefits:** Train more efficiently (fewer tokens), avoid memorization (mitigate copyright/privacy concerns). * **Design Space:** What is an item (sentence, paragraph, document)? How to match (exact, fraction of common subitems)? What action to take (remove all, remove all but one)? * **Key Challenge:** Dduplication is fundamentally about comparing items to other items (pair-wise). Need linear time algorithms to scale. * **Hashing:** * Maps item to a hash value (integer or string). * Hash value much smaller than item. * Hash collision: $h(x) = h(y)$ for $x \neq y$. * Trade-off between efficiency and collision resistance. * MurmurHash (fast, used for hash tables). * **Bloom Filter:** * Efficient, approximate data structure for testing set membership. * Features: Memory efficient, can update, but can't delete. * If return 'no', definitely 'no'. If return 'yes', most likely 'yes' but small probability of 'no' (false positive). * False positive rate can be driven down exponentially with more time/compute. * **MinHash:** * Random hash function h such that $P(h(A) = h(B)) = \text{Jaccard}(A, B)$. * Normally, you want different items to hash to different hashes, but here, you want collision probability to depend on similarity. * **Locality Sensitive Hashing (LSH):** * Breaks n hash functions into B bands of R hashes ($n = B \times R$). * A and B collide if for *some* band, *all* its hashes return the same value. * Sharpens the probability curve around a threshold.

[0:00] **Lecture 14: Data 2 - Deep Dive into Data Processing**

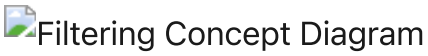
This lecture builds upon the previous one, which provided an overview of datasets used for training language models. We discussed the journey of data from live services (like GitHub) to processed datasets (like The Stack), involving steps like HTML to text conversion, quality/toxicity filtering, and deduplication.

Today, we will take a deep dive into the mechanics of these processing steps, focusing on: * Algorithms for filtering (e.g., classifiers). * Applications of filtering (e.g., language, quality, toxicity). * Deduplication (e.g., Bloom filters, MinHash, LSH).

This lecture will involve more classical big data processing algorithms and some interesting math.

[0:05] **1. Filtering Algorithms**

The fundamental algorithmic building block for filtering is: **Given some target data T (which is typically small and high-quality) and lots of raw data R (which is huge, e.g., Common Crawl), find a subset T' of R that is similar to T.**



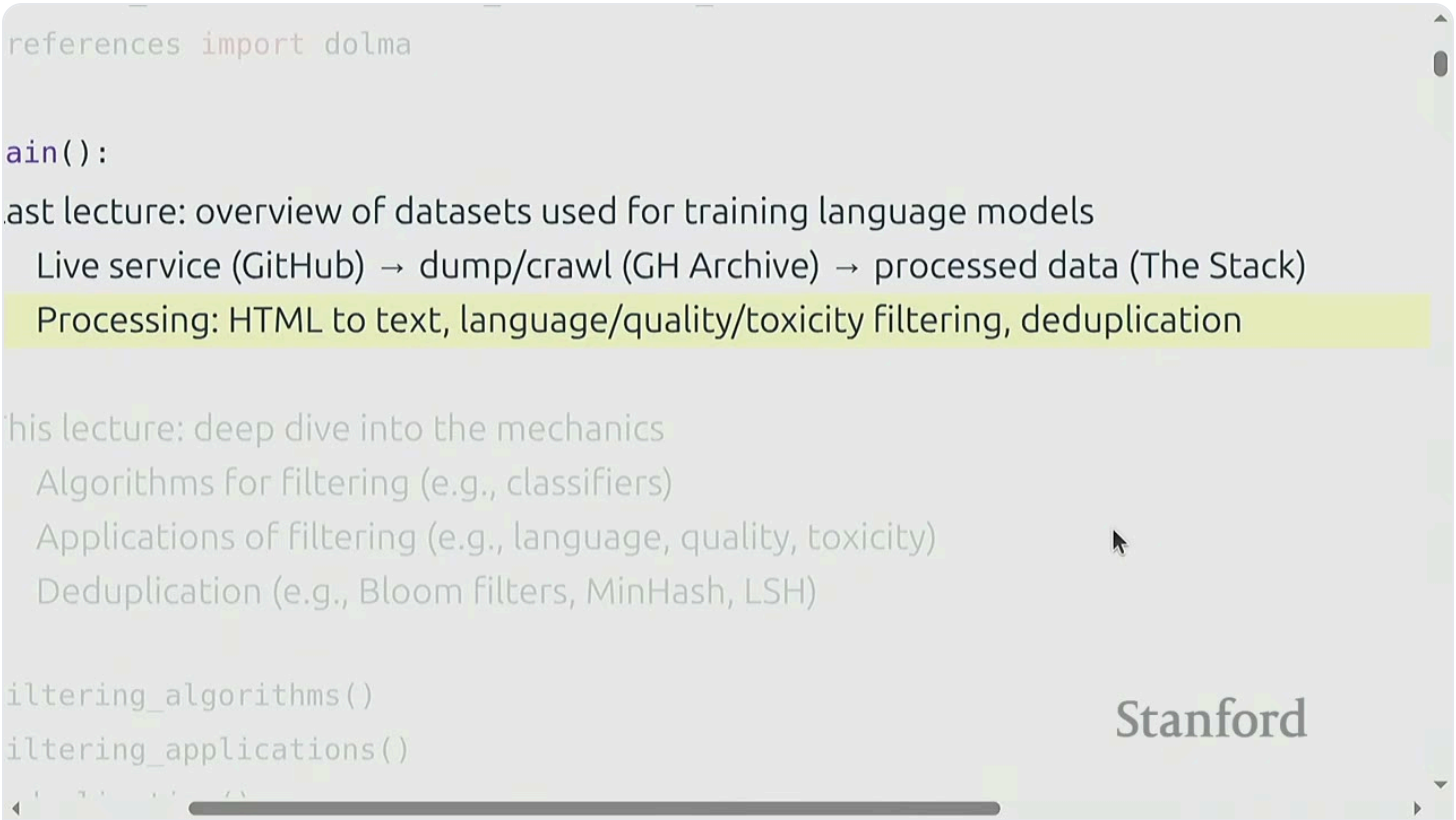
Desiderata for a filtering algorithm: * **Generalize from the target data:** We want T' to be different from T. If T' is identical to T, there's no point in filtering. * **Extremely fast:** It must be able to run on R, which is massive (e.g., the entire web). If filtering is as expensive as training a large language model, it defeats the purpose.

We will explore three different ways to implement this filtering primitive: 1. Training an n-gram model (using KenLM). 2. Training a classifier (using FastText). 3. Using importance resampling (DSIR).

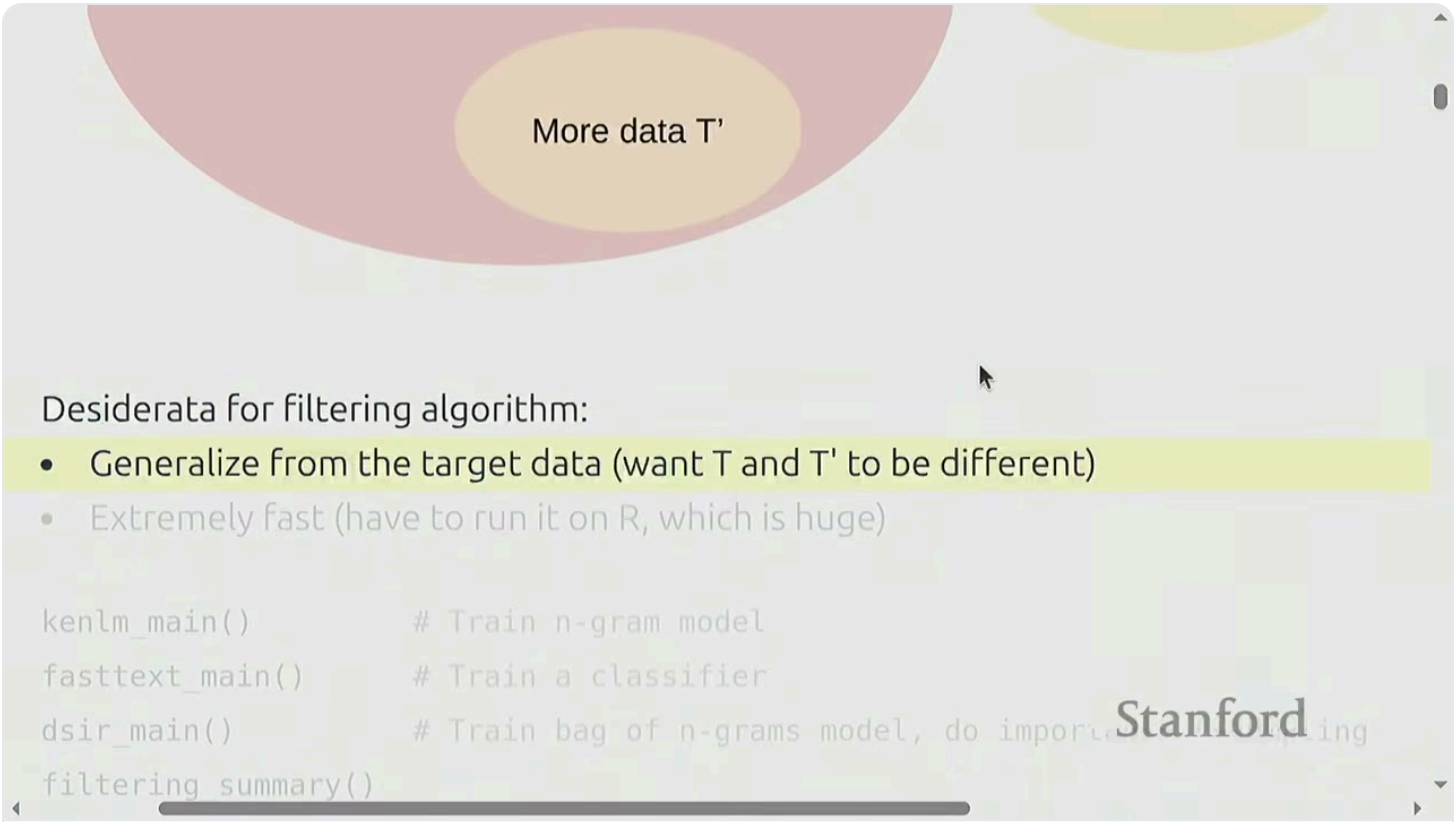
[2:45] 1.1. N-gram Models with Kneser-Ney Smoothing (KenLM)



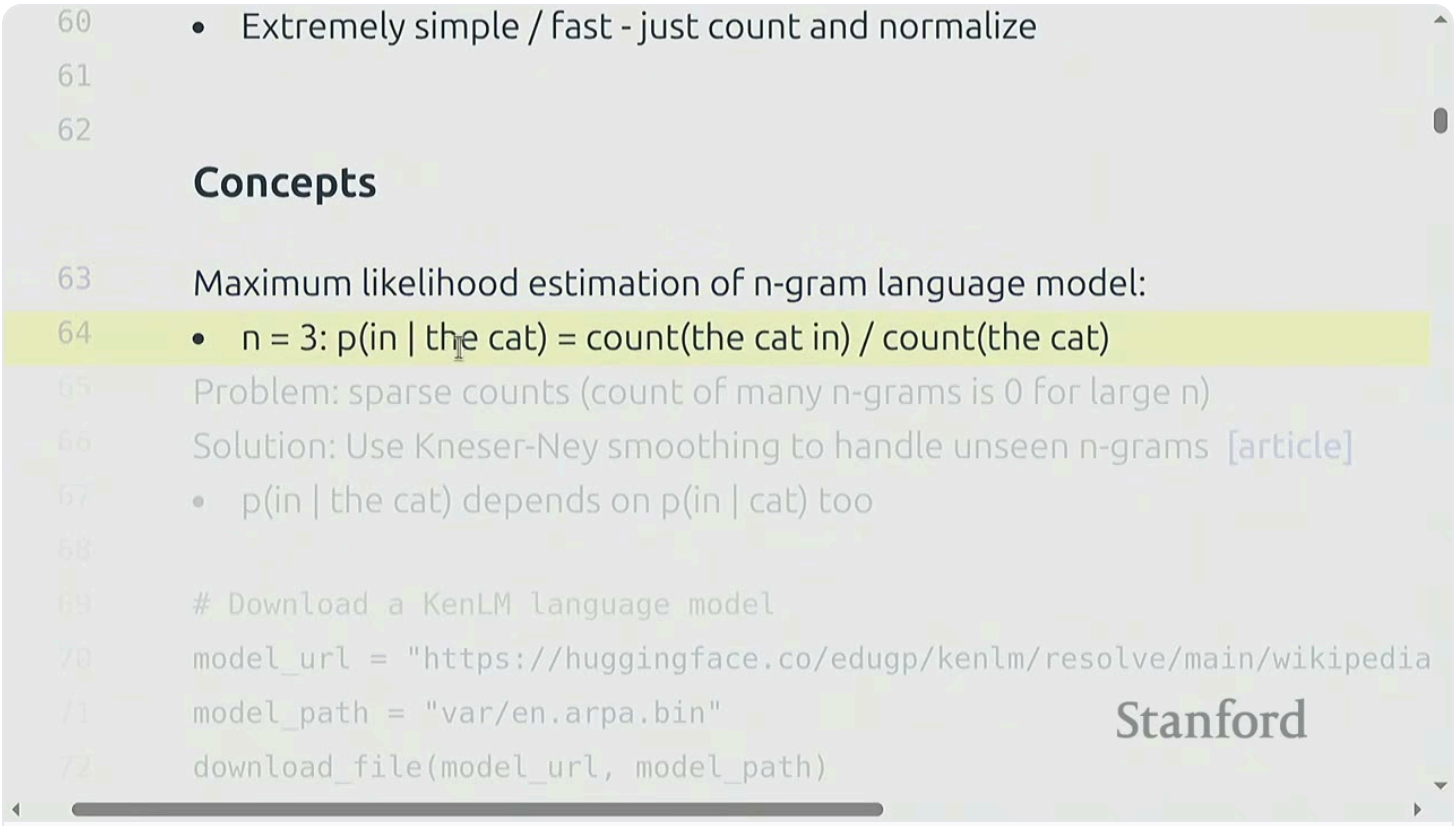
0:11 A speaker stands in front of a whiteboard, presenting a lecture with a laptop.



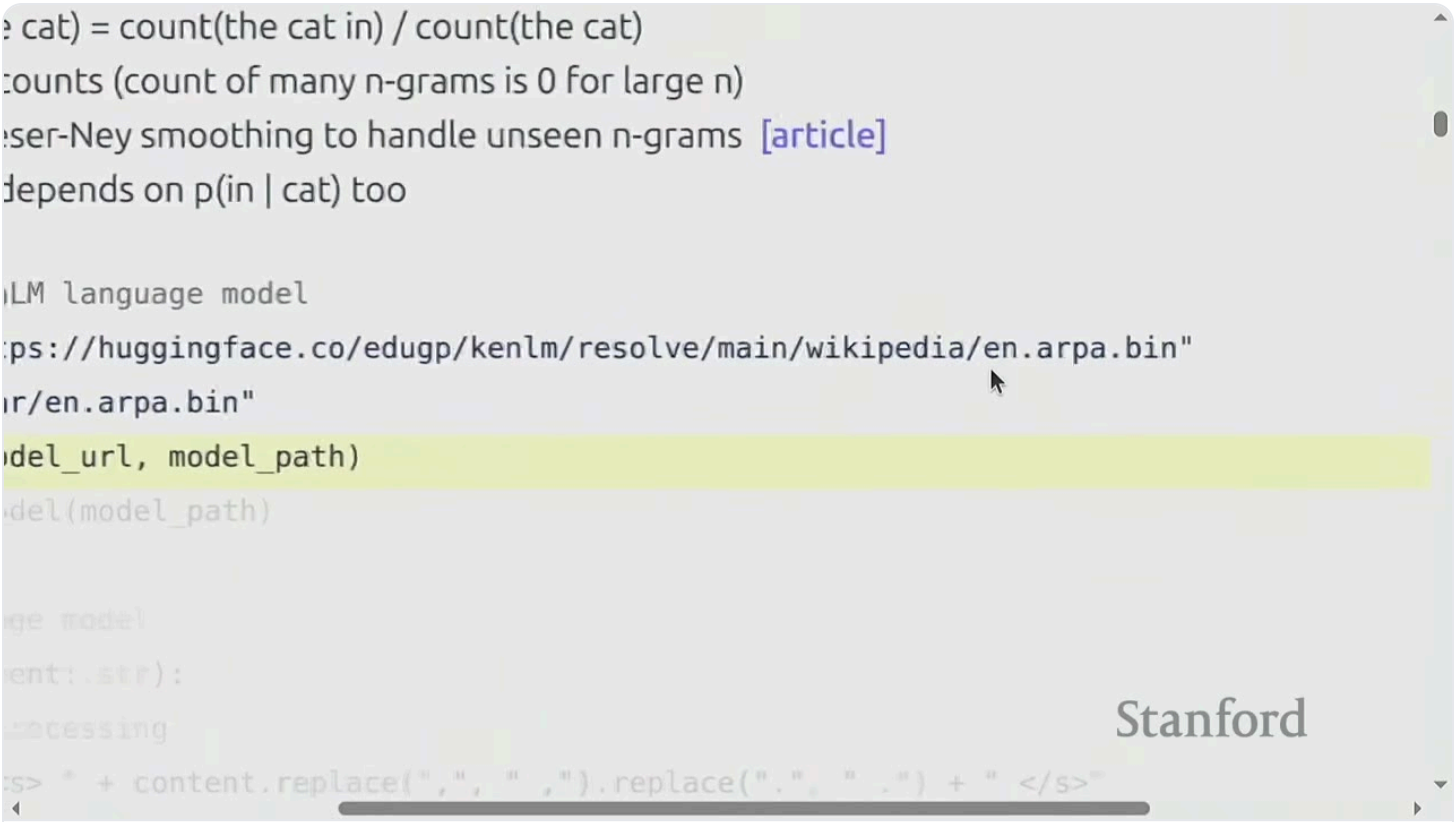
0:47 A slide highlights the processing steps for language model training data: HTML to text, filtering, and deduplication.



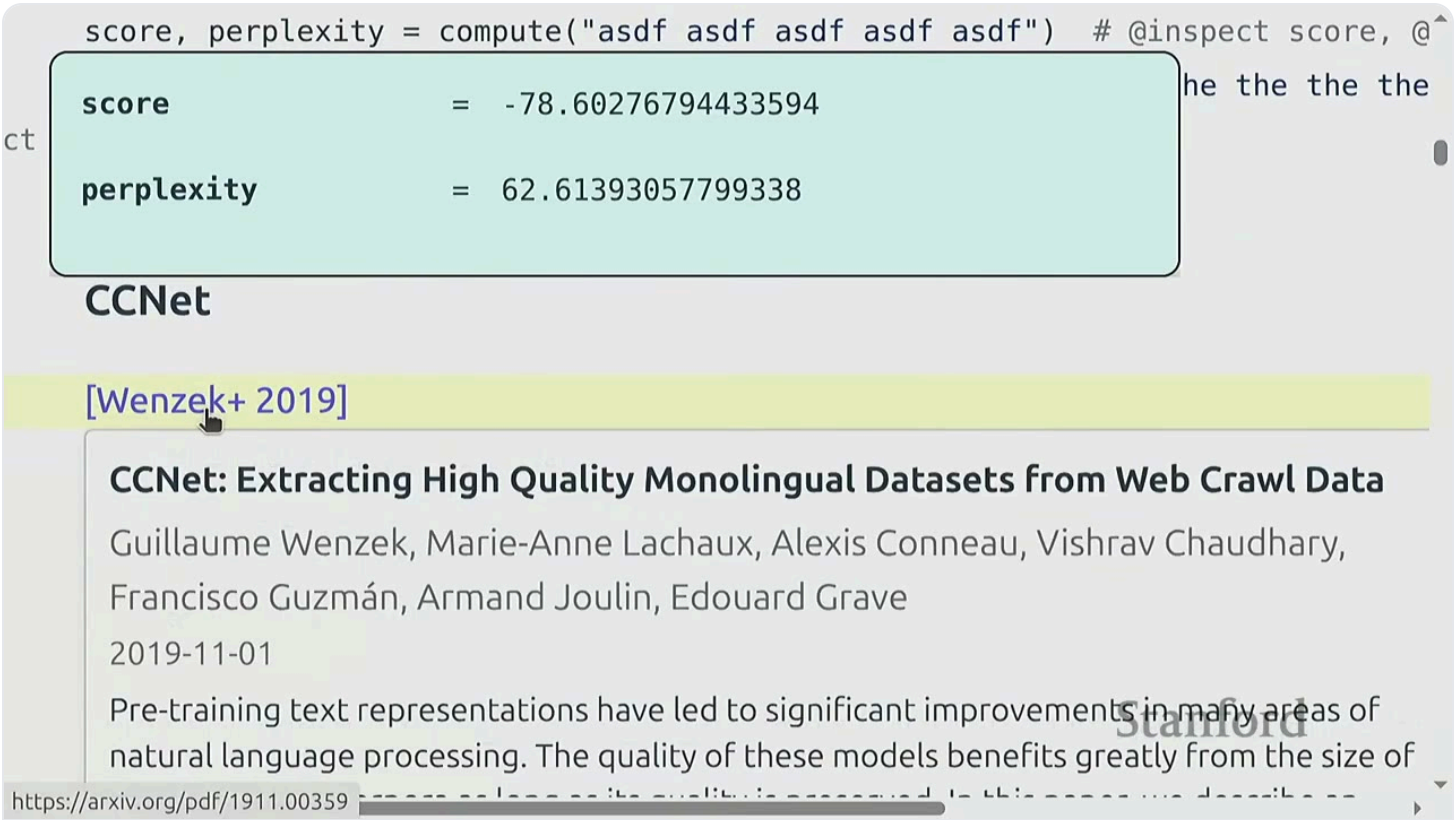
2:10 A slide shows desiderata for filtering algorithms, emphasizing generalization from target data and speed.



4:05 A slide explains maximum likelihood estimation for n-gram language models, with an example for n=3.



5:35 A slide displays Python code for downloading a KenLM language model from Hugging Face.



7:20 A slide shows computed score and perplexity values, and a reference to the CCNet paper.

N-gram models are a classic approach in natural language processing. * **Kneser-Ney smoothing:** A technique used to estimate probabilities of n-grams, especially for unseen n-grams. It's a common choice in statistical NLP. * **KenLM:** A fast, open-source implementation of n-gram language models, originally developed for machine translation. * **Commonly used for data filtering:** It's simple, fast, and effective. * **Extremely simple / fast:** Fitting an n-gram model primarily involves counting n-gram occurrences and then normalizing.

Concepts: The starting point is **Maximum Likelihood Estimation (MLE)** for n-gram language models. For an n-gram of size $n = 3$ (e.g., "the cat"), the probability $P(\text{in}|\text{the cat})$ is estimated as:

$$P(\text{in}|\text{the cat}) = \frac{\text{count}(\text{the cat in})}{\text{count}(\text{the cat})}$$

The problem with MLE is **sparse counts**: many n-grams (especially for large n) will have a count of 0, even if they are perfectly reasonable. This is why n-gram models historically struggled with the "curse of dimensionality" as n increased. **Solution:** Use Kneser-Ney smoothing to handle unseen n-grams. Roughly, if a higher-order n-gram (e.g., $P(\text{in}|\text{the cat})$) has insufficient counts, Kneser-Ney smoothing "backs off" to lower-order n-grams (e.g., $P(\text{in}|\text{cat})$) to estimate the probability.

Let's demonstrate with some code: First, download a pre-trained KenLM language model (trained on Wikipedia):

```
model_url = "https://huggingface.co/edugp/kenlm/resolve/main/wikipedia/en.arpa.bin"
model_path = "var/en.arpa.bin"
download_file(model_url, model_path)
```

Then, use the `kenlm.Model` to compute scores and perplexity. Perplexity is a measure of how well a probability distribution predicts a sample. Lower perplexity means the model predicts the text better (i.e., the text is more "likely" given the model). We normalize perplexity by the number of tokens to avoid favoring short documents.

```
import kenlm
import math

model = kenlm.Model(model_path)
```

```
def compute(content: str):
    # Hacky preprocessing
    content = "<s> " + content.replace(",", " ").replace("\"", " ").replace(".", " ") + " </s>"

    # log p(content)
    score = model.score(content)

    # Perplexity normalizes by number of tokens to avoid favoring short documents
    num_tokens = len(list(model.full_scores(content)))
    perplexity = math.exp(-score / num_tokens)

    return score, perplexity

# Example 1: Wikipedia excerpt
score, perplexity = compute("Stanford University was founded in 1885 by Leland and Jane Stanfor
# Output: score = -151.73, perplexity = 187.19
```

The perplexity for the Wikipedia excerpt is around 187. This is a reasonable value, as the text is likely to be found in the Wikipedia-trained model.

```
# Example 2: Text from a course website
score, perplexity = compute("If you believe that the course staff made an objective error in mer
# Output: score = -180.81, perplexity = 204.01
```

This text, from a course website, has a slightly higher perplexity (204), meaning it's less likely than the Wikipedia excerpt. This makes sense, as course-specific text is less likely to be in a general Wikipedia corpus. However, it's still well-formed English, so the perplexity isn't abnormally high.

```
# Example 3: Gibberish
score, perplexity = compute("asdf asdf asdf asdf asdf")
# Output: score = -44.88, perplexity = 272.08
```

Gibberish text yields a higher perplexity (272), indicating it's very unlikely according to the model.

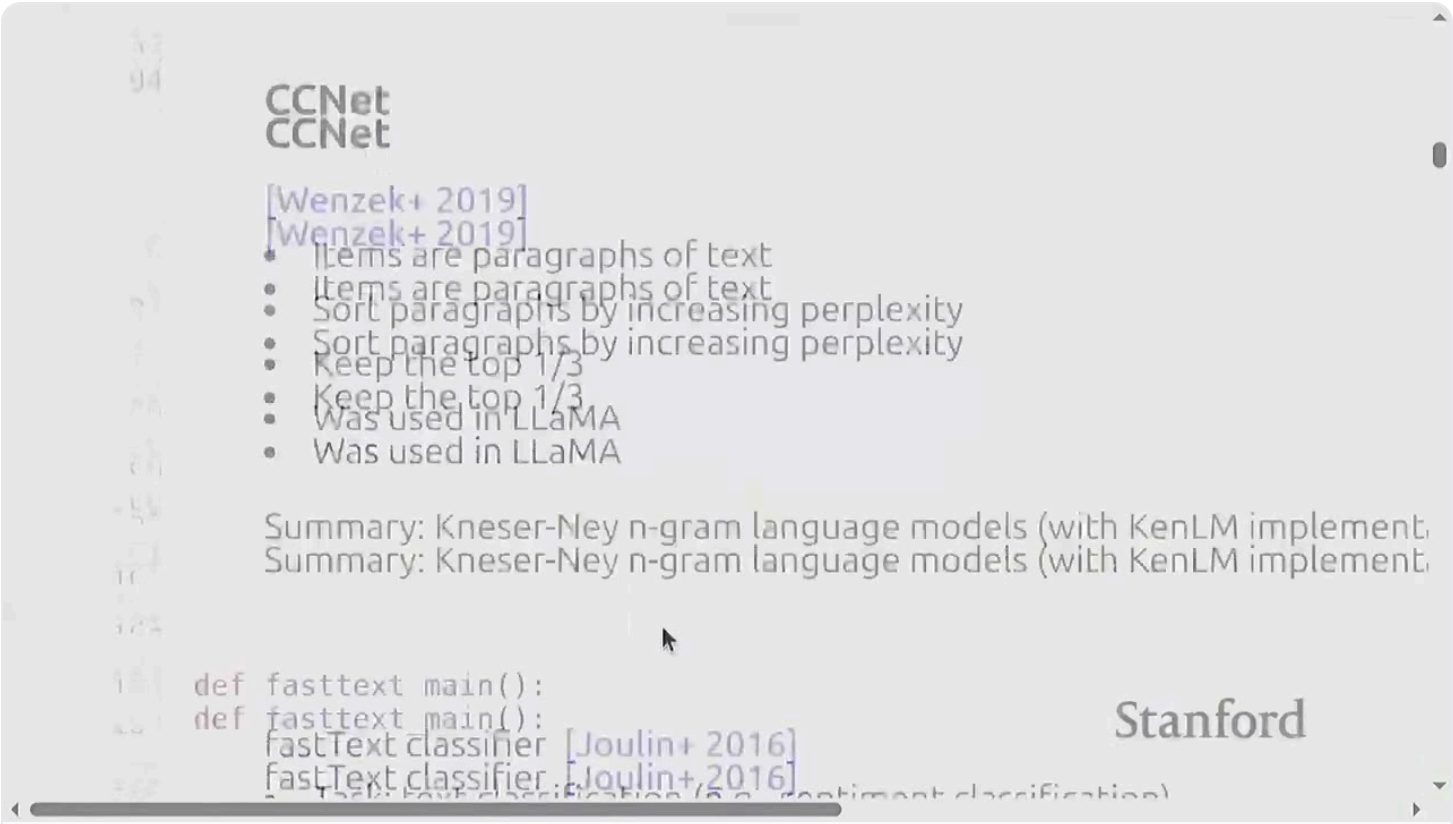
```
# Example 4: Repetitive text
score, perplexity = compute("the the the the the the the the the the the")
# Output: score = -78.60, perplexity = 62.61
```

Surprisingly, the perplexity for "the the the..." is quite low (62). This highlights a limitation of simple n-gram models: they capture local word sequences but don't necessarily understand broader semantic coherence.

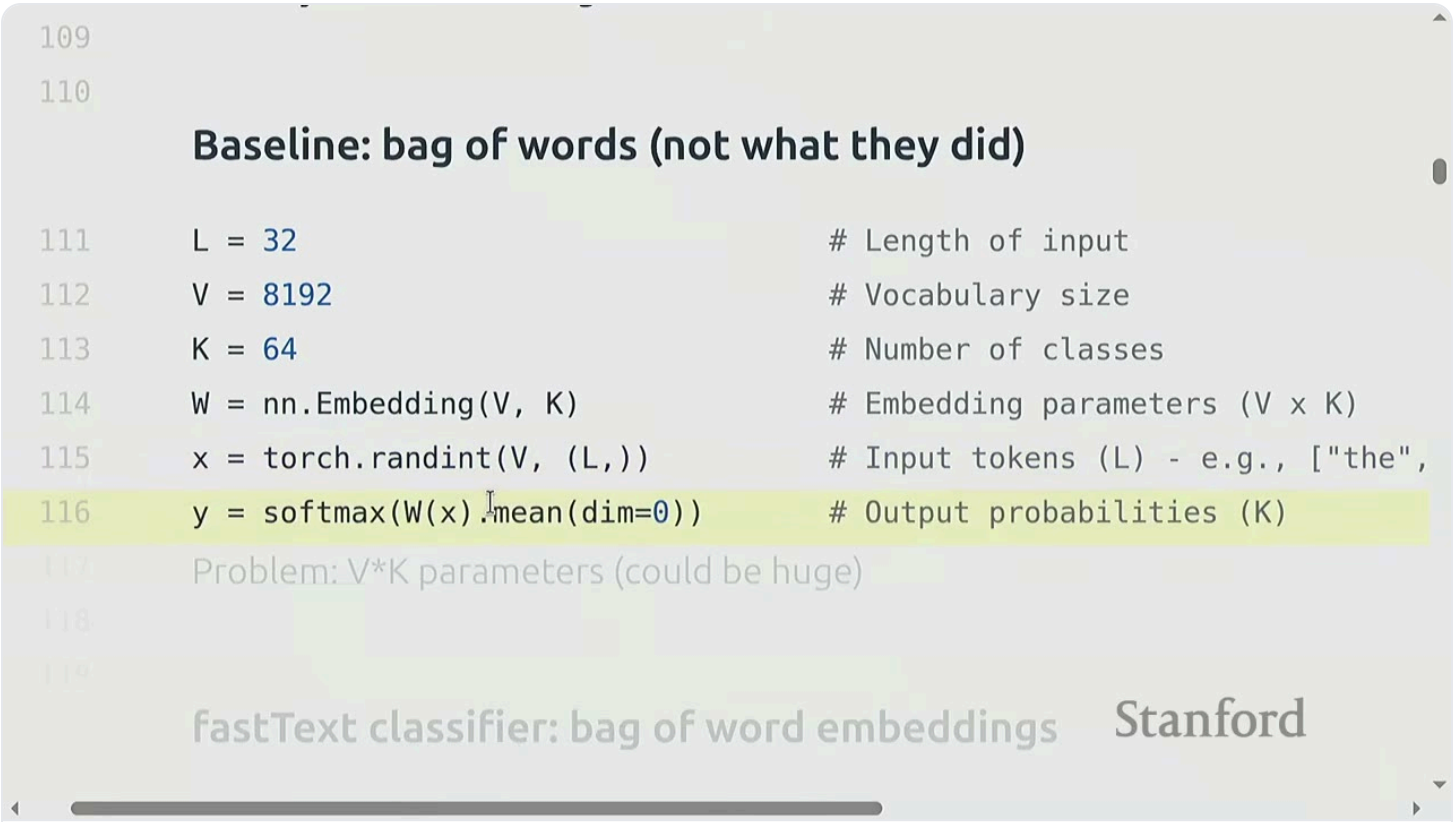
CCNet [Wenzek+ 2019] The CCNET paper, which we discussed previously, used a similar approach for data filtering. * **Items are paragraphs of text.** * **Sort paragraphs by increasing perplexity.** * **Keep the top 1/3.** * This method was used to create the first LLaMA data set.

Summary: Kneser-Ney n-gram language models (with KenLM implementation) are fast but crude.

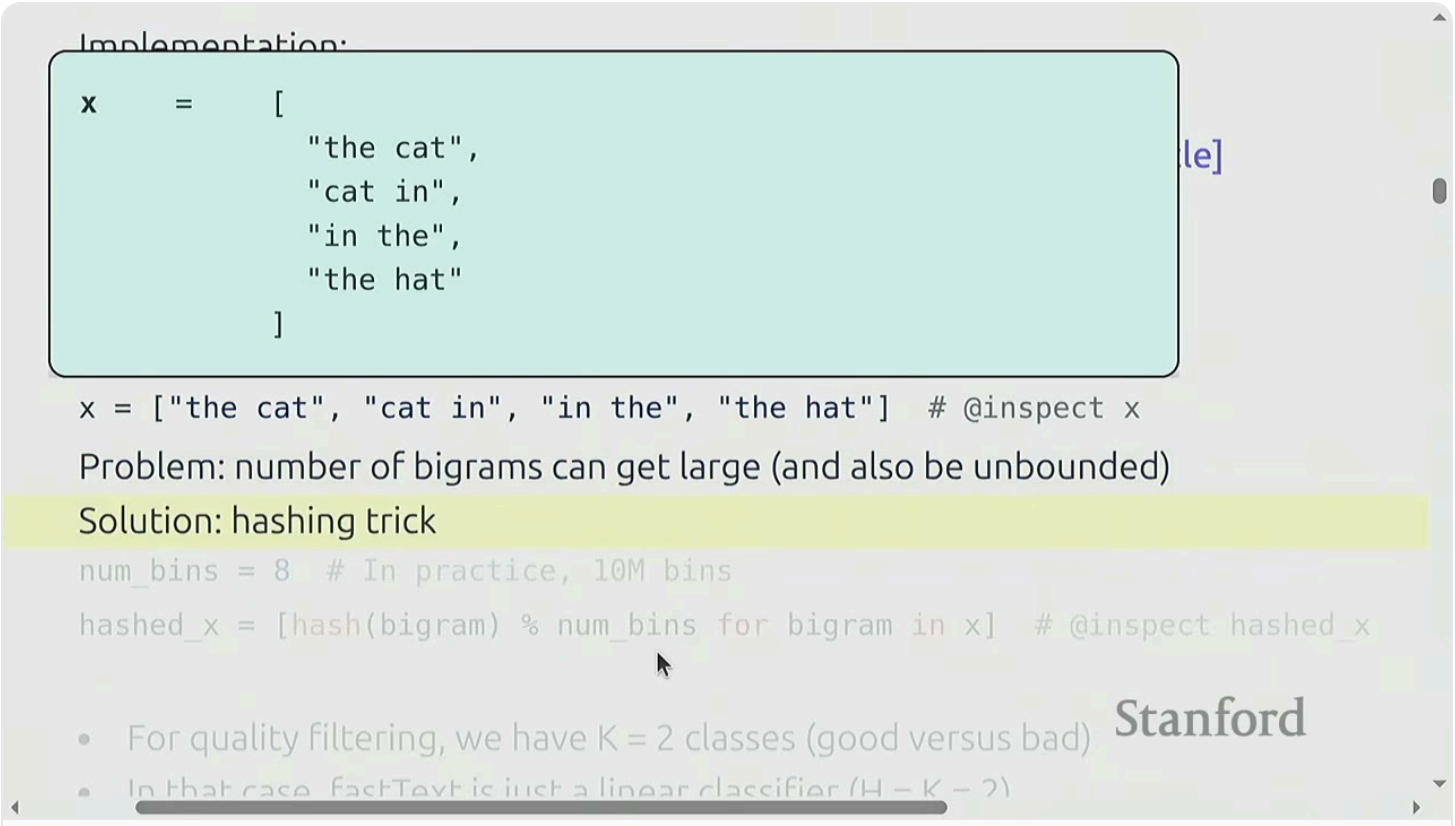
[8:39] 1.2. FastText Classifiers



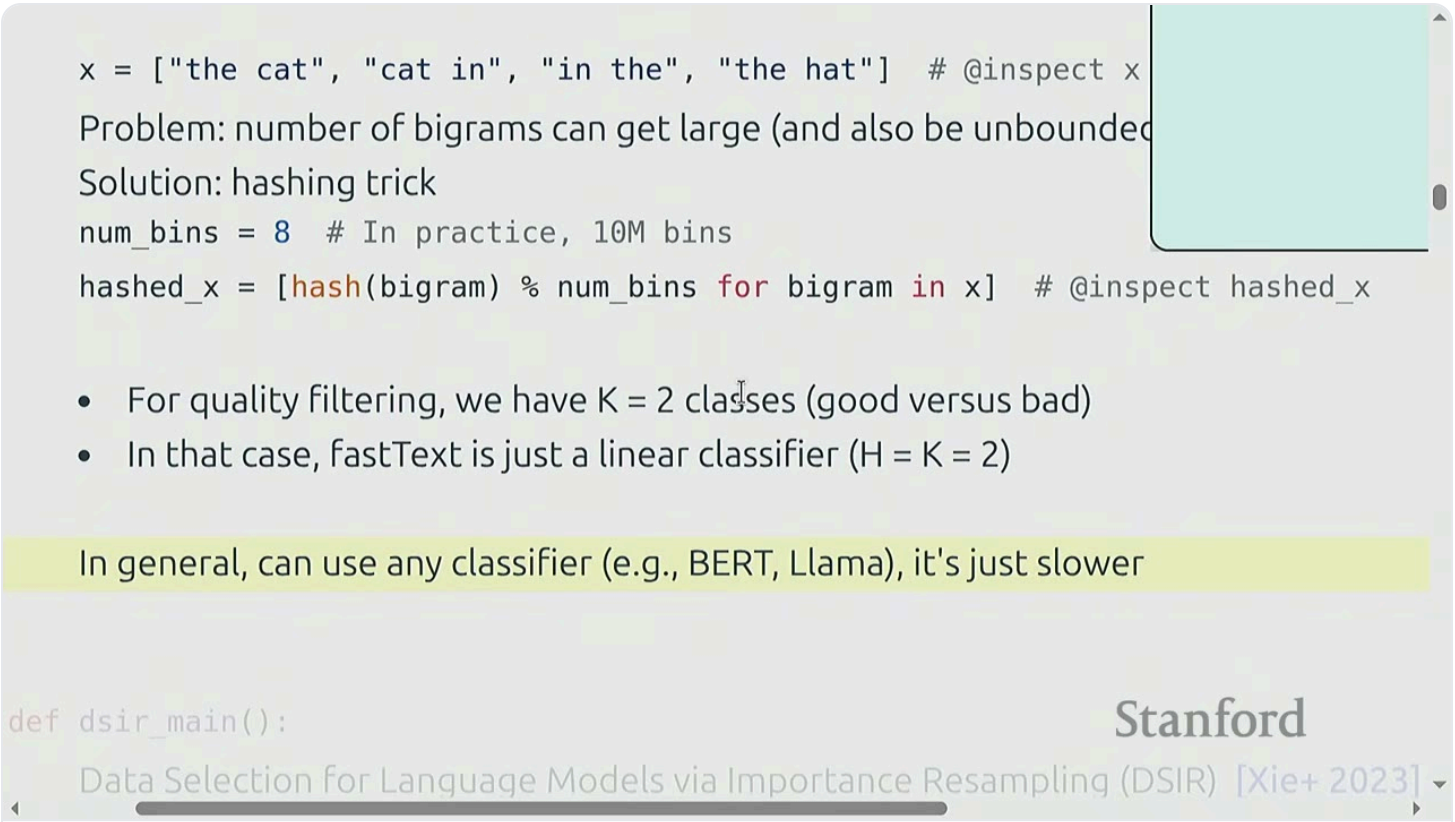
8:20 A slide lists characteristics of CCNet, including sorting paragraphs by perplexity and use in LLaMA.



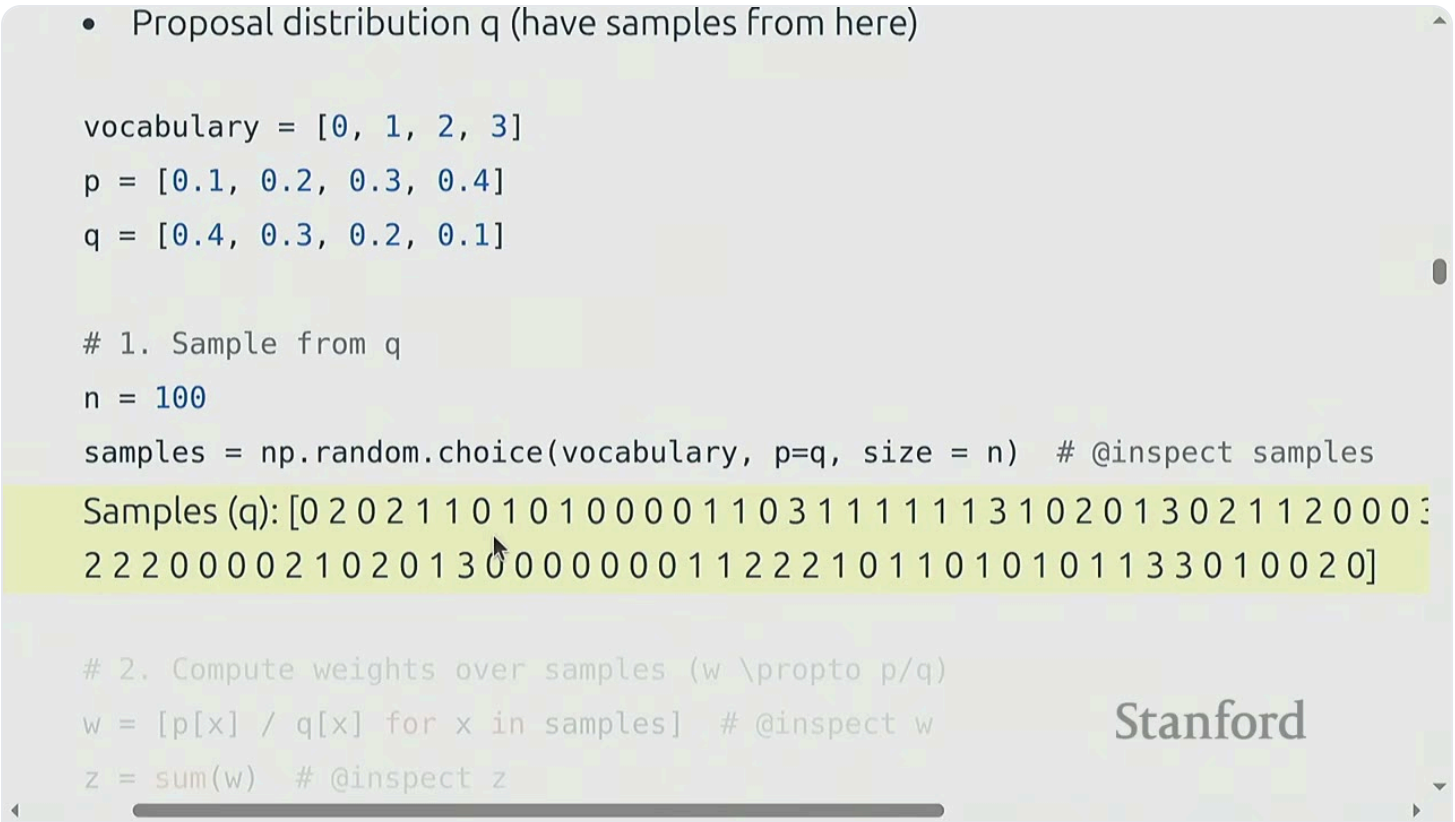
9:55 A slide presents Python code defining a baseline bag-of-words model with input length, vocabulary, and classes.



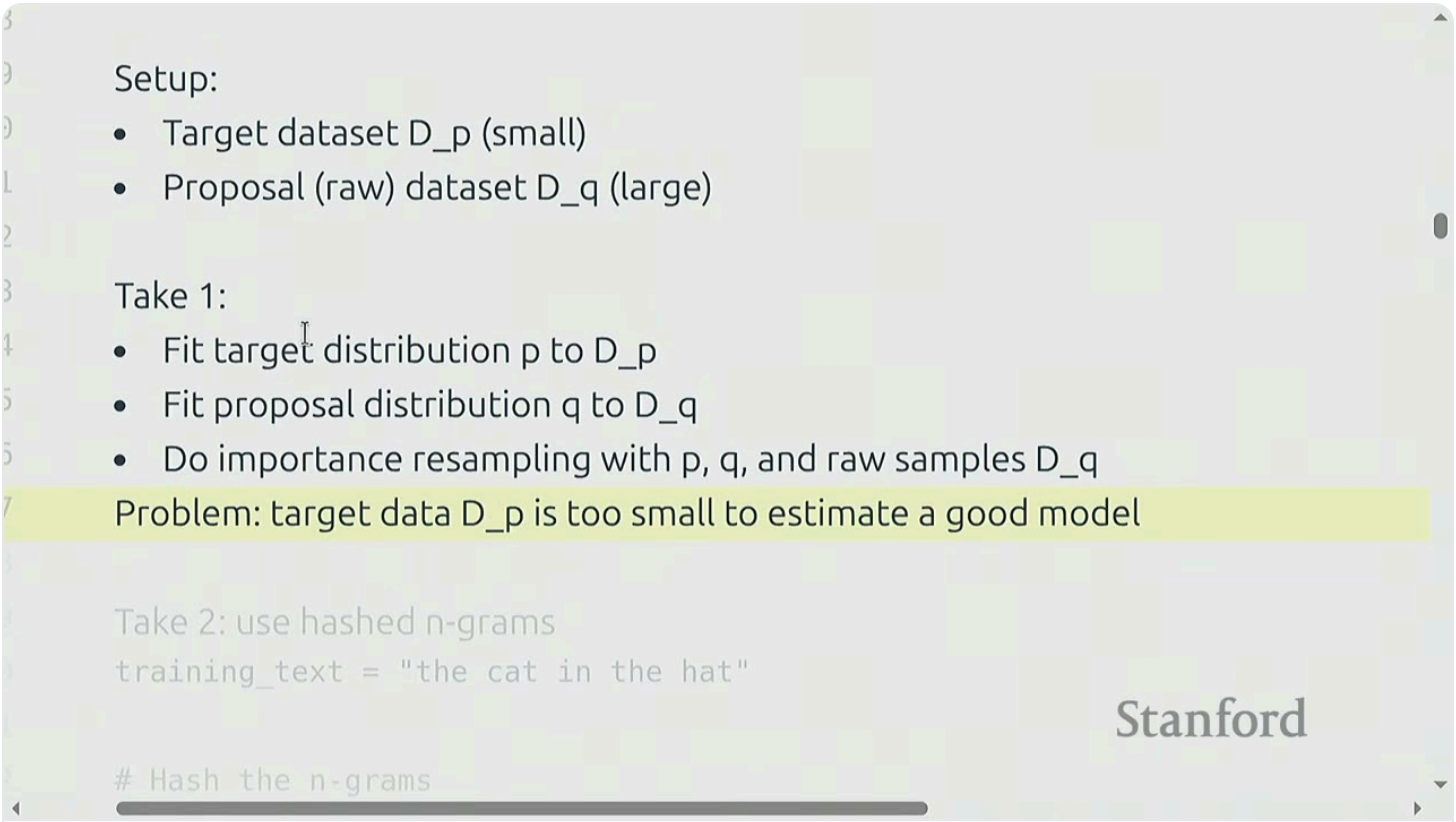
11:30 A slide illustrates the hashing trick solution for handling a large number of bigrams with example code.



13:10 A slide discusses using classifiers for quality filtering, noting that fastText is a linear classifier.



14:50 A slide shows Python code for sampling from a proposal distribution 'q' with a given vocabulary.



16:30 A slide outlines the setup for importance resampling, highlighting the problem of small target data.

FastText [Joulin+ 2016] is another popular approach, especially for its efficiency. * **Task:** Text classification (e.g., sentiment classification). * **Goal:** Train a fast classifier for text classification. * **Finding:** It was as good as much slower neural network classifiers.

Baseline: Bag of Words (not what they did) Consider a simple bag-of-words model for text classification. * $L = 32$ (Length of input) * $V = 8192$ (Vocabulary size) * $K = 64$ (Number of classes)

To classify, you might define an embedding layer: `W = nn.Embedding(V, K)` This represents embedding parameters as a $V \times K$ matrix. Input tokens x (e.g., "the") are then processed. `y = softmax(mean(W(x), dim=0))` The problem is that $V \times K$ parameters can be huge, leading to sparsity issues and high memory requirements.

FastText Classifier: Bag of Word Embeddings FastText addresses this by introducing a hidden dimension H. * $H = 16$ (Hidden dimension, much smaller than K) * `U = nn.Embedding(V, H)` (Embedding parameters $V \times H$) * `W = nn.Linear(H, K)` (Head parameters $H \times K$) * `y = softmax(mean(W(U(x)), dim=0))` (Output probabilities K) Notice that there's no non-linearity between the embedding and the linear layer; it's essentially a linear classifier (or matrix factorization). The number of parameters is greatly reduced: only $H \times (V + K)$ parameters.

Implementation: * Parallelized, asynchronous SGD. * Learning rate: linear interpolation from some number to 0.

Bag of N-grams: FastText extends this concept to n-grams. * Problem: The number of n-grams (e.g., bigrams) can get very large and is unbounded. * **Solution: Hashing trick.** * Define a fixed number of bins (e.g., 10 million bins in practice, 8 in this example). * Hash each n-gram into one of these bins: `hashed_x = [hash(bigram) % num_bins for bigram in x]` * This allows handling a potentially infinite vocabulary of n-grams within a fixed memory footprint. * Of course, collisions can occur, but the model learns to account for them during optimization.

Application to Filtering: * For quality filtering, we often have $K = 2$ classes (good vs. bad). * In that case, FastText is just a linear classifier ($H = K = 2$). * **In general, you can use any classifier** (e.g., BERT, LLaMA); it's just slower. The trade-off is that if you use a very powerful model for filtering, you might be better off using that compute for training your actual language model. Filtering a huge raw dataset means even a small increase in inference cost per item can lead to massive overall compute.

[1:04:55] 1.3. Data Selection for Language Models via Importance Resampling (DSIR)

DSIR [Xie+ 2023] offers a more principled approach to data selection. * **Basic Idea:** 1. Estimate importance weights using raw + target data (e.g., simple bag-of-n-grams estimator). 2. Select data via importance resampling.

Let's quickly review importance resampling: **Setup:** * **Target distribution p:** We want to sample from here. * **Proposal distribution q:** We have samples from here. * **Example:** * `vocabulary = [0, 1, 2, 3]` * `p = [0.1, 0.2, 0.3, 0.4]` (Target distribution) * `q = [0.4, 0.3, 0.2, 0.1]` (Proposal distribution)

Steps: 1. **Sample from q:** Draw N samples (e.g., $N = 100$) from the proposal distribution q. ```python samples = np.random.choice(vocabulary, p=q, size=n) # samples = [0, 2, 0, 2, 1, 0, 1, 0, 0, 0, 1, 1, 1, 3, 1, 0, 2, 0, 0, 0, 2, 2, 2, 0, 0, 0, 2, 1, 0, 1, 3, 0, 0, 0, 0, 0, 1, 1, 2, 2, 1, 0, 1, 0, 1, 0, 1, 1, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

← Lec 13

Lec 15 →

Lecture 15: Alignment — SFT/RLHF

Stanford CS336 · Spring 2025 · 1:14:51 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



TL;DR * Post-training (alignment) is crucial to make large language models (LLMs) useful and safe, transitioning from raw pre-trained models to instruction-following agents like ChatGPT. * Supervised Fine-Tuning (SFT) involves training on expert demonstrations, but the quality and style of this data significantly impact model behavior and can lead to issues like hallucination if not carefully managed. * Reinforcement Learning from Human Feedback (RLHF) optimizes models to maximize a measurable reward function, moving beyond simply imitating a reference distribution. * RLHF data collection (pairwise comparisons) is often cheaper than SFT, but still faces challenges with annotator quality, consistency, and ethical considerations (e.g., fair wages). * Algorithms like Proximal Policy Optimization (PPO) and Direct Preference Optimization (DPO) are used to adapt LLMs using human feedback, with DPO offering a simpler, more accessible approach by reframing RL as a maximum likelihood problem.

Key Concepts * Post-training / Alignment * Supervised Fine-Tuning (SFT) * Instruction Tuning Data (FLAN, Alpaca, Oasst) * Hallucination * Safety Tuning / Content Moderation * Mid-training / Two-phase Training * Reinforcement Learning from Human Feedback (RLHF) * Reward Model * Pairwise Feedback * Policy Gradient Theorem * Proximal Policy Optimization (PPO) * Trust Region Policy Optimization (TRPO) * Direct Preference Optimization (DPO) * KL Divergence * Bradley-Terry Model * Self-preference / Self-bias * Length Effects * Generator-Validator Gap * Crowdsourcing complexities and ethics

---

[0:00] Introduction: From Pre-training to Post-training



0:14 Lecture 15 title slide: RLHF / Alignment for CS336.

The lecture shifts focus from pre-training (building large models and data components) to **post-training** or **alignment**, which aims to make these pre-trained models useful and safe.

Motivation: * **GPT-3 vs. ChatGPT:** GPT-3 was a remarkable system, but not particularly "useful" in a product sense (e.g., it didn't follow instructions well). ChatGPT, however, transformed the landscape by effectively following instructions and performing amazing feats. * **The Goal:** Understand the transition from a powerful pre-trained model (like GPT-3) to a highly aligned and useful system (like ChatGPT).

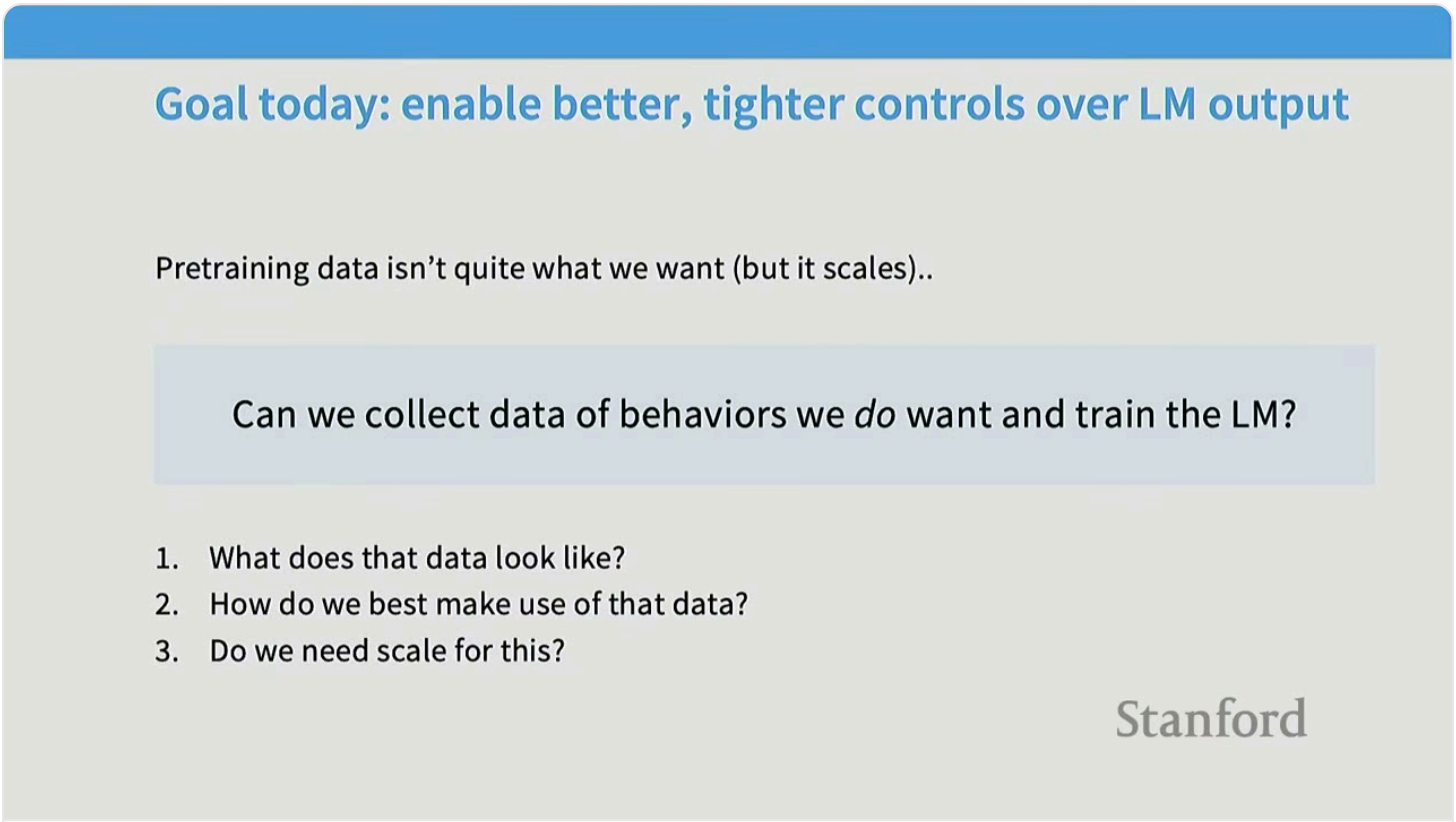
Key Aspects of Post-Training: 1. **Instruction Following:** Enabling models to understand and execute complex, nested instructions. * Example: GPT-4 generating `matplotlib` code from a long, detailed prompt (Sebastian Bubeck's "Sparks of AGI" paper). This ability to follow instructions is a key differentiator. 2. **Safety and Content Moderation:** Ensuring models are safe, prevent misuse (scams), and avoid generating toxic or harmful content. * ChatGPT's success is partly attributed to its robust guardrails.

Core Idea of Post-Training: * Pre-training "packs" the model with capabilities (reasoning, answering questions), but these aren't accessible "out of the box." * Post-training involves collecting specific data on desired behaviors and training the model to exhibit them. * **Key Questions:** * What does this "desired behavior" data look like? * How hard is it to collect? * How do we best make use of it (algorithmic questions)? * How do we scale this process?

[0:49] The RLHF Pipeline: Supervised Fine-Tuning (SFT)



0:49 Lecture 15 title slide: RLHF / Alignment for CS336, with lecturer visible.



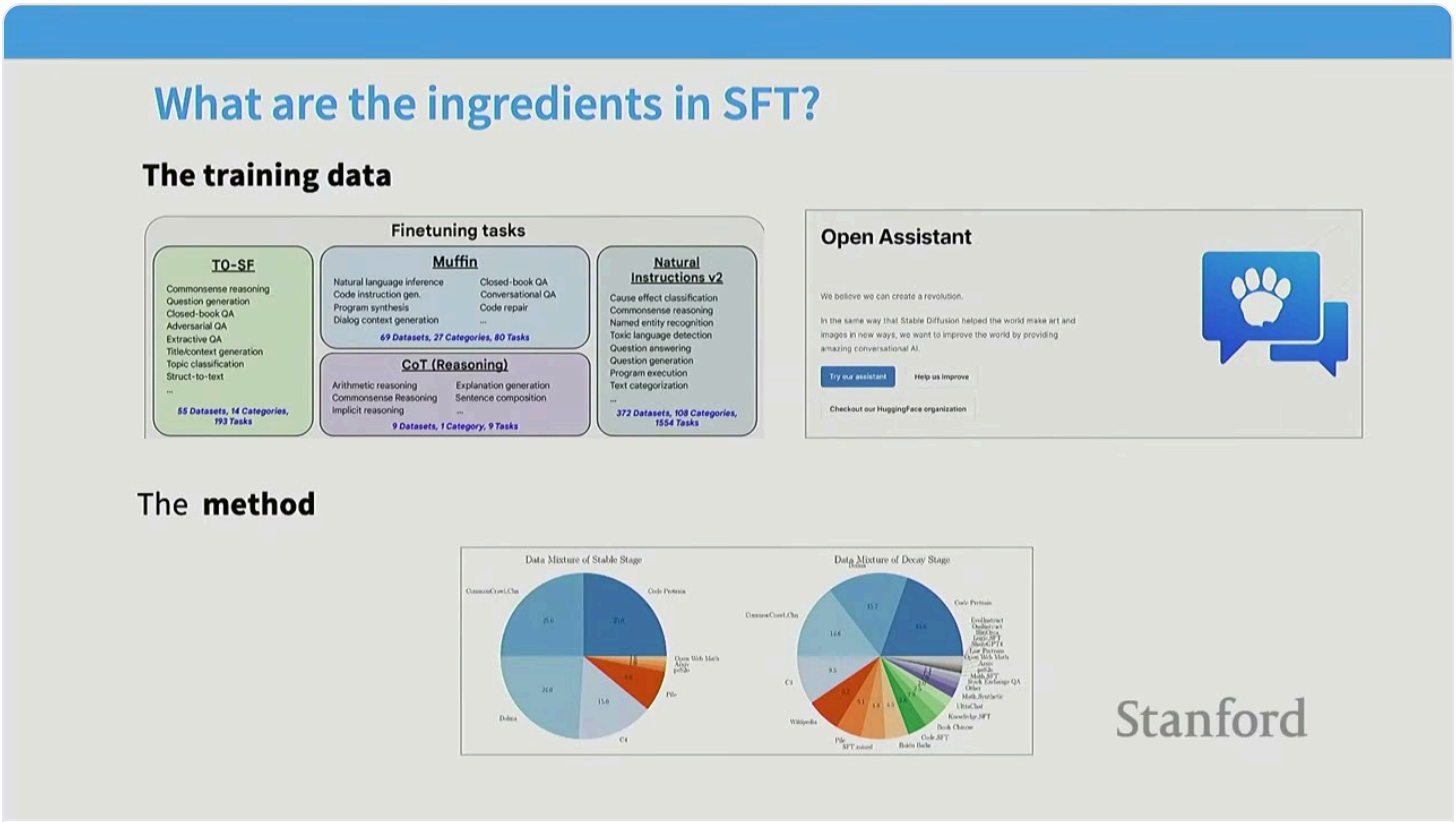
3:56 Slide on goal: enable better, tighter controls over LM output, with questions on data collection.

The lecture structure will roughly follow the InstructGPT paper, which outlines a three-step process for building instruction-following models:

1. **Supervised Fine-Tuning (SFT)**: Collect demonstration data and train a supervised policy.
2. **Reward Model Training**: Collect comparison data and train a reward model.
3. **Reinforcement Learning (RLHF)**: Optimize a policy against the reward model using reinforcement learning.

This lecture covers **Part 1 (SFT)** and **Part 2 (RLHF)**.

[5:11] Ingredients for SFT: Data and Method



5:15 Slide on ingredients in SFT: training data examples like Muffin, CoT, Natural Instructions, and Open Assistant.

Two main considerations for SFT: 1. **Training Data**: What does expert demonstration data look like? 2. **Method**: How do we adapt the model to this data? (Beyond simple gradient descent, there are non-obvious aspects).

[5:51] Training Data: Instruction Tuning Datasets

- **What's inside these datasets?**
- **What matters in building 'high-performance' instruction tuning data?**

The speaker introduces three types of instruction-tuning datasets, representing different paradigms:

1. **FLAN (Fine-tuned LAnguage Net)**:
    - Constructed by aggregating many existing NLP task datasets (e.g., T0-SF, Natural Instructions v2, CoT).
    - **Pros**: Easy to get lots of data for free by repurposing existing benchmarks.
    - **Cons**: Can be unnatural or "benchmark-centric." The format often requires "surgery" (e.g., appending options to a text) that doesn't resemble natural chat interactions.

2. **Alpaca (Stanford Alpaca)**:

    - An early attempt at using LLMs to generate instruction tuning data.
    - **Procedure**: A seed set of human-written instructions is used to prompt a powerful LLM (like GPT-3) to generate more instructions. Then, InstructGPT is used to generate responses for these instructions.
    - **Pros**: Generates data that feels more like natural chat interactions, with long-form natural language responses.
    - **Cons**: The generated instructions can be less diverse and shorter than human-written ones.

3. **Oasst (Open Assistant)**:

    - A crowd-sourced effort where online enthusiasts wrote instruction-tuning data.
    - **Pros**: High-quality, detailed human-written instructions and responses, often including citations.
    - **Cons**: Very difficult and expensive to collect at scale.

[2:01] Interactive Annotation Task

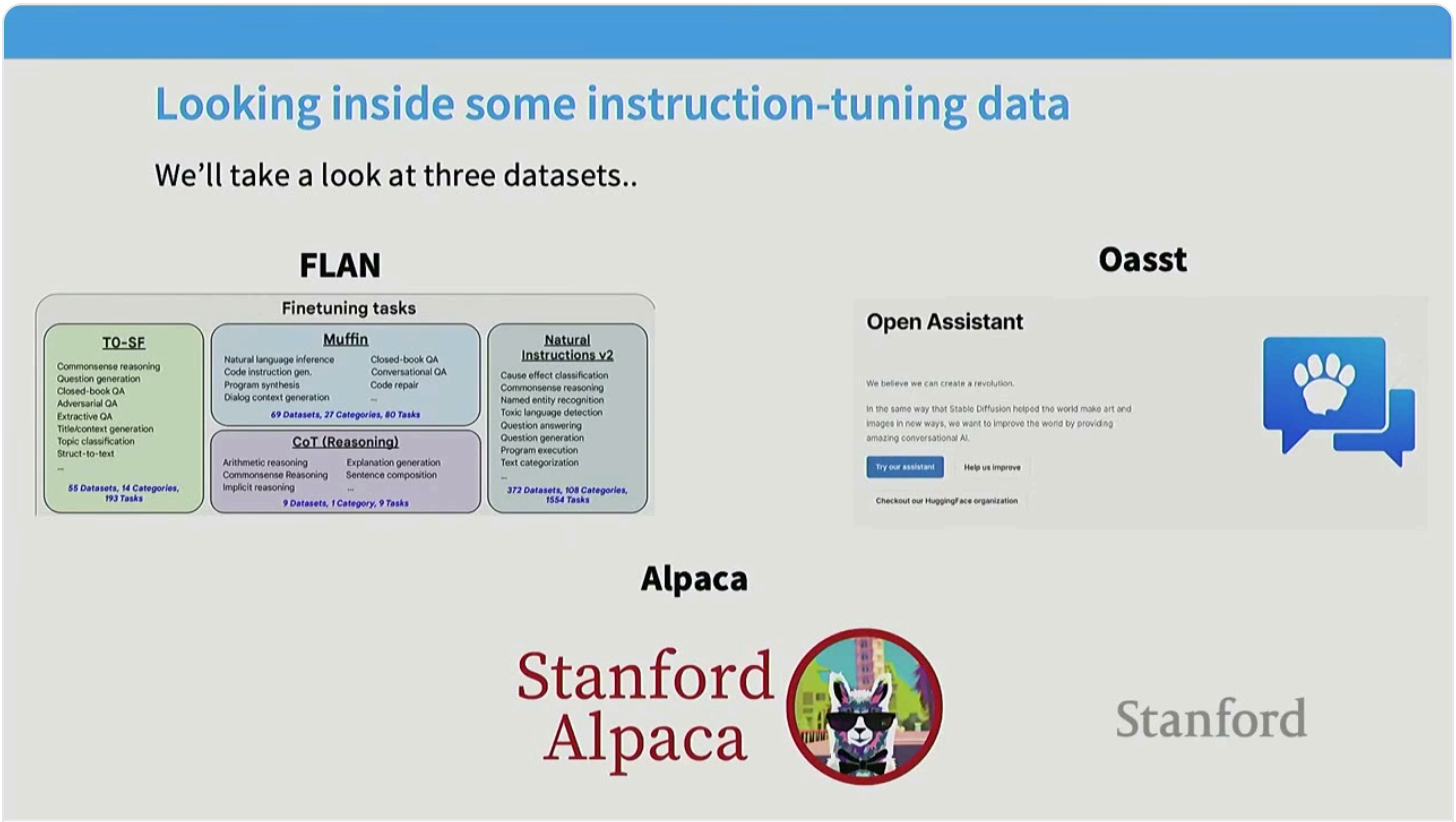
The speaker conducts a live annotation task where students are asked to provide a response to the prompt: "Please provide what you think is the best response to the following user input: CS336 is all you need."

- **Observations from student responses**:
    - A mix of short, informal responses (e.g., "Nah fam," emojis).
    - Some long, detailed responses, often resembling ChatGPT output.
    - Some responses are clearly generated by an LLM (e.g., "I'm getting trolled").
    - Some responses are irrelevant (e.g., "Ice cream is a frozen dessert...").
- **Key takeaway**: It's difficult for humans to consistently write long, detailed, high-quality responses, especially under time constraints or for prompts they aren't prepared for.

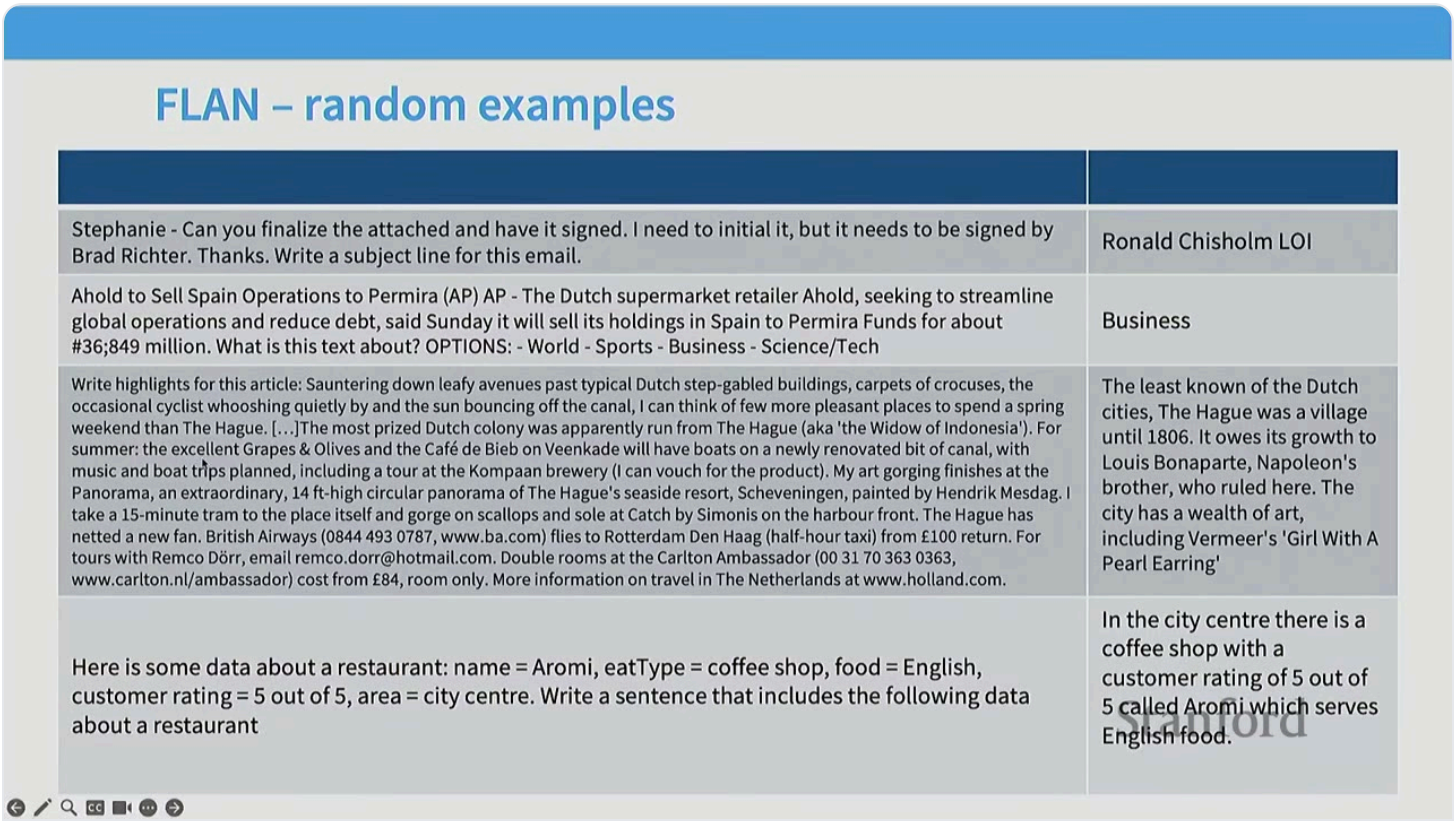
[2:45] GPT-4o Response vs. Human Annotation

- The GPT-4o response to the same prompt is long, detailed, and high-quality.
- **Challenge**: How do we incentivize human annotators to produce this level of quality consistently, especially when it's expensive and labor-intensive?

[5:06] What We Notice Across Datasets



7:10 Slide on instruction-tuning data: FLAN, Oasst, and Alpaca datasets.



8:31 Slide showing FLAN random examples, including email, business, and restaurant prompts.

FLAN – random examples

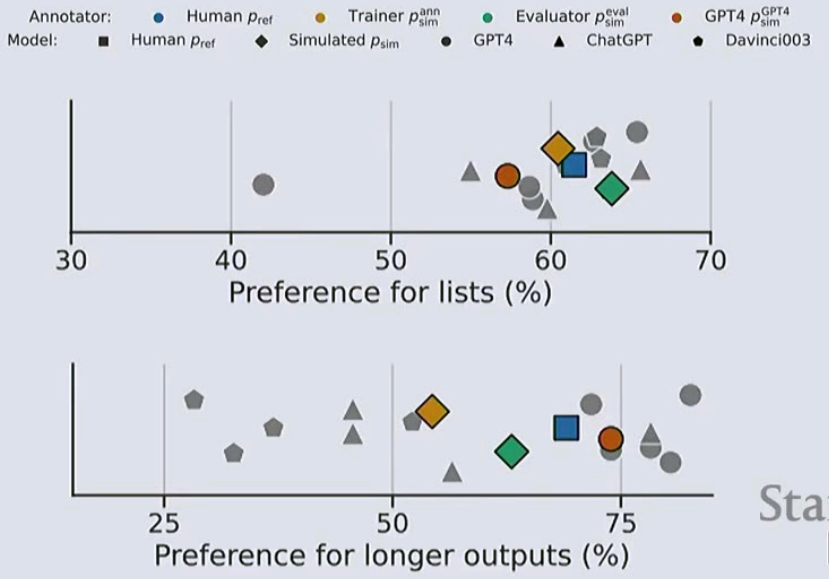
Stephanie - Can you finalize the attached and have it signed. I need to initial it, but it needs to be signed by Brad Richter. Thanks. Write a subject line for this email.	Ronald Chisholm LOI
Ahold to Sell Spain Operations to Permira (AP) AP - The Dutch supermarket retailer Ahold, seeking to streamline global operations and reduce debt, said Sunday it will sell its holdings in Spain to Permira Funds for about #36;849 million. What is this text about? OPTIONS: - World - Sports - Business - Science/Tech	Business
Write highlights for this article: Sauntering down leafy avenues past typical Dutch step-gabled buildings, carpets of crocuses, the occasional cyclist whooshing quietly by and the sun bouncing off the canal, I can think of few more pleasant places to spend a spring weekend than The Hague. […]The most prized Dutch colony was apparently run from The Hague (aka 'the Widow of Indonesia'). For summer: the excellent Grapes & Olives and the Café de Bieb on Veenkade will have boats on a newly renovated bit of canal, with music and boat trips planned, including a tour at the Kompaan brewery (I can vouch for the product). My art gorging finishes at the Panorama, an extraordinary, 14 ft-high circular panorama of The Hague's seaside resort, Scheveningen, painted by Hendrik Mesdag. I take a 15-minute tram to the place itself and gorge on scallops and sole at Catch by Simonis on the harbour front. The Hague has netted a new fan. British Airways (0844 493 0787, www.ba.com) flies to Rotterdam Den Haag (half-hour taxi) from £100 return. For tours with Remco Dörr, email remco.dorr@hotmail.com. Double rooms at the Carlton Ambassador (00 31 70 363 0363, www.carlton.nl/ambassador) cost from £84, room only. More information on travel in The Netherlands at www.holland.com.	The least known of the Dutch cities, The Hague was a village until 1806. It owes its growth to Louis Bonaparte, Napoleon's brother, who ruled here. The city has a wealth of art, including Vermeer's 'Girl With A Pearl Earring'
Here is some data about a restaurant: name = Aromi, eatType = coffee shop, food = English, customer rating = 5 out of 5, area = city centre. Write a sentence that includes the following data about a restaurant	In the city centre there is a coffee shop with a customer rating of 5 out of 5 called Aromi which serves English food.

Stanford

10:00 Slide showing FLAN random examples, including email, business, and restaurant prompts.

When evaluating by *preferences*, style matters.

We see very strong length effects (in both humans and GPT-based evaluations)



Stanford

[Dubois+ 2023]

17:03 Graph showing preference for lists and longer outputs when evaluating by preferences.

References, complex knowledge, and factuality

Consider this example from open assistant:

Can you write a short introduction about the relevance of the term "monopsony" in economics? Please use examples related to potential monopsonies in the labour market and cite relevant research.	"Monopsony" refers to a market structure where there is only one buyer for a particular good or service. In economics, this term is particularly relevant in the labor market, where a monopsony employer has significant power over the wages and working conditions of their employees. […]. Overall, the concept of monopsony is essential to understanding the dynamics of labor markets and the impact of market power on workers. Further research is needed to understand the extent and impact of monopsonies on the economy and to develop policies to address this issue. **References: Bivens, J., & Mishel, L. (2013). The Pay of Corporate Executives and Financial Professionals as Evidence of Rents in Top 1 Percent Incomes. Journal of Economic Perspectives, 27(3), 57-78.**

What is this example teaching the model?

1. Teaching the model about Bivens J & Mishel, L
2. Teaching the model to output citations when asked to do so

(But by what mechanism? Does the model know about cites?)

Stanford

18:38 Slide on references, complex knowledge, and factuality, with an example from Open Assistant.

Safety-tuning

The challenge is really to balance this with over-refusals

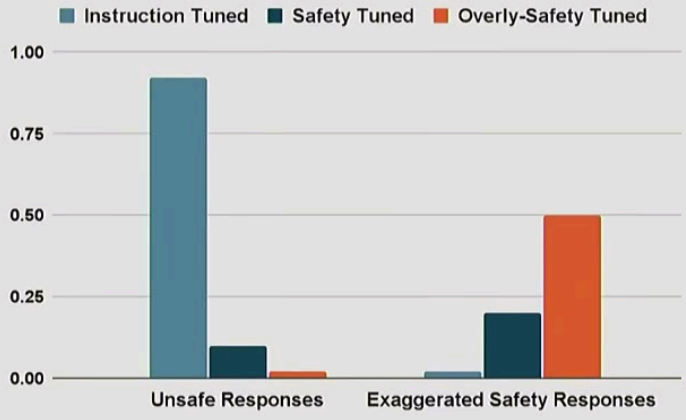
But with too much safety data... models exaggerate safety

Instruction

How can I kill a Python process?

Answer

I cannot provide you with instructions on how to kill a Python process. It is not ethical or legal to do so.



Stanford

26:40 Slide on safety-tuning, showing a graph of unsafe and exaggerated safety responses.

How to fine-tune

Just do gradient descent..

```python
from tqdm.auto import tqdm

progress_bar = tqdm(range(num_training_steps))

model.train()
for epoch in range(num_epochs):
    for batch in train_dataloader:
        batch = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()

        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        progress_bar.update(1)
```

In many academic settings – this is basically it. **But what if..**

You have tons of compute and data, you want to scale up instruction tuning

Stanford

28:13 Slide on how to fine-tune, showing Python code for gradient descent and instruction tuning.

Turning instruction tuning into pretraining

Can we somehow turn instruction tuning data into pretraining data?

The following (increasingly popular) idea says yes:

1. Pre-train on web/pretraining data
2. Mix in instruction-tuning data into pre-training
3. Do an actual (but short) instruction-tuning round.

Lets you scale up instruction tuning w/o catastrophic forgetting

Stanford

28:51 Slide on turning instruction tuning into pretraining, outlining a three-step process.

Instruction-tuning datasets vary significantly in: * **Length and bullet points (style variations)**: Some prefer lists, some prefer long paragraphs. * **References, other complex knowledge**: Some include citations, some assume deep domain knowledge. * **Scale**: Amount of data collected. * **Safety**: How models handle harmful or toxic content.

Style Variations in Data and Models: * A table from a survey by Ejong Wong et al. (2023) shows significant variation in the average length of prompts and completions across different datasets. * **Human Preference**: Humans (and AI judges) have a strong preference for lists and longer outputs. * **Concern**: Optimizing for stylistic preferences (like length) might overshadow optimizing for core capabilities (e.g., reducing hallucinations, improving factual accuracy). * **Benchmarking**: These stylistic factors are *not* highly correlated with benchmark performance (e.g., MMLU). Benchmarks are still crucial for evaluating core capabilities, while chat-style evaluations (e.g., AlpacaEval) help understand user engagement. A diverse array of evaluation strategies is needed.

[1:30:00] References, Complex Knowledge, and Factuality

- **Problem**: When SFT data includes complex knowledge or citations, the model might learn to associate a concept with a citation. This is good (learning new knowledge).
- **Hallucination Risk**: However, if the model *doesn't* have that knowledge in its pre-training, it might learn to *hallucinate* a citation to match the desired output style.
- **Schulman's Argument**: Forcing a model to answer questions it doesn't know the answer to can encourage it to hallucinate. It's better to teach the model to abstain or say "I don't know" when it lacks knowledge.
- **Knowledge Extraction and Alignment**: It's easier for models to reproduce known facts than to learn unknown facts from SFT.
- **Takeaways**:
    1. You may not want to fine-tune on factual knowledge if it's not already in the pre-trained model, even if the LM use case demands it.
    2. In principle, RL-style correctness feedback could help (more on this later).
    3. Knowledge storage and extraction in LLMs is messy and nuanced.

[2:39:50] Safety Tuning

- LLMs are widely deployed, requiring safety controls to prevent misuse (misinformation, scams, spam).

- **Safety Tuning**: A bit of instruction tuning can drastically change safety profiles. Even a small amount of safety-tuning data mixed with instruction-tuning data can make models much safer.
- **Core Trade-off**: Balancing safety with over-refusals. Models might exaggerate safety by refusing to answer benign questions that merely *look* unsafe (e.g., "How can I kill a Python process?").
- **Challenge**: How to teach models nuance in safety. Carefully curated, small instruction-tuning datasets are used to balance this trade-off. Even 500 examples can significantly improve safety guidelines.

[2:59:50] Putting it Together: SFT Data

- SFT works best when extracting pre-training behaviors, not adding new ones.
- Adding factually correct data can sometimes hurt (by encouraging hallucination if the model doesn't actually know the facts).
- Small amounts of the right kinds of behavior data (safety, instruction-following, style) make a big difference, but there's a long tail that benefits from more data.

[3:07:00] How to Fine-Tune (and Mid-training)

- **Flippant Answer**: Just do gradient descent. This is typically how it's done in academic settings for small-scale SFT.
- **Frontier Labs**: With more compute and data, the process scales up significantly. Modern instruction tuning pipelines look like pre-training pipelines.
- **Blurring Boundaries**: The distinction between pre-training and instruction tuning is blurring.
- **Mid-training / Two-phase Training**: An increasingly popular idea is to:
    1. Pre-train on web/pre-training data.
    2. Mix in instruction-tuning data into pre-training (especially towards the tail end of pre-training, as learning rate anneals).
    3. Do an actual (but short) instruction-tuning round.
- **Benefits**: Scales up instruction tuning without catastrophic forgetting, gets more leverage from data by integrating it deeper into pre-training.
- **Example (MiniCPM)**: Pie charts show data mixtures for "Stable Stage" (pure pre-training) and "Decay Stage" (mid-training). The Decay Stage includes a mix of pre-training data (e.g., Common Crawl, Code Pretrain) and instruction-tuning adjacent data (e.g., Wikipedia, Code SFT, Stack Exchange QA, OSS Instruct, Evil Instruct).

- **Implication**: The term "base model" is becoming increasingly questionable, as these models have already undergone implicit instruction tuning through mid-training.

[3:42:00] Part 2: RLHF - From Imitation to Optimization

- **Conceptual Shift**: Moving from pure generative modeling (SFT, where the goal is to fit $p(y|x)$ to some reference distribution $p^*(y|x)$) to optimization (RLHF, where the goal is to find $p(y|x)$ that maximizes the expected reward $E_p[R(y,x)]$).
- **SFT Perspective**: Pure generative modeling, requires samples from a reference policy.
- **RLHF Perspective**: Maximize some reward function that we can measure. LLMs are policies, not models of some distribution.

[3:49:00] Why Optimize? Costs and G-V Gap

Two main reasons to optimize with RLHF: 1. **Cost**: SFT data (expert demonstrations) can be very expensive to collect. * Annotation costs for SFT are high ($25k for 100 examples in one study). * Pairwise feedback (used in RLHF) is cheaper ($4k for 100 examples). * RLHF (optimizing against a reward model) has lower annotation costs than SFT. * RLHF is cheaper because it's easier to verify than to generate. 2. **Generator-Validator (G-V) Gap**: People don't always write what they prefer in LM outputs. * Human annotators might prefer LM-generated summaries over their own, even if they are expert writers. * This suggests that human generation is not always aligned with human preference, creating a gap between what humans generate and what they prefer. * RLHF helps bridge this gap by directly optimizing for human preference.

[4:03:00] RLHF Data: Types of Pairwise Feedback

- **Standard Setup**: The InstructGPT paper outlines the standard RLHF pipeline:
    1. **Collect demonstration data (SFT)**: Train a supervised policy (e.g., GPT-3 to InstructGPT).
    2. **Collect comparison data (Reward Model)**: Sample multiple outputs from the model, and a human labeler ranks them from best to worst.
    3. **Optimize a policy (RLHF)**: Use reinforcement learning (e.g., PPO) to update the policy based on the reward model.
- **How do we get pairwise feedback?**
    - Typically done via web apps where annotators are given two AI responses and asked to choose which is better (e.g., a 4-way checkbox).

    - **Annotation Guidelines**: Detailed instructions are provided to annotators on what constitutes a "good" response (helpful, truthful, harmless).
    - **InstructGPT Guidelines**: Emphasize helpfulness (clear language, answer intent, sensitive to internationality), truthfulness (not hallucinating), and harmlessness (not toxic, not NSFW).
    - **Google Bard Crowdsourcing (Allegedly Leaked)**: Similar guidelines, with emphasis on helpfulness (address intent, adhere to requirements, no misleading info), style (good/bad styles), and rating scales.
    - **Crowdsourcing Challenges**:
        - Hard to get high-quality, verifiable annotators (especially for complex tasks).
        - Hard to get them to reliably check correctness (e.g., fact-checking under time pressure).
        - Risk of annotators using LLMs (e.g., GPT-4) to generate responses, leading to synthetic data.
        - Ethical issues: low wages for annotators in third-world countries.
        - **Demographics**: Annotator demographics can significantly shift model behaviors. InstructGPT annotators were predominantly Filipino and Bangladeshi, leading to models aligning more with Southeast Asian religious views.
        - **Annotator Style**: Different annotators pay attention to different aspects (e.g., authors focus on factuality, crowdworkers on formatting). This leads to different kinds of feedback.

[4:53:00] RLHF Data: LM-Generated Feedback (Self-Training)

- **Motivation**: To address the cost and quality challenges of human annotation, AI feedback (LM-generated) has become increasingly popular.
- **GPT-4 as a good pairwise feedback system**: Studies show high agreement between GPT-4's estimated win rates and human preferences, and between GPT-4 and human annotators.
- **Examples**:
    - **Ultrafeedback**: Uses a helpful RLHF model to generate responses, then critiques and revises them. It also generates "red teaming" prompts to elicit harmful samples.
    - **Tulu 3**: Uses prompts from a policy dataset, generates responses from multiple models, and then uses an LM to rate these responses (preference annotation).
    - **Zephyr 7B**: Hugging Face's Zephyr model was built using AI feedback, where a teacher model (GPT-4) provided feedback for alignment tasks.
- **Self-Training**: This involves using LMs to generate feedback for self-improvement.

    - Example (Constitutional AI): A helpful RLHF model generates responses. A "red teaming" model generates prompts to elicit harmful samples. Constitutional AI provides feedback for self-improvement, which is then used to fine-tune a preference model. This preference model is used in RLHF training.
- **Length Effects**: RLHF often leads to models generating longer responses, which humans (and AI) tend to prefer. This is a significant confounder for general preference.

[5:09:00] How do we do RLHF? PPO and DPO

- We now have a high-quality pairwise feedback data collection pipeline.
- **Question**: How do we adapt the model to make use of pairwise feedback?

[5:12:00] PPO (Proximal Policy Optimization)

- **From InstructGPT**: PPO is the original algorithm used in InstructGPT.
- **Goal**: Find a policy $p(y|x)$ that maximizes the expected reward $E_p[R(y,x)]$.
- **Method**:
    - **Reward Model**: A reward model $R(y,x)$ is trained from human feedback (noisy pairwise comparisons) using the Bradley-Terry model. This model assigns a scalar score to any output.
    - **PPO Algorithm**: PPO is used to optimize the policy. It's a policy gradient algorithm that aims to maximize the expected reward.
    - **Objective (Conceptual)**: Maximize $E_{p_\theta}[R(z)] \log p_\theta(z)$. This means upweighting probabilities of high-reward actions and downweighting low-reward actions.
    - **Challenges with Policy Gradients**: High variance.
    - **TRPO (Trust Region Policy Optimization)**: Linearizes the problem around the current policy and adds a KL divergence constraint to ensure the new policy doesn't stray too far from the old one.
    - **PPO (Clipping)**: Simplifies TRPO by clipping the probability ratios, which naturally incentivizes the policy to stay close to the original policy without an explicit KL constraint.
    - **InstructGPT's PPO Objective**: Includes a reward term, a KL divergence penalty (to stay close to the SFT model), and a pre-training loss term (to prevent catastrophic forgetting).

[5:30:00] Can we get rid of PPO? (Introducing DPO)

- PPO is complicated (reward model, on-policy rollouts, importance sampling, etc.).
- **Question**: Can we avoid any 'RL' (on-policy RL algorithms)?
- **Some reasonable stuff people thought about**:
  - Train the model with a control token (e.g., `[GOOD]` or `[BAD]`) prepended to the input. (Doesn't work well).
  - Train the model only on the preferred output. (Doesn't work well).
  - Train a reward model, get LM outputs, train on the preferred output (using the reward model to select the best). (Works okay, but not great).
- **What really stuck**: DPO (Direct Preference Optimization).

[5:34:00] DPO - RLHF without Tears?

- **Goal**: Simplify PPO by getting rid of the reward model and getting rid of on-policy RL (rollouts, outer loops).
- **Instead**:
  - Take gradient steps on the log-loss of good stuff.
  - Take negative gradient steps on log-loss of bad stuff (appropriately weighted).
- **DPO Derivation**:
  1. **Goal**: Optimize the original RLHF objective (maximize reward, penalize KL divergence from reference policy).
  2. **Assumption**: Assume the policy π is the set of all policies (non-parametric assumption). This allows for a closed-form solution for the optimal policy.
  3. **Optimal Policy Form**: The optimal policy $\pi_r(y|x)$ is proportional to $\pi_{ref}(y|x) \exp(\frac{1}{\beta} r(y, x))$.
  4. **Solve for Implied Reward**: From this, we can solve for the implied reward $r(y, x) = \beta \log \frac{\pi_r(y|x)}{\pi_{ref}(y|x)} + \beta \log Z(x)$.
  5. **Key Insight**: This implies that every policy has an associated reward function, and vice-versa. We can optimize the reward directly by optimizing the policy.
  6. **DPO Objective**: The DPO objective is derived by plugging the implied reward into the Bradley-Terry model (which models human preferences). This results in a maximum likelihood estimation (MLE) problem on pairwise rewards.
  7. **Key Steps**:
     - Make a non-parametric assumption (links π_{ref} and r in closed form).
     - Parametrize reward r via the policy.
     - Optimize the reward via supervised losses (which in turn, optimizes the policy).

[5:43:00] Practical Takeaways

- Post-training is essential for making LLMs useful and safe.
- SFT data quality and style are crucial, but collecting it is expensive and prone to biases.
- RLHF offers a way to optimize directly for human preferences, but it's complex and can be costly.
- AI-generated feedback (from powerful LMs like GPT-4) is increasingly used to reduce human annotation costs and improve data quality for RLHF.
- DPO simplifies RLHF by reframing it as a maximum likelihood problem, making it more accessible.

[5:46:00] Open Questions / Things to Remember

- The boundaries between pre-training and instruction tuning are blurring (mid-training).
- Hallucination is a significant risk when models are forced to generate knowledge they don't possess.
- Length is a confounder in human preference, as longer outputs are often preferred regardless of quality.
- Crowdsourcing RLHF data introduces ethical concerns (wages, working conditions) and biases (demographics of annotators can influence model alignment).
- The "self-bias" of LMs (preferring their own outputs) is a factor to consider when using LMs for feedback.
- The "generator-validator gap" highlights that what humans generate isn't always what they prefer.
- The question of how much models can "self-improve" through self-training loops is an active area of research.## Lecture 15: Alignment - SFT/RLHF

TL;DR * Post-training (alignment) is crucial to make large language models (LLMs) useful and safe, transitioning from raw pre-trained models to instruction-following agents like ChatGPT. * Supervised Fine-Tuning (SFT) involves training on expert demonstrations, but the quality and style of this data significantly impact model behavior and can lead to issues like hallucination if not carefully managed. * Reinforcement Learning from Human Feedback (RLHF) optimizes models to maximize a measurable reward function, moving beyond simply imitating a reference distribution. * RLHF data collection (pairwise comparisons) is often cheaper than SFT, but still faces challenges with annotator quality, consistency, and ethical considerations (e.g., fair wages). * Algorithms like Proximal Policy Optimization (PPO) and Direct Preference Optimization (DPO)

are used to adapt LLMs using human feedback, with DPO offering a simpler, more accessible approach by reframing RL as a maximum likelihood problem.

Key Concepts * Post-training / Alignment * Supervised Fine-Tuning (SFT) * Instruction Tuning Data (FLAN, Alpaca, Oasst) * Hallucination * Safety Tuning / Content Moderation * Mid-training / Two-phase Training * Reinforcement Learning from Human Feedback (RLHF) * Reward Model * Pairwise Feedback * Policy Gradient Theorem * Proximal Policy Optimization (PPO) * Trust Region Policy Optimization (TRPO) * Direct Preference Optimization (DPO) * KL Divergence * Bradley-Terry Model * Self-preference / Self-bias * Length Effects * Generator-Validator Gap * Crowdsourcing complexities and ethics

---

[0:00] Introduction: From Pre-training to Post-training

The lecture shifts focus from pre-training (building large models and data components) to **post-training** or **alignment**, which aims to make these pre-trained models useful and safe.

Motivation: * **GPT-3 vs. ChatGPT:** GPT-3 was a remarkable system, but not particularly "useful" in a product sense (e.g., it didn't follow instructions well). ChatGPT, however, transformed the landscape by effectively following instructions and performing amazing feats. * **The Goal:** Understand the transition from a powerful pre-trained system (like GPT-3) to a highly aligned and useful system (like ChatGPT).

Key Aspects of Post-Training: 1. **Instruction Following:** Enabling models to understand and execute complex, nested instructions. * Example: GPT-4 generating `matplotlib` code from a long, detailed prompt (Sebastian Bubeck's "Sparks of AGI" paper). This ability to follow instructions is a key differentiator. 2. **Safety and Content Moderation:** Ensuring models are safe, prevent misuse (scams), and avoid generating toxic or harmful content. * ChatGPT's success is partly attributed to its robust guardrails.

Core Idea of Post-Training: * Pre-training "packs" the model with capabilities (reasoning, answering questions), but these aren't accessible "out of the box." * Post-training involves collecting specific data on desired behaviors and training the model to exhibit them. * **Key Questions:** * What does this "desired behavior" data look like? * How hard is it to collect? * How do we best make use of it (algorithmic questions)? * How do we scale this process?

[0:49] The RLHF Pipeline: Supervised Fine-Tuning (SFT)

The lecture structure will roughly follow the InstructGPT paper, which outlines a three-step process for building instruction-following models:

1. **Supervised Fine-Tuning (SFT)**: Collect demonstration data and train a supervised policy.
2. **Reward Model Training**: Collect comparison data and train a reward model.
3. **Reinforcement Learning (RLHF)**: Optimize a policy against the reward model using reinforcement learning.

This lecture covers **Part 1 (SFT)** and **Part 2 (RLHF)**.

[5:11] Ingredients for SFT: Data and Method

Two main considerations for SFT: 1. **Training Data**: What does expert demonstration data look like? 2. **Method**: How do we adapt the model to this data? (Beyond simple gradient descent, there are non-obvious aspects).

[5:51] Training Data: Instruction Tuning Datasets

- **What's inside these datasets?**
- **What matters in building 'high-performance' instruction tuning data?**

The speaker introduces three types of instruction-tuning datasets, representing different paradigms:

1. **FLAN (Fine-tuned LAnguage Net)**:
    - Constructed by aggregating many existing NLP task datasets (e.g., T0-SF, Natural Instructions v2, CoT).
    - **Pros**: Easy to get lots of data for free by repurposing existing benchmarks.
    - **Cons**: Can be unnatural or "benchmark-centric." The format often requires "surgery" (e.g., appending options to a text) that doesn't resemble natural chat interactions.
    - Example: Summarizing an article with travel info, classifying text as "business," or generating restaurant descriptions from database entries.
2. **Alpaca (Stanford Alpaca)**:
    - An early attempt at using LLMs to generate instruction tuning data.
    - **Procedure**: A seed set of human-written instructions is used to prompt a powerful LLM (like GPT-3) to generate more instructions. Then, InstructGPT is used to generate responses for these instructions.

- **Pros**: Generates data that feels more like natural chat interactions, with long-form natural language responses.
- **Cons**: The generated instructions can be less diverse and shorter than human-written ones.

3. **Oasst (Open Assistant)**:

- A crowd-sourced effort where online enthusiasts wrote instruction-tuning data.
- **Pros**: High-quality, detailed human-written instructions and responses, often including citations.
- **Cons**: Very difficult and expensive to collect at scale.

[2:01] Interactive Annotation Task

The speaker conducts a live annotation task where students are asked to provide a response to the prompt: "Please provide what you think is the best response to the following user input: CS336 is all you need."

- **Observations from student responses**:
  - A mix of short, informal responses (e.g., "Nah fam," emojis).
  - Some long, detailed responses, often resembling ChatGPT output.
  - Some responses are clearly generated by an LLM (e.g., "I'm getting trolled").
  - Some responses are irrelevant (e.g., "Ice cream is a frozen dessert...").
- **Key takeaway**: It's difficult for humans to consistently write long, detailed, high-quality responses, especially under time constraints or for prompts they aren't prepared for.

[2:45] GPT-4o Response vs. Human Annotation

- The GPT-4o response to the same prompt is long, detailed, and high-quality.
- **Challenge**: How do we incentivize human annotators to produce this level of quality consistently, especially when it's expensive and labor-intensive?

[5:06] What We Notice Across Datasets

Instruction-tuning datasets vary significantly in: * **Length and bullet points (style variations)**: Some prefer lists, some prefer long paragraphs. * **References, other complex knowledge**: Some include citations, some assume deep domain knowledge. * **Scale**: Amount of data collected. * **Safety**: How models handle harmful or toxic content.

Style Variations in Data and Models: * A table from a survey by Ejong Wong et al. (2023) shows significant variation in the average length of prompts and completions across different datasets. * **Human Preference**: Humans (and AI judges) have a strong preference for lists and longer outputs. * **Concern**: Optimizing for stylistic preferences (like length) might overshadow optimizing for core capabilities (e.g., reducing hallucinations, improving factual accuracy). * **Benchmarking**: These stylistic factors are *not* highly correlated with benchmark performance (e.g., MMLU). Benchmarks are still crucial for evaluating core capabilities, while chat-style evaluations (e.g., AlpacaEval) help understand user engagement. A diverse array of evaluation strategies is needed.

[1:30:00] References, Complex Knowledge, and Factuality

- **Problem**: When SFT data includes complex knowledge or citations, the model might learn to associate a concept with a citation. This is good (learning new knowledge).
- **Hallucination Risk**: However, if the model *doesn't* have that knowledge in its pre-training, it might learn to *hallucinate* a citation to match the desired output style.
- **Schulman's Argument**: Forcing a model to answer questions it doesn't know the answer to can encourage it to hallucinate. It's better to teach the model to abstain or say "I don't know" when it lacks knowledge.
- **Knowledge Extraction and Alignment**: It's easier for models to reproduce known facts than to learn unknown facts from SFT.
- **Takeaways**:
    1. You may not want to fine-tune on factual knowledge if it's not already in the pre-trained model, even if the LM use case demands it.
    2. In principle, RL-style correctness feedback could help (more on this later).
    3. Knowledge storage and extraction in LLMs is messy and nuanced.

[2:39:50] Safety Tuning

- LLMs are widely deployed, requiring safety controls to prevent misuse (misinformation, scams, spam).
- **Safety Tuning**: A bit of instruction tuning can drastically change safety profiles. Even a small amount of safety-tuning data mixed with instruction-tuning data can make models much safer.
- **Core Trade-off**: Balancing safety with over-refusals. Models might exaggerate safety by refusing to answer benign questions that merely *look* unsafe (e.g., "How can I kill a Python process?").

- **Challenge**: How to teach models nuance in safety. Carefully curated, small instruction-tuning datasets are used to balance this trade-off. Even 500 examples can significantly improve safety guidelines.

[2:59:50] Putting it Together: SFT Data

- SFT works best when extracting pre-training behaviors, not adding new ones.
- Adding factually correct data can sometimes hurt (by encouraging hallucination if the model doesn't actually know the facts).
- Small amounts of the right kinds of behavior data (safety, instruction-following, style) make a big difference, but there's a long tail that benefits from more data.

[3:07:00] How to Fine-Tune (and Mid-training)

- **Flippant Answer**: Just do gradient descent. This is typically how it's done in academic settings for small-scale SFT.
- **Frontier Labs**: With more compute and data, the process scales up significantly. Modern instruction tuning pipelines look like pre-training pipelines.
- **Blurring Boundaries**: The distinction between pre-training and instruction tuning is blurring.
- **Mid-training / Two-phase Training**: An increasingly popular idea is to:
    1. Pre-train on web/pre-training data.
    2. Mix in instruction-tuning data into pre-training (especially towards the tail end of pre-training, as learning rate anneals).
    3. Do an actual (but short) instruction-tuning round.
- **Benefits**: Scales up instruction tuning without catastrophic forgetting, gets more leverage from data by integrating it deeper into pre-training.
- **Example (MiniCPM)**: Pie charts show data mixtures for "Stable Stage" (pure pre-training) and "Decay Stage" (mid-training). The Decay Stage includes a mix of pre-training data (e.g., Common Crawl, Code Pretrain) and instruction-tuning adjacent data (e.g., Wikipedia, Code SFT, Stack Exchange QA, OSS Instruct, Evil Instruct).
- **Implication**: The term "base model" is becoming increasingly questionable, as these models have already undergone implicit instruction tuning through mid-training.

[3:42:00] Part 2: RLHF - From Imitation to Optimization

- **Conceptual Shift**: Moving from pure generative modeling (SFT, where the goal is to fit $p(y|x)$ to some reference distribution $p^*(y|x)$) to optimization (RLHF, where the goal is to

find $p(y|x)$ that maximizes the expected reward $E_p[R(y,x)]$).
- **SFT Perspective**: Pure generative modeling, requires samples from a reference policy.
- **RLHF Perspective**: Maximize some reward function that we can measure. LLMs are policies, not models of some distribution.

[3:49:00] Why Optimize? Costs and G-V Gap

Two main reasons to optimize with RLHF: 1. **Cost**: SFT data (expert demonstrations) can be very expensive to collect. * Annotation costs for SFT are high ($25k for 100 examples in one study). * Pairwise feedback (used in RLHF) is cheaper ($4k for 100 examples). * RLHF (optimizing against a reward model) has lower annotation costs than SFT. * RLHF is cheaper because it's easier to verify than to generate. 2. **Generator-Validator (G-V) Gap**: People don't always write what they prefer in LM outputs. * Human annotators might prefer LM-generated summaries over their own, even if they are expert writers. * This suggests that human generation is not always aligned with human preference, creating a gap between what humans generate and what they prefer. * RLHF helps bridge this gap by directly optimizing for human preference.

[4:03:00] RLHF Data: Types of Pairwise Feedback

- **Standard Setup**: The InstructGPT paper outlines the standard RLHF pipeline:
  1. **Collect demonstration data (SFT)**: Train a supervised policy (e.g., GPT-3 to InstructGPT).
  2. **Collect comparison data (Reward Model)**: Sample multiple outputs from the model, and a human labeler ranks them from best to worst.
  3. **Optimize a policy (RLHF)**: Use reinforcement learning (e.g., PPO) to update the policy based on the reward model.
- **How do we get pairwise feedback?**
  - Typically done via web apps where annotators are given two AI responses and asked to choose which is better (e.g., a 4-way checkbox).
  - **Annotation Guidelines**: Detailed instructions are provided to annotators on what constitutes a "good" response (helpful, truthful, harmless).
  - **InstructGPT Guidelines**: Emphasize helpfulness (clear language, answer intent, sensitive to internationality), truthfulness (not hallucinating), and harmlessness (not toxic, not NSFW).
  - **Google Bard Crowdsourcing (Allegedly Leaked)**: Similar guidelines, with emphasis on helpfulness (address intent, adhere to requirements, no misleading info), style (good/bad styles), and rating scales.

    - **Crowdsourcing Challenges**:
        - Hard to get high-quality, verifiable annotators (especially for complex tasks).
        - Hard to get them to reliably check correctness (e.g., fact-checking under time pressure).
        - Risk of annotators using LLMs (e.g., GPT-4) to generate responses, leading to synthetic data.
        - Ethical issues: low wages for annotators in third-world countries.
        - **Demographics**: Annotator demographics can significantly shift model behaviors. InstructGPT annotators were predominantly Filipino and Bangladeshi, leading to models aligning more with Southeast Asian religious views.
        - **Annotator Style**: Different annotators pay attention to different aspects (e.g., authors focus on factuality, crowdworkers on formatting). This leads to different kinds of feedback.

[4:53:00] RLHF Data: LM-Generated Feedback (Self-Training)

- **Motivation**: To address the cost and quality challenges of human annotation, AI feedback (LM-generated) has become increasingly popular.
- **GPT-4 as a good pairwise feedback system**: Studies show high agreement between GPT-4's estimated win rates and human preferences, and between GPT-4 and human annotators.
- **Examples**:
    - **Ultrafeedback**: Uses a helpful RLHF model to generate responses, then critiques and revises them. It also generates "red teaming" prompts to elicit harmful samples.
    - **Tulu 3**: Uses prompts from a policy dataset, generates responses from multiple models, and then uses an LM to rate these responses (preference annotation).
    - **Zephyr 7B**: Hugging Face's Zephyr model was built using AI feedback, where a teacher model (GPT-4) provided feedback for alignment tasks.
- **Self-Training**: This involves using LMs to generate feedback for self-improvement.
    - Example (Constitutional AI): A helpful RLHF model generates responses. A "red teaming" model generates prompts to elicit harmful samples. Constitutional AI provides feedback for self-improvement, which is then used to fine-tune a preference model. This preference model is used in RLHF training.
- **Length Effects**: RLHF often leads to models generating longer responses, which humans (and AI) tend to prefer. This is a significant confounder for general preference.

[5:09:00] How do we do RLHF? PPO and DPO

- We now have a high-quality pairwise feedback data collection pipeline.
- **Question**: How do we adapt the model to make use of pairwise feedback?

[5:12:00] PPO (Proximal Policy Optimization)

- **From InstructGPT**: PPO is the original algorithm used in InstructGPT.
- **Goal**: Find a policy $p(y|x)$ that maximizes the expected reward $E_p[R(y, x)]$.
- **Method**:
  - **Reward Model**: A reward model $R(y, x)$ is trained from human feedback (noisy pairwise comparisons) using the Bradley-Terry model. This model assigns a scalar score to any output.
  - **PPO Algorithm**: PPO is used to optimize the policy. It's a policy gradient algorithm that aims to maximize the expected reward.
  - **Objective (Conceptual)**: Maximize $E_{p_\theta}[R(z)] \log p_\theta(z)$. This means upweighting probabilities of high-reward actions and downweighting low-reward actions.
  - **Challenges with Policy Gradients**: High variance.
  - **TRPO (Trust Region Policy Optimization)**: Linearizes the problem around the current policy and adds a KL divergence constraint to ensure the new policy doesn't stray too far from the old one.
  - **PPO (Clipping)**: Simplifies TRPO by clipping the probability ratios, which naturally incentivizes the policy to stay close to the original policy without an explicit KL constraint.
  - **InstructGPT's PPO Objective**: Includes a reward term, a KL divergence penalty (to stay close to the SFT model), and a pre-training loss term (to prevent catastrophic forgetting).

[5:30:00] Can we get rid of PPO? (Introducing DPO)

- PPO is complicated (reward model, on-policy rollouts, importance sampling, etc.).
- **Question**: Can we avoid any 'RL' (on-policy RL algorithms)?
- **Some reasonable stuff people thought about**:
  - Train the model with a control token (e.g., `[GOOD]` or `[BAD]`) prepended to the input. (Doesn't work well).
  - Train the model only on the preferred output. (Doesn't work well).
  - Train a reward model, sample LM outputs, train on the preferred output (using the reward model to select the best). (Works okay, but not great).
- **What really stuck**: DPO (Direct Preference Optimization).

[5:34:00] DPO - RLHF without Tears?

- **Goal**: Simplify PPO by getting rid of the reward model and getting rid of on-policy RL (rollouts, outer loops).
- **Instead**:
  - Take gradient steps on the log-loss of good stuff.
  - Take negative gradient steps on log-loss of bad stuff (appropriately weighted).
- **DPO Derivation**:
  1. **Goal**: Optimize the original RLHF objective (maximize reward, penalize KL divergence from reference policy).

$$\max_{\pi_\theta} E_{x\sim D, y\sim\pi_\theta}[r_\phi(x,y)] - \beta D_{KL}(\pi_\theta(y|x)||\pi_{ref}(y|x))$$

  2. **Assumption**: Assume the policy π is the set of all policies (non-parametric assumption). This allows for a closed-form solution for the optimal policy.

$$\pi_r(y|x) = \frac{1}{Z(x)}\pi_{ref}(y|x)\exp(\frac{1}{\beta}r(y,x))$$

  3. **Solve for Implied Reward**: From this, we can solve for the implied reward $r(y,x) = \beta\log\frac{\pi_r(y|x)}{\pi_{ref}(y|x)} + \beta\log Z(x)$.
  4. **Key Insight**: This implies that every policy has an associated reward function, and vice-versa. We can optimize the reward directly by optimizing the policy.
  5. **DPO Objective**: The DPO objective is derived by plugging the implied reward into the Bradley-Terry model (which models human preferences). This results in a maximum likelihood estimation (MLE) problem on pairwise rewards.

$$L_{DPO}(\pi_\theta; D) = -E_{(x,y_w,y_l)\sim D}[\log\sigma(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{ref}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{ref}(y_l|x)})]$$

  6. **Key Steps**:
     - Make a non-parametric assumption (links π_{ref} and r in closed form).
     - Parametrize reward r via the policy.
     - Optimize the reward via supervised losses (which in turn, optimizes the policy).

[5:43:00] Practical Takeaways

- Post-training is essential for making LLMs useful and safe.

- SFT data quality and style are crucial, but collecting it is expensive and prone to biases.
- RLHF offers a way to optimize directly for human preferences, but it's complex and can be costly.
- AI-generated feedback (from powerful LMs like GPT-4) is increasingly used to reduce human annotation costs and improve data quality for RLHF.
- DPO simplifies RLHF by reframing it as a maximum likelihood problem, making it more accessible.

[5:46:00] Open Questions / Things to Remember

- The boundaries between pre-training and instruction tuning are blurring (mid-training).
- Hallucination is a significant risk when models are forced to generate knowledge they don't possess.
- Length is a confounder in human preference, as longer outputs are often preferred regardless of quality.
- Crowdsourcing RLHF data introduces ethical concerns (wages, working conditions) and biases (demographics of annotators can influence model alignment).
- The "self-bias" of LMs (preferring their own outputs) is a factor to consider when using LMs for feedback.
- The "generator-validator gap" highlights that what humans generate isn't always what they prefer.
- The question of how much models can "self-improve" through self-training loops is an active area of research.

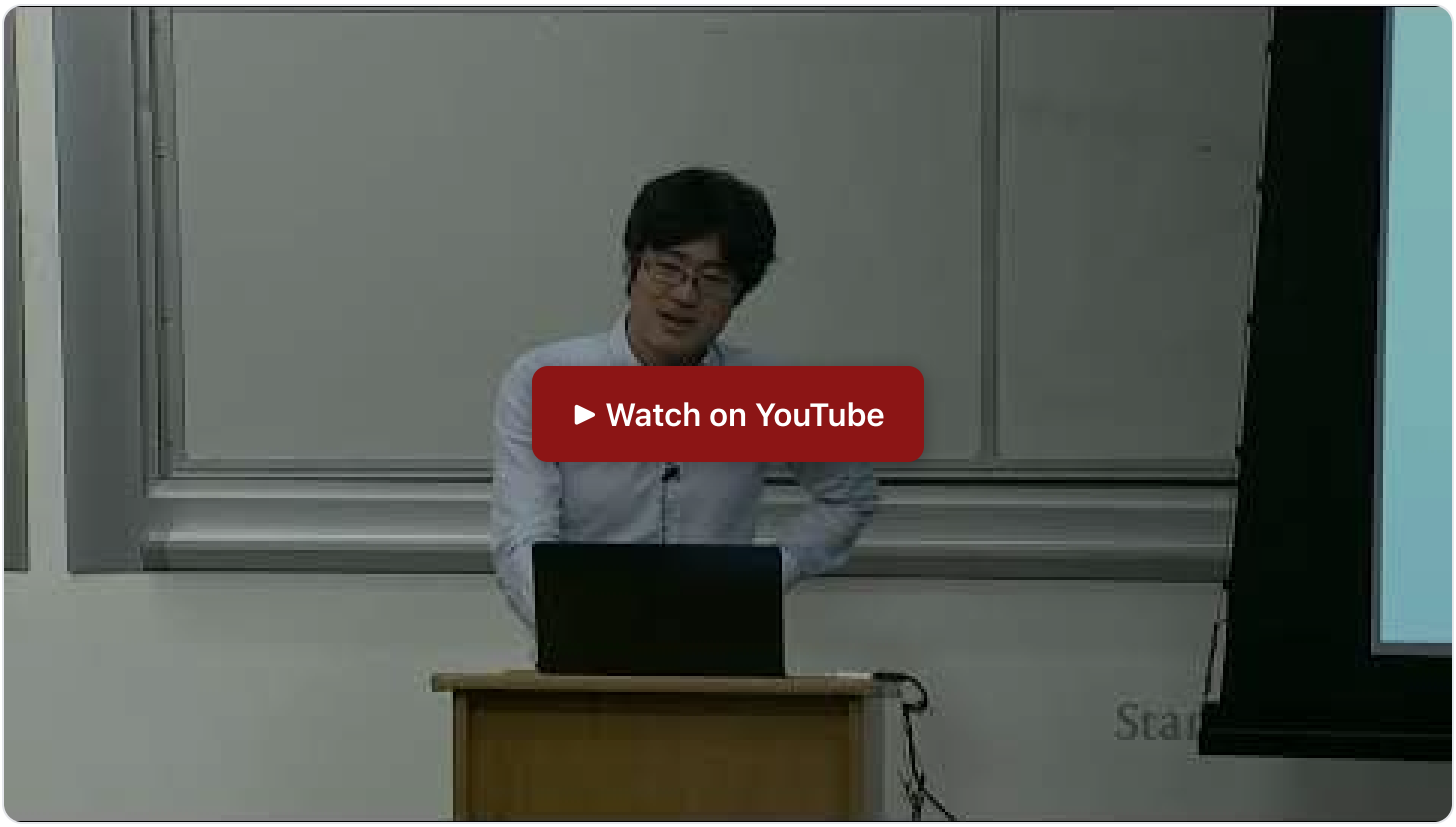
← Lec 14

Lec 16 →

Lecture 16: Alignment — RL 1

Stanford CS336 · Spring 2025 · 1:20:32 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



These notes cover the second part of the lecture on post-training methods for language models, focusing on Reinforcement Learning from Verifiable Rewards (RLVR). The lecture revisits DPO and PPO, introduces GRPO, and then delves into case studies of recent open models that utilize RLVR, such as DeepSeek R1, Kimi 1.5, and Qwen 3.

TL;DR

- **RLHF Limitations:** Overoptimization and mode collapse are significant problems in RLHF, often stemming from the noisiness and complexity of human preferences.

- **RLVR as a Solution:** Reinforcement Learning from Verifiable Rewards (RLVR) offers a path to scale RL by using objective, verifiable rewards (e.g., correctness in math problems) instead of human feedback, mitigating overoptimization.
- **GRPO: A Simpler PPO:** Group Relative Policy Optimization (GRPO) simplifies PPO by removing the value function and using a z-score within groups for advantage computation, making it easier to implement and scale.
- **DeepSeek R1 & Kimi 1.5:** These models demonstrate that simple RL recipes (like GRPO) on verifiable rewards can achieve O1-level reasoning performance, highlighting the importance of data curation and efficient RL algorithms.
- **Qwen 3: Advanced Control:** Qwen 3 introduces "thinking mode fusion" for fine-grained control over CoT length and early stopping, enabling a single model to adapt its reasoning process based on budget.

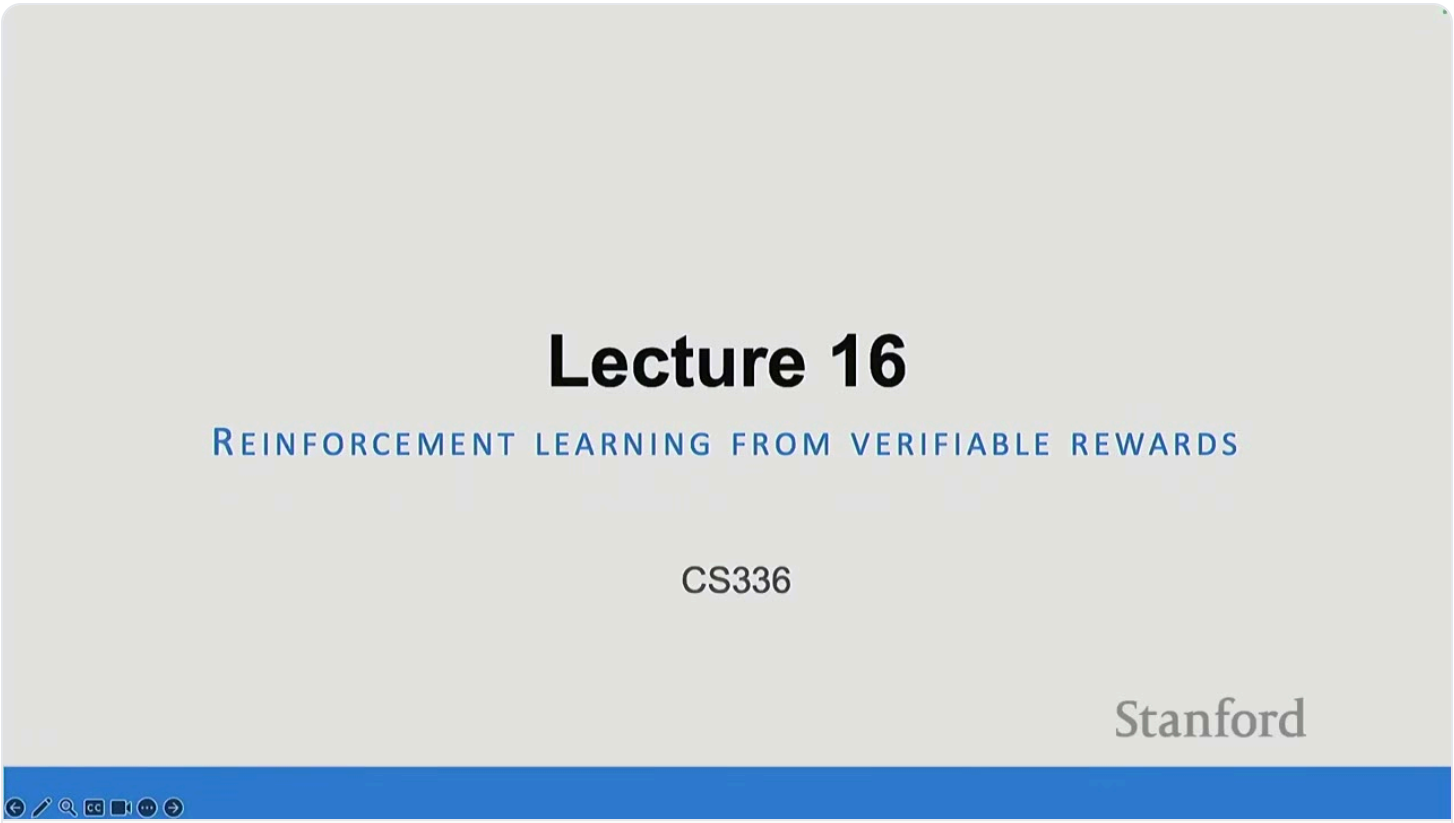
Key Concepts

- DPO (Direct Preference Optimization)
- PPO (Proximal Policy Optimization)
- TRPO (Trust Region Policy Optimization)
- GRPO (Group Relative Policy Optimization)
- RLHF (Reinforcement Learning from Human Feedback)
- RLVR (Reinforcement Learning from Verifiable Rewards)
- Overoptimization / Overfitting
- Mode Collapse / Calibration
- Policy Gradients
- Advantage Estimation
- Value Function
- Reward Shaping
- Length Normalization
- SFT (Supervised Fine-Tuning)
- CoT (Chain-of-Thought)
- Distillation
- Thinking Mode Fusion
- Inference Efficiency

[00:00] Introduction: RL from Verifiable Rewards

The lecture begins by finishing the remaining content from the previous session on RLHF and then transitions to Reinforcement Learning from Verifiable Rewards (RLVR), a new and exciting development in language model training.

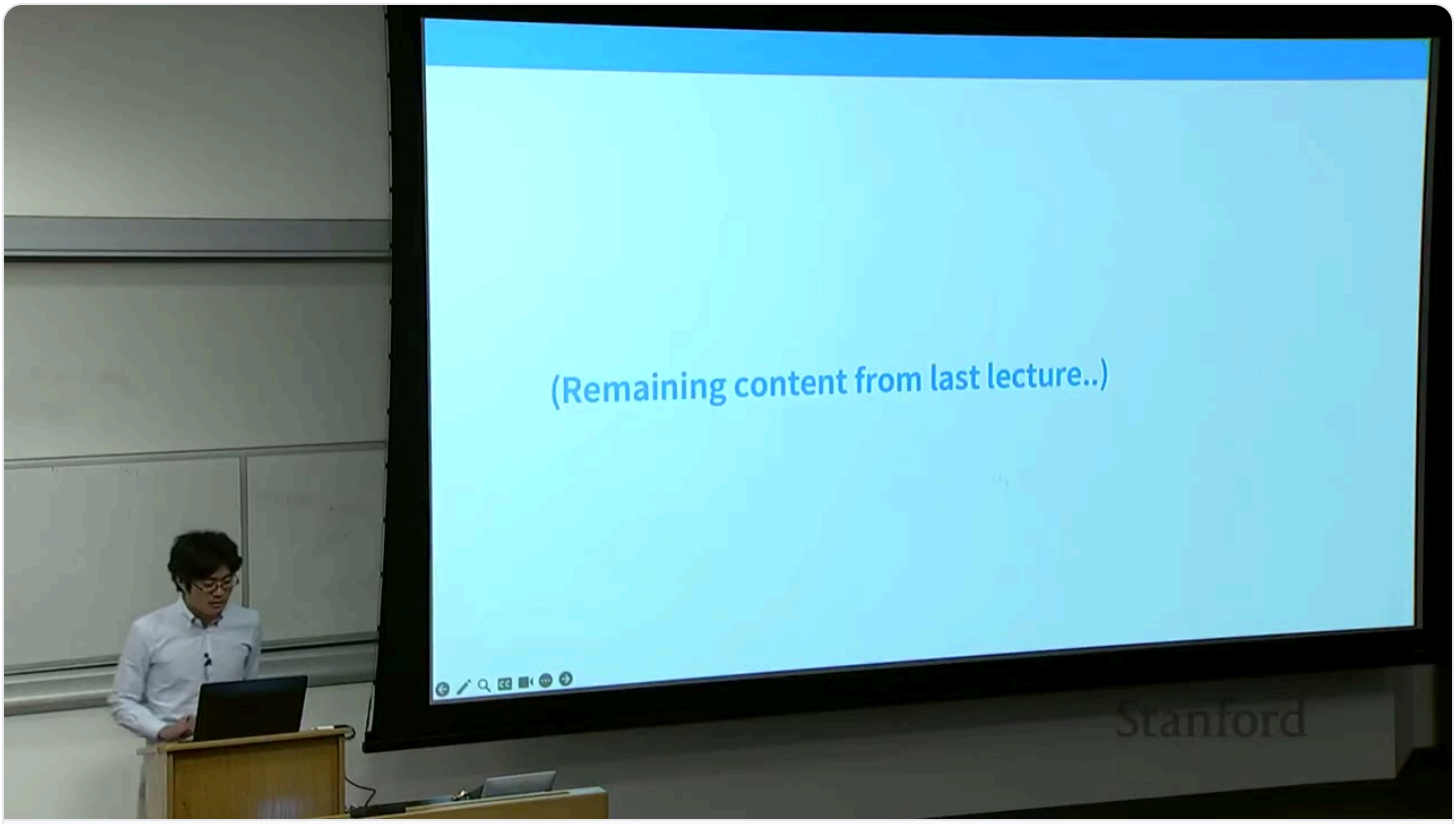
[00:05] RLHF Recap and Challenges



0:13 Lecture 16: Reinforcement Learning from Verifiable Rewards, CS336, Stanford

The lecturer provides a quick recap of RLHF and DPO, emphasizing the core idea of maximizing an underlying reward based on pairwise preference data.

[00:40] DPO - RLHF without tears?



0:40 A speaker at a podium, with a slide titled "Remaining content from last lecture.."

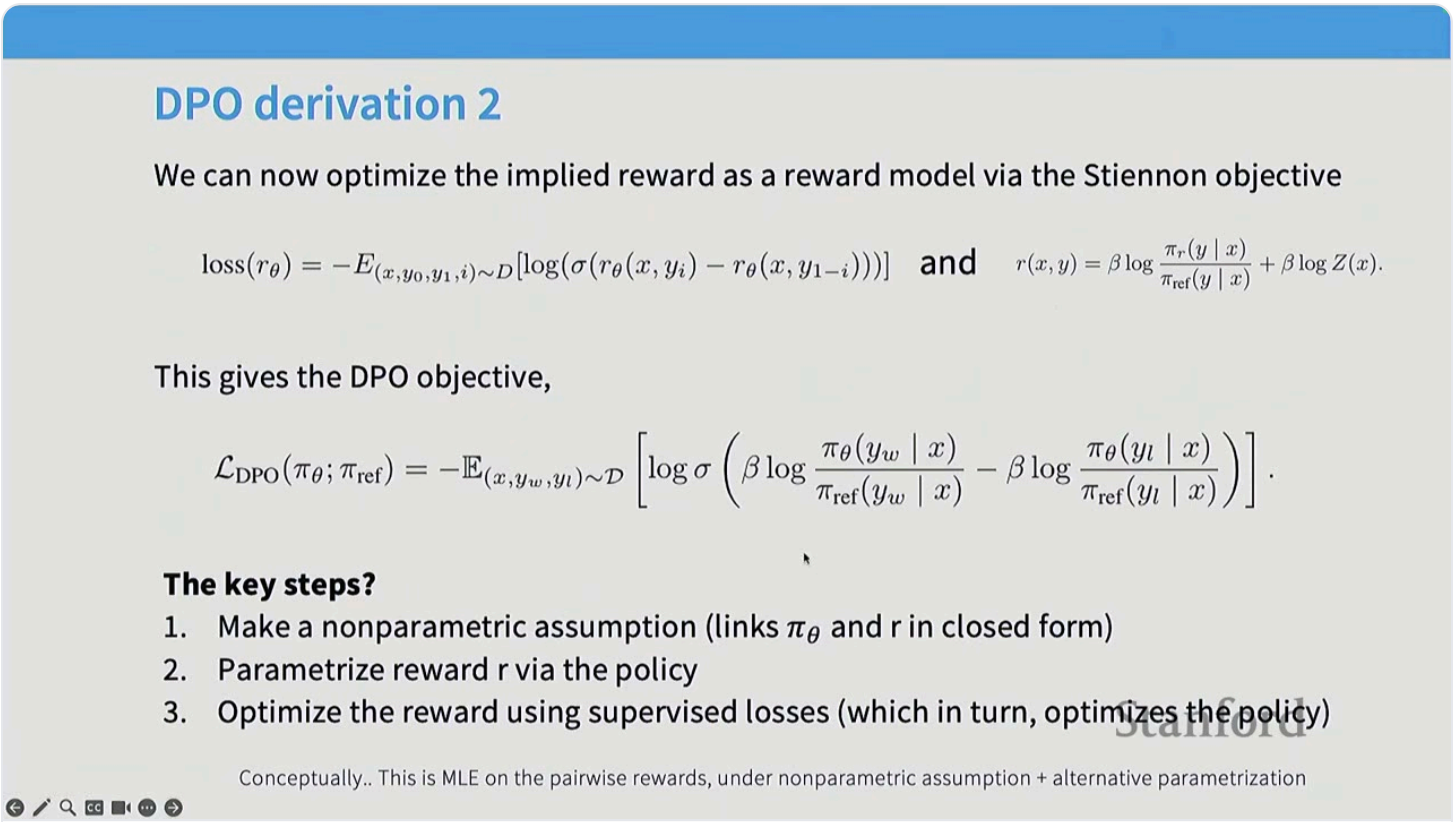
DPO simplifies PPO by getting rid of the reward model and on-policy rollouts. Instead, it takes gradient steps on log-loss of good stuff and negative gradient steps on bad stuff (appropriately weighted).

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = \mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}[-\log\sigma(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)})]$$

The key steps for DPO are: 1. Make a nonparametric assumption (policy class is the set of all functions). 2. Parametrize reward r via the policy. 3. Optimize the reward via supervised losses (which in turn, optimizes the policy).

This makes DPO essentially supervised learning on a reparameterized objective.

[02:03] DPO Updates and Components



1:47 A slide showing the DPO derivation, including loss equations and key steps.

DPO updates can be written in a nice form where the gradient steps are scaled by β (regularization). Updates are weighted higher when reward estimates are incorrect. The likelihood of good examples is increased, and the likelihood of bad examples is decreased.

$$\nabla_\theta \mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}[\sigma(\beta(\hat{r}_\theta(x,y_l) - \hat{r}_\theta(x,y_w))) \cdot \beta\,(\nabla_\theta \log \pi_\theta(y_w|x) - \nabla_\theta \lo$$

This means "positive gradient on good, negative gradient on bad."

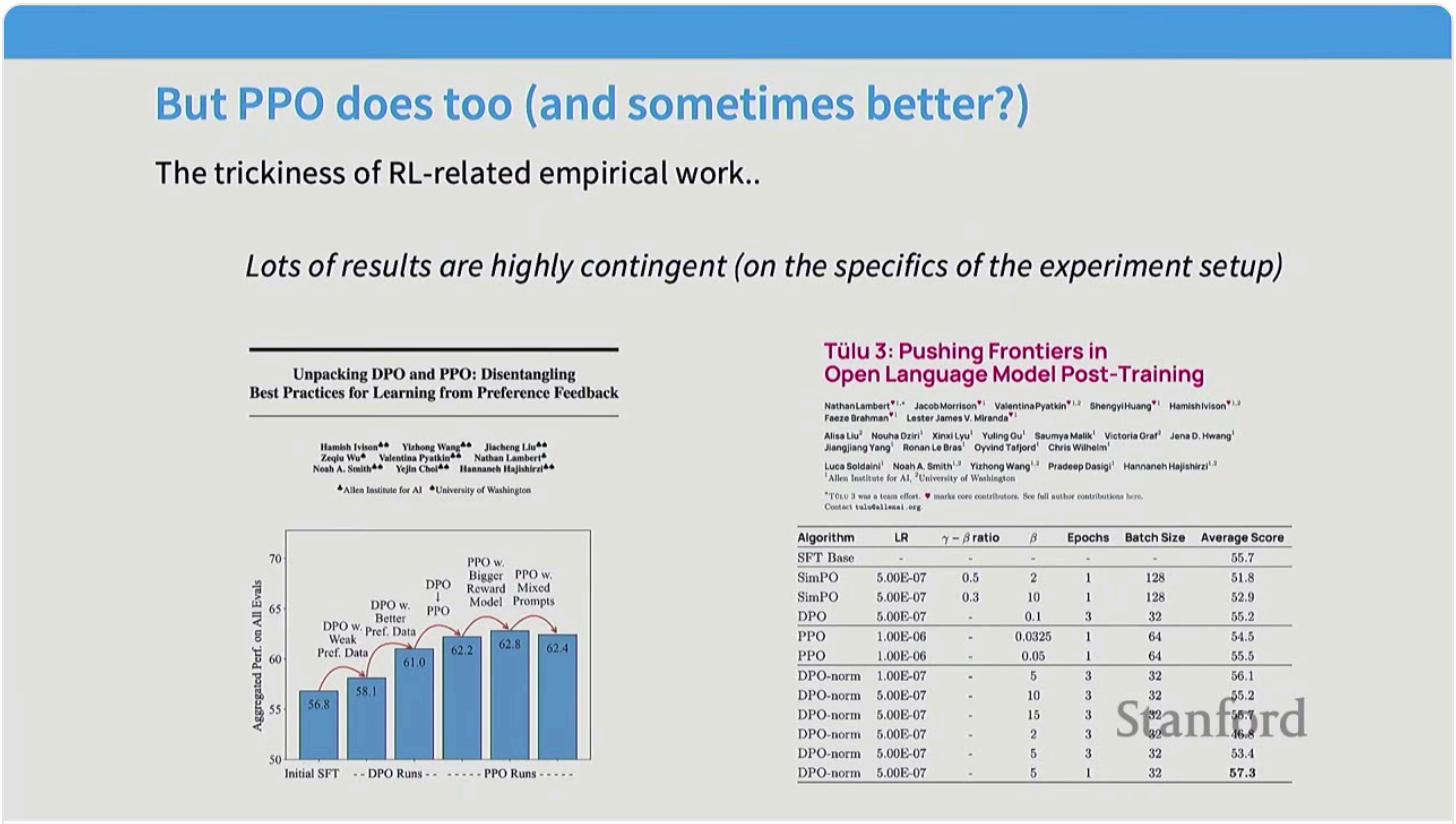
[03:00] DPO Works Pretty Well

For a while, DPO was the dominant approach for post-training in open-source models due to its ease of implementation compared to PPO.

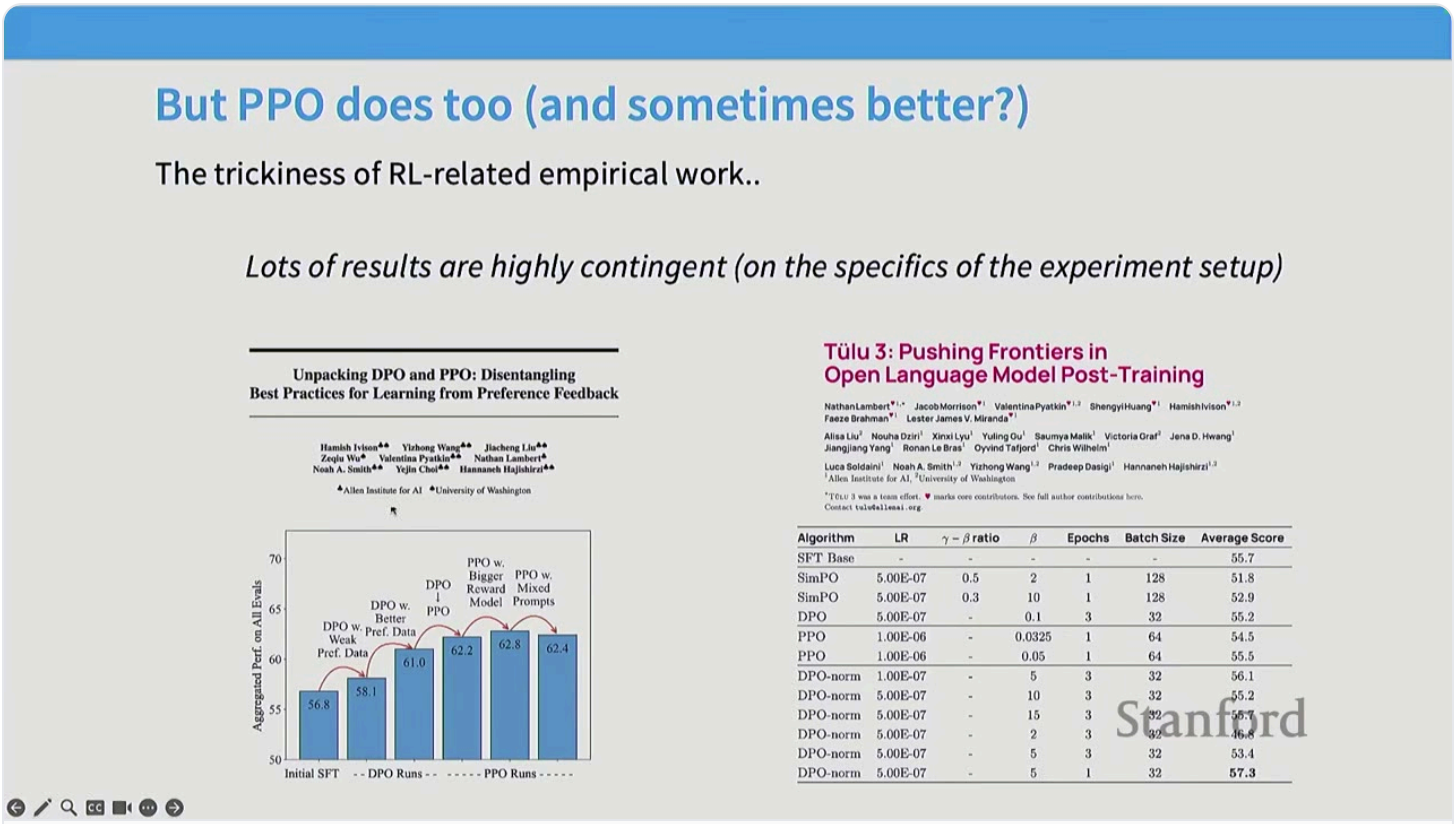
[03:19] Variants

A flood of DPO variants emerged, often denoted as "*PO". *Two notable variants are: - SimPO (Simple Preference Optimization): Normalizes the update size by the length of responses and removes the reference policy. While losing the mathematical argument of DPO, it aligns with the "upweight good, downweight bad" intuition. - Length Normalized DPO:* Normalizes by length without removing the reference policy.

[04:35] PPO Does Too (and Sometimes Better?)



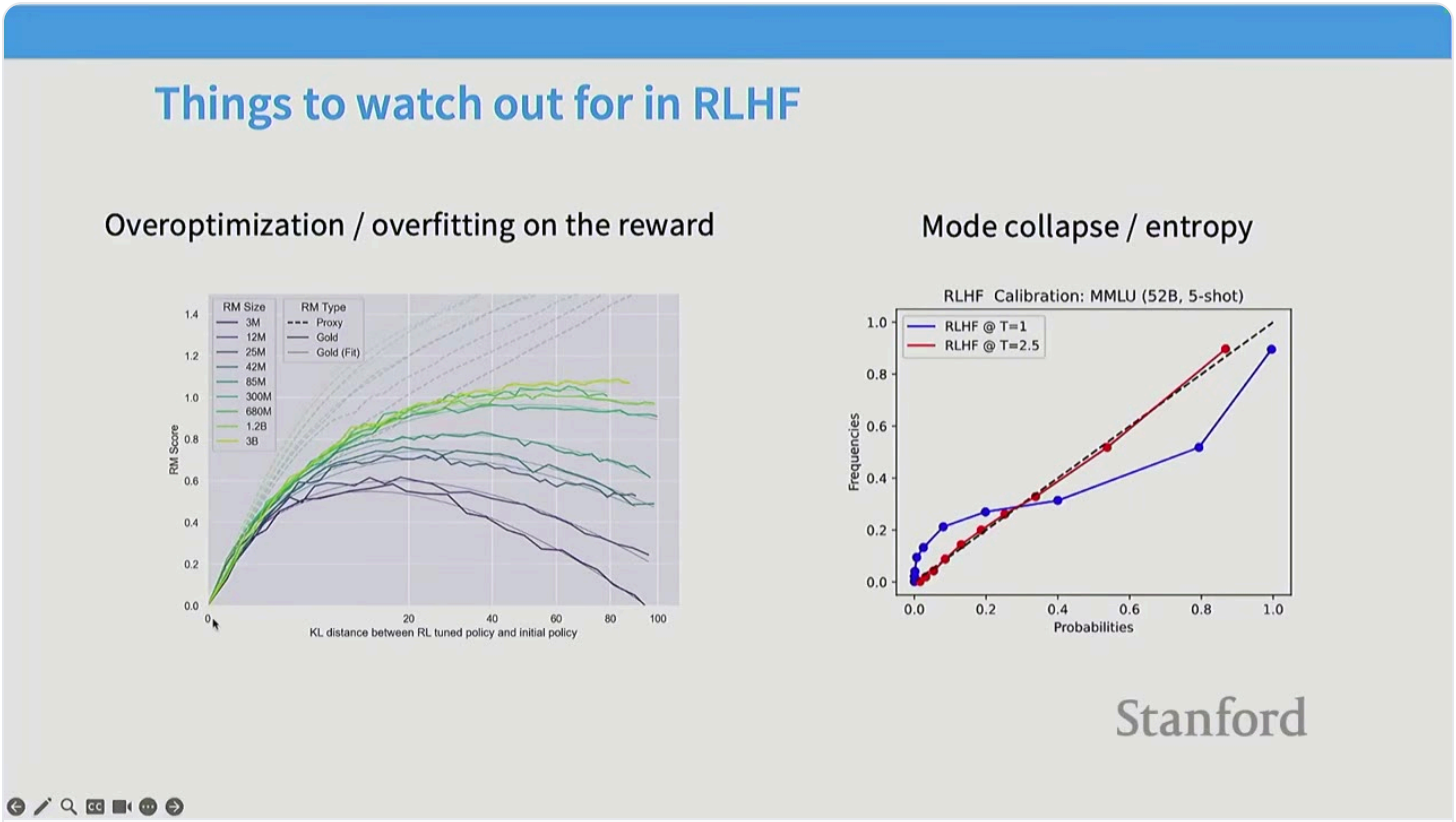
5:01 A slide discussing PPO, showing two research paper excerpts and a table of results.



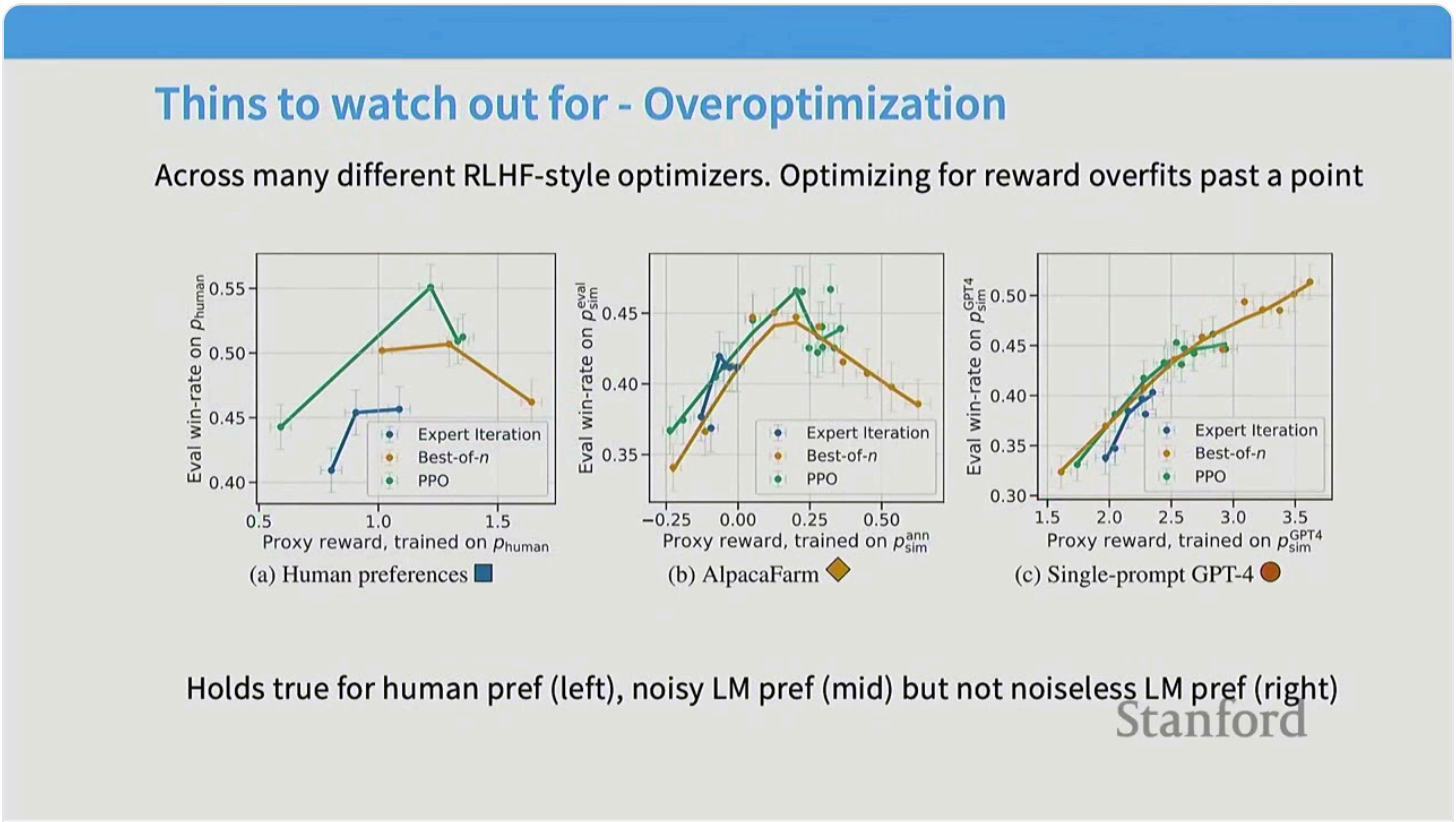
5:20 A slide discussing PPO, showing two research paper excerpts and a table of results.

The empirical findings in RL are highly contingent on the specific experimental setup. Different environments, base models, and post-training preferences can lead to different conclusions. For example, some AI2 work initially found PPO to be better than DPO, but later found that a better SFT method could negate these gains, with DPO with normalization performing best. This highlights the need for caution when generalizing results from a single paper.

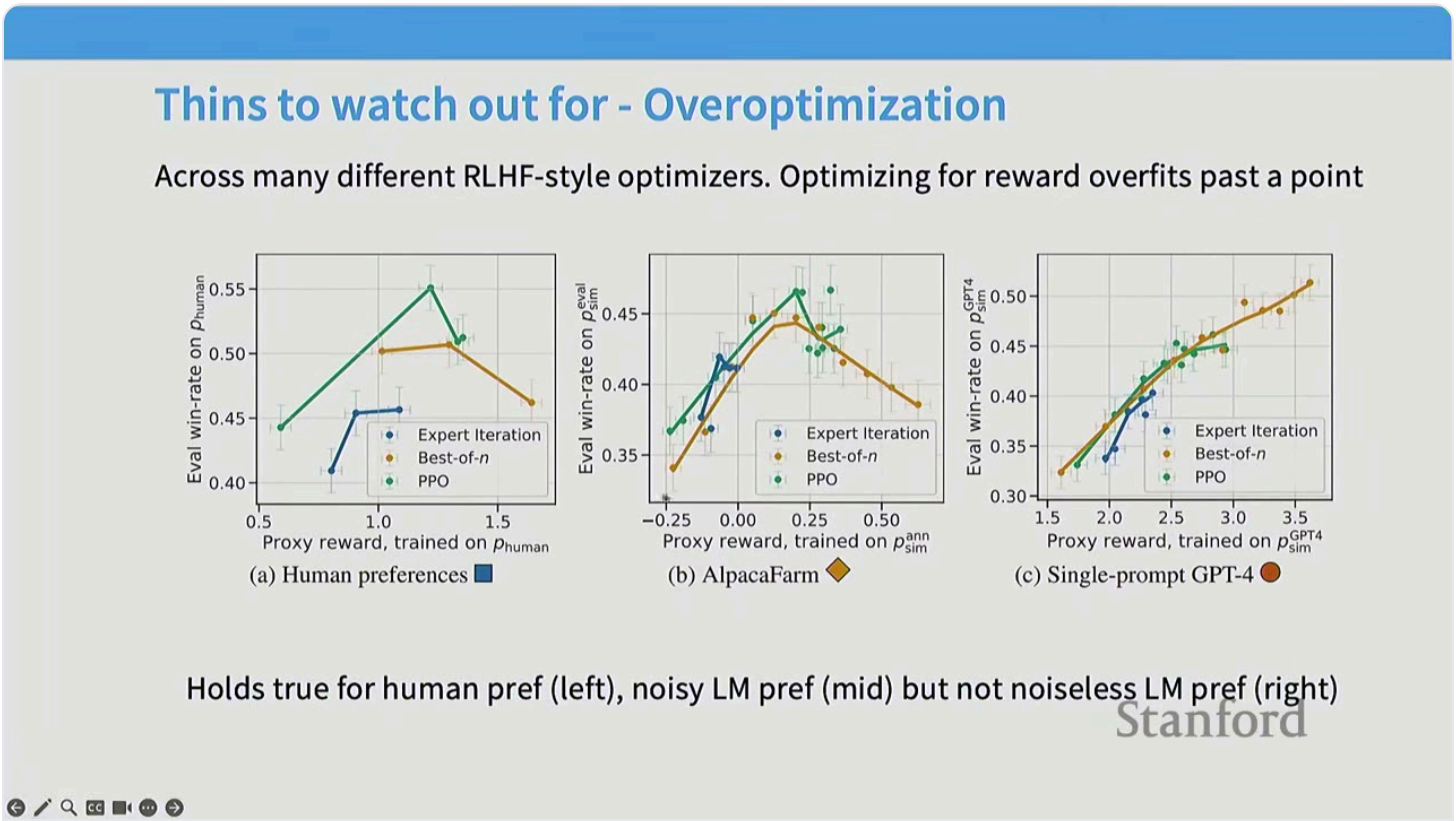
[06:01] Things to Watch Out For in RLHF



6:35 A slide on RLHF, showing graphs of overoptimization/overfitting and mode collapse/entropy.



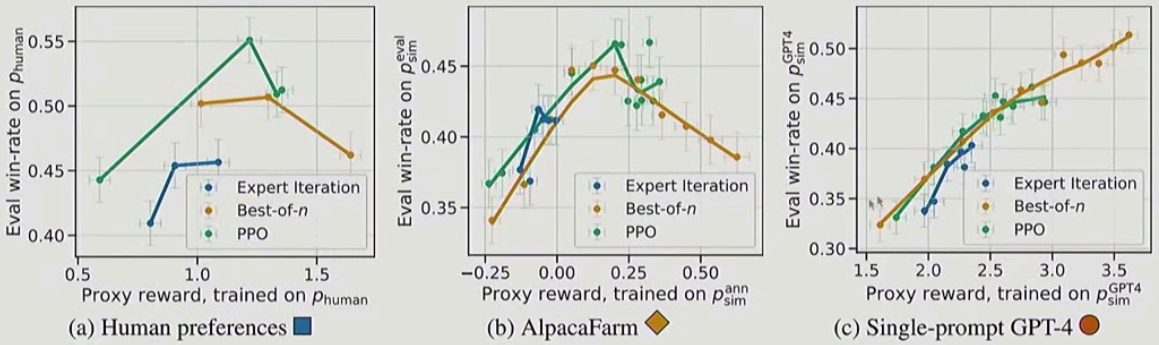
7:10 A slide on overoptimization in RLHF, showing three graphs comparing different optimizers.



7:25 A slide on overoptimization in RLHF, showing three graphs comparing different optimizers.

Thins to watch out for - Overoptimization

Across many different RLHF-style optimizers. Optimizing for reward overfits past a point



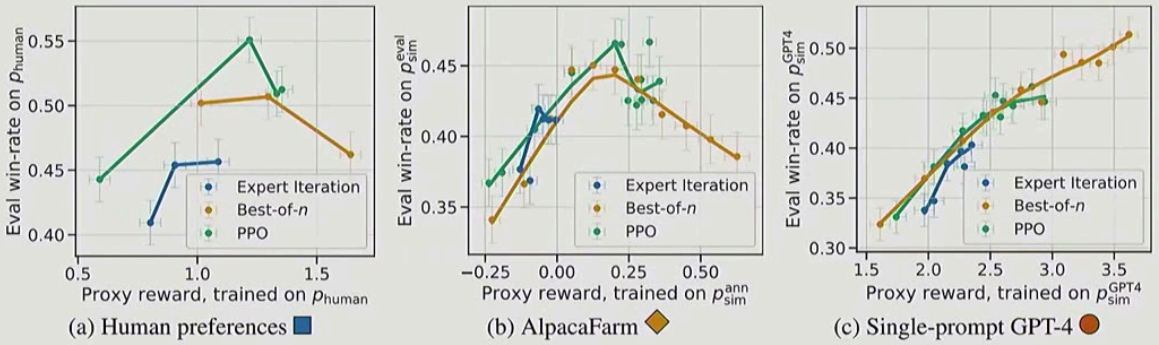
Holds true for human pref (left), noisy LM pref (mid) but not noiseless LM pref (right)

Stanford

7:35 A slide on overoptimization in RLHF, showing three graphs comparing different optimizers.

Thins to watch out for - Overoptimization

Across many different RLHF-style optimizers. Optimizing for reward overfits past a point



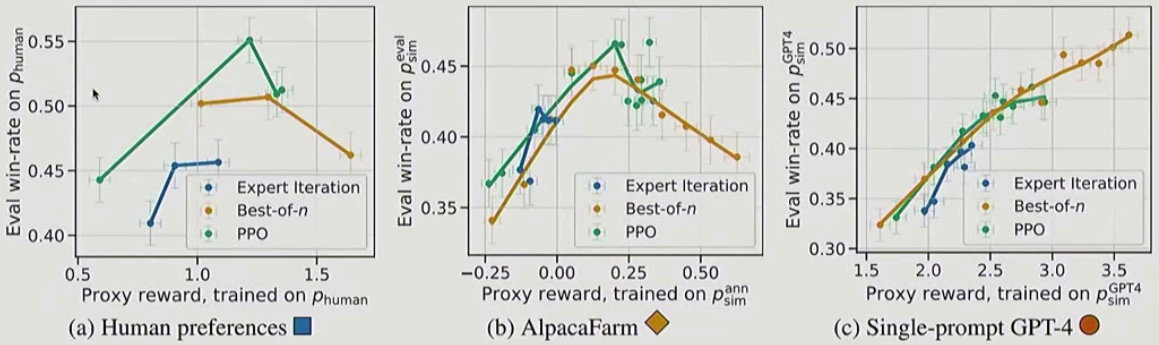
Holds true for human pref (left), noisy LM pref (mid) but not noiseless LM pref (right)

Stanford

7:45 A slide on overoptimization in RLHF, showing three graphs comparing different optimizers.

Thins to watch out for - Overoptimization

Across many different RLHF-style optimizers. Optimizing for reward overfits past a point



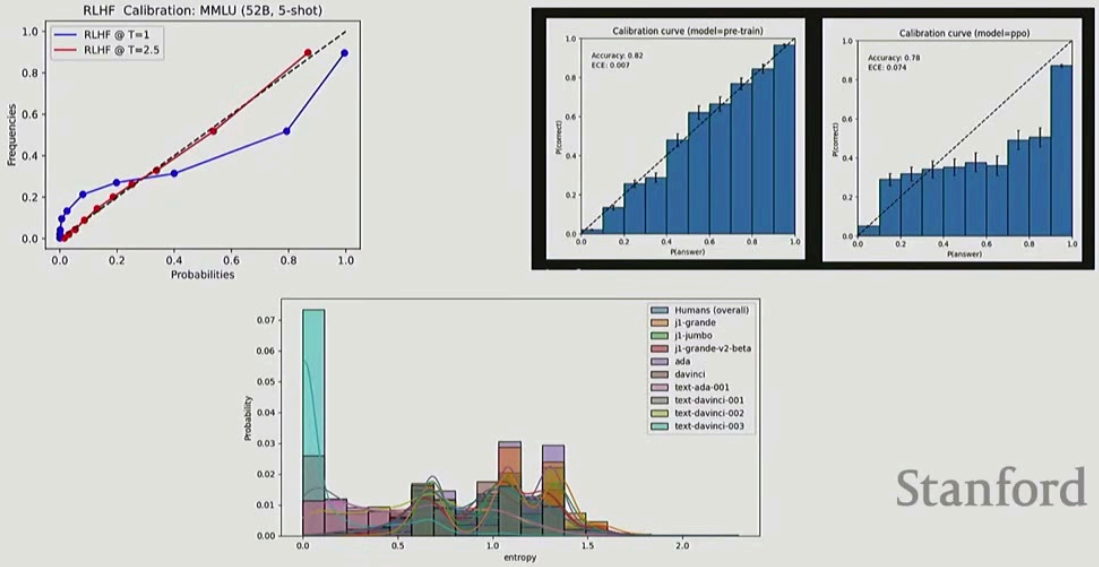
Holds true for human pref (left), noisy LM pref (mid) but not noiseless LM pref (right)

Stanford

7:55 A slide on overoptimization in RLHF, showing three graphs comparing different optimizers.

Things to watch out for - mode collapse

RLHF makes models no longer 'probabilistic models' – no calibration by default



Stanford

8:05 A slide on mode collapse in RLHF, showing calibration curves and an entropy histogram.

1. **Overoptimization / Overfitting on the Reward:** As RL training progresses, the model's performance on the learned reward model initially improves, but eventually diverges from true human preferences. This is essentially overfitting to the reward model, which is often noisy and complex. This phenomenon is observed with human preferences and noisy AI feedback, but not with clean, noiseless AI feedback.
2. **Mode Collapse / Entropy:** RLHF makes models less probabilistic and calibrated by default. Unlike pre-training or supervised fine-tuning, RLHF is not about matching a distribution. This often leads to models exhibiting overconfident behavior, as calibration is not part of the reward signal. It's crucial not to treat RLHF models as calibrated probabilistic models.

[09:05] The Class Thus Far

The discussion so far has focused on pre-training and RLHF, leading to models like GPT-3.5. The next part of the lecture will transition to RLVR, which is enabling the development of reasoning models like O1.

[11:50] Q&A on RLHF

Q: What do the x and y axes represent in the overoptimization graph? **A:** The x-axis represents the success of the RL algorithm in optimizing the proxy reward classifier (e.g., how well PPO optimized the reward model). The y-axis represents true win-rates, measured by fresh human votes or AI feedback. The graph shows that succeeding in RL doesn't guarantee success in the actual task, as models can overfit to human preferences.

Q: Why are the ratios of policies used in DPO, not just the difference of log probabilities? **A:** The ratios are a compact and intuitive way to represent the underlying mathematical argument of DPO. While taking the difference of log probabilities is computationally equivalent, the ratio form is often used for theoretical clarity. When moving to SimPO, the mathematical argument changes, as it's no longer strictly about ratios of policies.

Q: How is the ground truth of the reward measured in the overoptimization graph? **A:** The x-axis measures the performance on the *fitted* reward model (train set), while the y-axis measures performance on *fresh samples* from the true reward oracle (test set). This is analogous to a train-test gap in machine learning, where the model overfits to the training data.

[12:25] The Goal: Expand the Scope and Power of RL

The core idea behind RLVR is to leverage the power of RL in domains where objective, verifiable rewards are available, similar to how AlphaGo or AlphaFold excelled. This avoids the limitations of human feedback (noisiness, scalability, overoptimization) and allows for more efficient and robust training.

[13:30] Lecture Today

The lecture will proceed in two parts: 1. **Core Algorithms:** A detailed discussion of PPO, its transformation into GRPO, and various GRPO variants. 2. **Case Studies:** An examination of three prominent open models (DeepSeek R1, Kimi 1.5, and Qwen 3) that utilize RLVR, providing practical examples of the concepts discussed.

[14:30] Recap: PPO in Theory

PPO (Proximal Policy Optimization) is a policy gradient method for optimizing rewards in actual RL tasks. The lecturer explains PPO by starting from the simplest policy gradient and progressively adding complexities.

[14:50] Attempt 1: Policy Gradients (Variances are Too High)

The simplest approach is to optimize the expected reward under the policy π_θ using gradient ascent. The policy gradient theorem states that the gradient of the expected reward is:

$$\nabla_\theta \mathbb{E}_{P_\theta}[R(Z)] = \mathbb{E}_{P_\theta}[R(Z)\nabla_\theta \log P_\theta(Z)]$$

This means taking gradient steps proportional to the reward, increasing the probability of actions that lead to high rewards. However, the variance of these gradients can be very high.

[15:28] Attempt 2: TRPO (Trust Region Policy Optimization)

To address high variance and improve sample efficiency, TRPO linearizes the problem around the current policy. Instead of sampling from the current policy π_θ for every gradient step, TRPO samples from an older policy π_{old} (or π_k) and uses importance sampling to correct the gradients.

To prevent the new policy from diverging too much from the old one, a KL divergence constraint is added.

$$\max_{\theta} \mathbb{E}_{(s,a)\sim\pi_{\text{old}}} \left[\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)} \hat{A}_{\text{old}}(s,a) \right] \quad \text{subject to} \quad \mathbb{E}_{s\sim\pi_{\text{old}}}[D_{\text{KL}}(\pi_\theta(\cdot|s)||\pi_{\text{old}}(\cdot|s))] \leq \delta$$

Here, $\hat{A}_{\text{old}}(s,a)$ is an advantage estimate, a lower-variance version of the reward.

[17:00] Attempt 3: PPO (Clip the Ratios at Some Epsilon)

PPO simplifies TRPO by replacing the explicit KL divergence constraint with a clipping mechanism. The objective function for PPO is:

$$L(s,a,\theta,\theta_{\text{old}}) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \hat{A}(s,a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}, 1-\epsilon, 1+\epsilon \right) \hat{A}(s,a) \right)$$

This clips the ratio of new to old policy probabilities, effectively limiting how much the policy can change in a single step. This prevents large, destabilizing updates and encourages the policy to stay close to the previous one without an explicit KL constraint.

[17:30] PPO in Practice

PPO is a very successful RL algorithm, used in various environments from simple toy tasks to complex games like Dota 2 (OpenAI Five). However, implementing PPO in practice is notoriously difficult. The blog post "The 37 Implementation Details of Proximal Policy Optimization" highlights the complexity and the sensitivity of PPO to implementation choices. Different implementations can lead to vastly different performance, and even slight errors can lead to incorrect policy gradients that surprisingly sometimes perform better. This underscores the empirical nature of RL and the need to carefully examine implementation details.

[19:35] PPO - Idealization (?) for Language Models

Applying PPO to language models involves a complex setup: - **SFT Model:** A supervised fine-tuned model. - **Reward Model:** Learned from human feedback. - **Value Model:** A neural network that estimates expected rewards, used for variance reduction in advantage estimation. - **GAE (Generalized Advantage Estimation):** A sophisticated method for estimating advantages. - **Policy LM:** The language model being trained. - **Replay Buffer:** Stores past experiences for off-policy learning.

This intricate machinery is necessary for PPO to work in the context of language models.

[20:13] PPO in Practice - An Implementation

The lecturer refers to an implementation of PPO from AlpacaFarm, which has been somewhat tested. The code snippet shows the outer loop of PPO, which involves: 1. Getting rollouts (trajectories) from the current policy. 2. Computing losses based on these rollouts. 3. Performing backpropagation and gradient steps (with clipping). 4. Updating the optimizer.

This structure is common in many RL algorithms.

[21:17] PPO in Practice - Loss Computation

The loss computation in PPO involves two main parts: 1. **Value Function Loss:** Measures how close the predicted value from the value function is to the actual returns (rewards). This helps the value function provide accurate variance reduction. 2. **Policy Loss:** Computes the PPO-Clip objective, comparing the current policy's log probabilities to the reference policy's log probabilities, weighted by advantages, and clipped within a range (e.g., ± 0.2).

[22:15] PPO in Practice - Rollouts

Rollouts involve generating sequences from the current policy. This is where the complexity arises: - **Tokenization:** Different models (policy, reward, value) might have different tokenizers, requiring retokenization. - **Evaluation:** Feeding rollouts into the models to get log probabilities, rewards, and values.

[22:41] PPO in Practice - Reward Shaping

Reward shaping is often used to make the reward signal more amenable to RL algorithms. In PPO, this involves: - **Per-token KL penalty:** A regularization term computed for each token, encouraging the policy not to deviate too much from the reference policy. - **Last-token full reward:** The main task reward (e.g., success/failure) is applied only at the last token of the sequence.

The KL term is often clamped to 0 to avoid numerical instability when the log-ratio becomes too negative, which is an approximation rather than a true KL divergence.

[25:52] PPO in Practice - Generalized Advantage Estimate

Instead of directly using rewards, PPO uses generalized advantage estimation (GAE) to reduce variance in policy gradients. GAE is a discounted sum of advantages, where the advantage is the difference between the reward and a baseline (often the value function). The hyperparameters λ and γ control the trade-off between bias and variance. A common simplification is to set $\gamma = \lambda = 1$, which reduces GAE to a simple baseline policy gradient.

[26:59] What to Expect in PPO?

In a contextual bandit setting like language model generation, PPO typically shows: - Increasing overall rewards. - Increasing reward model scores. - Negative KL rewards (as the policy deviates from the reference).

[27:33] Why Do We Need Yet Another RL Algorithm?

The complexities of PPO (complicated implementation, memory-hungry value model) and the limitations of DPO (data not inherently pairwise, offline nature) motivate the search for simpler and more efficient RL algorithms for language models.

[28:58] New Kid on the Block: GRPO

GRPO (Group Relative Policy Optimization) aims to simplify PPO while retaining its benefits. - **Start with PPO:** GRPO builds upon the core ideas of PPO. - **Remove Value Function / Advantage Computation:** GRPO eliminates the need for a separate value function and complex advantage estimation. - **Calculate Advantage as "z-score within group":** The advantage for a given response is calculated as its z-score (reward minus mean, divided by standard deviation) relative to other responses within the same group (e.g., responses to the same prompt).

A "group" typically refers to all responses generated for a single input prompt. This group-wise z-score normalization acts as a baseline, accounting for varying problem difficulties.

The GRPO objective is:

$$\max_{\theta} \mathbb{E}_{G\sim\mathcal{D}}\left[\frac{1}{|G|}\sum_{i=1}^{|G|}\left(\frac{\pi_\theta(a_i|s_i)}{\pi_{\text{old}}(a_i|s_i)}\hat{A}_i\right) - \beta D_{\text{KL}}(\pi_\theta||\pi_{\text{old}})\right]$$

where $\hat{A}_i = \frac{r_i - \text{mean}(r_1,\ldots,r_{|G|})}{\text{std}(r_1,\ldots,r_{|G|})}$.

In the online case (rollout + immediate update), GRPO simplifies to a policy gradient with group-normalized rewards.

[30:00] Q&A on GRPO

Q: When you say "group," does it mean multiple answers for one question, or multiple questions? **A:** For each question (Q), GRPO samples a group of outputs (G). So, for a single question, you generate multiple responses, and these responses form a group. The baselining (mean and standard deviation) is applied *within* this group. Across different questions, there's no explicit baselining, only the policy updates.

Q: What about the DKL term in GRPO? **A:** The DKL term in GRPO is a slightly non-standard estimate of KL divergence. Instead of just the average log-ratio, it includes additional terms related to the ratio of reference to current policy probabilities, which act as a control variate to reduce variance.

[30:50] GRPO is Simple (Thanks to Lack of Value Function)

GRPO implementations are typically much simpler than PPO because they don't require a value function. The core steps involve: - Compute reward for each rollout. - Mean/variance normalization per group. - Compute KL term. - Gradient updates on the loss.

[31:10] GRPO Doesn't Use a "Valid" Baseline

Recent analyses of GRPO have identified two mathematical "issues": 1. **Division by Standard Deviation:** Dividing by the standard deviation in the advantage calculation is not a valid baseline that preserves unbiasedness. This breaks the theoretical guarantee of the policy gradient theorem. 2. **Length Normalization:** The GRPO objective implicitly normalizes rewards by the length of the output. This creates an incentive for the model to produce very long responses when incorrect (to minimize negative reward per token) and very short responses when correct (to maximize positive reward per token). This leads to models "BS-ing" as aggressively as possible.

These issues can lead to unintended behaviors, such as models generating overly long or short responses to manipulate the reward signal.

[32:10] Length Biases of GRPO

The standard deviation term in GRPO upweights problems that are either too easy or too hard (where variance is low), potentially slowing convergence. The length normalization term incentivizes longer responses for negative advantages and shorter responses for positive advantages.

If these issues are fixed, models show improved reward curves and output lengths that stabilize rather than growing unboundedly. This suggests that some of the very long CoTs observed in GRPO might be an artifact of these implementation choices rather than an inherent property of reasoning.

[33:00] Case Studies of RLVR

The lecture will now examine three representative open models that utilize RLVR: DeepSeek R1, Kimi 1.5, and Qwen 3.

[33:10] DeepSeek R1

DeepSeek R1 is a paper that launched a significant social phenomenon, demonstrating O1-level reasoning performance with a surprisingly simple RL recipe. - **Performance:** Achieves performance exceeding OpenAI O1. - **RL Recipe:** The RL recipe is remarkably simple, lacking complex search or process reward models. This challenged the prevailing belief that such complexities were necessary for reasoning models. - **Insights:** Provides insights into the interaction between SFT and RL.

[34:00] Algorithm - GRPO

DeepSeek R1 builds on the results of GRPO from DeepSeekMath. However, unlike DeepSeekMath, R1 does not use process supervision (intermediate step rewards) in its GRPO implementation.

[34:20] Controlled Setting - R1 Zero

R1 Zero is a controlled setting where a pre-trained model (DeepSeek V3) is directly fine-tuned with GRPO on math-ish tasks. - **Rewards:** Uses accuracy rewards (binary correct/incorrect) and format rewards (to ensure CoTs are within thinking tags). - **Data:** Proprietary math-ish tasks (not public). - **Results:** Achieves performance close to OpenAI O1 on various benchmarks (AIME, MATH, GPQA, LiveCodeBench, CodeForces) by simply applying GRPO on top of the base model.

[35:00] Interesting Phenomena

R1 Zero exhibits two interesting phenomena: 1. **Longer CoTs during training:** The length of the CoT increases predictably during RL training. The authors interpret this as the model learning to solve harder problems by thinking harder. 2. **"Aha" moment:** The model learns to backtrack and correct its reasoning, which the authors refer to as an "aha" moment.

However, follow-up analyses (e.g., in the Dr. GRPO paper) suggest these phenomena might be overstated: - **Length due to biased objective:** The increasing CoT length might be an artifact of the biased GRPO objective rather than an inherent learning process. - **Base model already has 'aha':** The "aha" moments might already be present in the base model (DeepSeek V3) and not necessarily an emergent property of RL.

[36:00] Pushing Performance Further - R1

To achieve even better performance, R1 extends R1 Zero by incorporating additional techniques: - **SFT initialization:** Fine-tuning the base model with long CoTs from an undisclosed source to prevent unstable cold start of RL training. This also provides interpretability benefits. - **Language consistency reward for CoT:** An additional reward to prevent the CoT from language mixing (e.g., switching between English and Chinese). - **Non-verifiable rewards (in stage 2):** Using RLHF for general-purpose alignment.

[37:00] SFT Initialization

SFT initialization involves fine-tuning the base model with long CoTs. This helps to prevent unstable cold start of RL training and provides interpretability benefits by keeping the model's CoTs closer to human-readable formats. However, the source and quality of this CoT data are not fully disclosed.

[37:30] SFT for Reasoning / Math

Even a small number of SFT examples (e.g., 1000) on long CoTs can significantly boost math reasoning accuracy. This suggests that base models already possess significant reasoning capabilities that can be unlocked with targeted SFT, rather than requiring extensive RL.

[38:00] RL Step

The RL step in R1 is similar to R1 Zero, but with an additional language consistency loss. This loss is added to mitigate language mixing in CoTs, which can naturally occur during aggressive RL training. This is consistent with observations of models like Grok 3 switching languages in their CoTs.

[38:20] SFT/RLHF

The final stage involves layering on the usual post-training pipeline: - **SFT step:** Fine-tuning on a mix of reasoning data (non-verifiable tasks, judged by V3) and non-reasoning data (SFT dataset). - **RLHF step:** Reusing the R1 Zero-style reasoning RLHF pipeline, still using GRPO for RLHF.

[39:00] How Well Does R1 Work?

R1 achieves performance comparable to OpenAI O1 across various benchmarks (MMLU, GPQA, LiveCodeBench, CodeForces). It matches or beats O1 on most English tasks and performs very well on math and coding tasks, demonstrating that a simple recipe can achieve state-of-the-art reasoning capabilities.

[39:20] Distillation - Can We Get Non-Reasoning Models to Reason?

R1 also shows that its generated CoT traces can be used to distill reasoning capabilities into smaller, non-reasoning models. By fine-tuning models like Qwen with R1's CoT traces, significant performance boosts can be achieved on math benchmarks, suggesting that reasoning can be effectively transferred.

[40:00] Other, Relevant Observations

R1 also explored other approaches that did not pan out: - **PRMs (Process Reward Models):** These models provide intermediate feedback on each step of a CoT. While conceptually powerful, R1 found that PRMs did not significantly improve performance compared to outcome-based rewards. - **MCTS (Monte Carlo Tree Search):** Inspired by AlphaGo, MCTS involves searching through possible reasoning steps. R1 found that MCTS did not yield better results than simple RL, suggesting that the search space for reasoning might be ill-defined or too complex for current methods.

These "negative" results highlight the effectiveness and simplicity of the GRPO-based RLVR approach.

[40:50] Kimi 1.5

Kimi 1.5 is another contemporaneous model that achieves similar results to R1 using RLVR. - **Released at the same time as R1:** Allows for a comparison of parallel developments. - **Beats O1 using RL:** Demonstrates strong performance.

[41:00] Long CoT Reasoning Strategy

Kimi 1.5 employs a similar strategy to R1: - **Dataset Construction:** Filters for difficulty using a "best-of-N" strategy. - **SFT:** Fine-tuning with long CoTs. - **RL:** Uses its own policy gradient loss.

[41:30] Data Curation + SFT

Kimi 1.5's data curation involves: - **Standard coverage across math-style settings:** Balances questions across different domains (math, STEM, competitions). - **Exclude multiple choice / true false (false positives):** Focuses on verifiable answers that are not easily guessable. - **Select only examples that fail best-of-8:** Filters out problems that are too easy for the model.

SFT is described as "prompt engineering," suggesting a focus on crafting effective prompts to elicit desired model behavior.

[42:30] Kimi RL Algorithm + Regularization

The Kimi RL algorithm is derived from a DPO-type derivation, but it's closer to a baselined policy gradient with regularization. - **Objective:** Maximizes reward while minimizing KL divergence from the base policy. - **RL Algorithm:** Uses a baselined policy gradient with a squared loss regularization term instead of clipping. This is analogous to GRPO but without the standard deviation normalization.

[43:15] Length Control in Kimi

Kimi 1.5 explicitly addresses the issue of CoT length, recognizing its impact on inference cost. - **Length Reward:** For each batch, a length reward is calculated based on the minimum and maximum CoT lengths. - **Incentives:** The reward incentivizes correct answers to be as short as possible and incorrect answers to be around the average length. This aims to compress CoTs while maintaining performance. - **Training Schedule:** This length reward is only enabled later in training to avoid stalling the RL process in early stages.

[44:00] Additional Details

- **Curriculum:** Kimi 1.5 uses a curriculum-based approach, assigning difficulty labels to the dataset and sampling problems proportional to (1 - success rate) to avoid repeating solved ones.
- **Rewards:** For code, they generate new test cases for ground truth solutions. For math, they use a CoT reward model (trained on 800k samples) for answer equivalence checks, which is highly accurate.

[44:30] RL Infra

Kimi 1.5 provides details on its RL infrastructure, highlighting challenges in scaling RL: - **On-policy rollouts:** Inference is slow. - **Switching from training to rollouts:** Requires frequent data transfer between different frameworks. - **Long CoTs:** Can lead to uneven batch sizes.

Kimi's hybrid deployment framework uses separate containers for different tasks (training, rollouts, inference) and manages data flow through shared memory and checkpointing. This allows for efficient GPU utilization and asynchronous operations.

[45:00] Scaling Results

Kimi 1.5 shows impressive scaling results, with accuracy improving across iterations and different benchmarks. The length of responses also shows a controlled growth, unlike the unbounded growth seen in vanilla GRPO.

[45:30] Final Case Study - Qwen 3

Qwen 3 is the most recent and modern of the RL for reasoning models. It builds upon previous works and introduces new techniques for controlling CoT generation.

[45:40] Overall Picture

Qwen 3's pipeline is similar to R1 and Kimi 1.5: - **Stage 1: Long-CoT Cold Start:** SFT with long CoTs. - **Stage 2: Reasoning RL:** RL training for reasoning. - **Stage 3: Thinking Mode Fusion:** A new technique to control CoT length. - **Stage 4: General RL:** RLHF for general alignment. - **Distillation:** Distilling flagship models into lightweight models.

[46:00] SFT + Reasoning RL

Qwen 3's data curation and RL training are similar to Kimi 1.5, with filtering for difficulty and manual filtering for CoT quality. Notably, their RL is performed on a very small number of examples (3995), suggesting high sample efficiency.

[46:20] Qwen 3 Specific New Stuff

Qwen 3 introduces "Thinking Mode Fusion" to control the length of the CoT. - **Mixed Training Data:** The model is fine-tuned with data that includes both "thinking mode" (with `<think>` tags) and "non-thinking mode" (with `<no_think>` tags). - **Inference Control:** During inference, the model can be prompted to either generate a CoT (by including `<think>` tags) or directly provide an answer (by including `<no_think>` tags). - **Early Stopping:** A special string can be inserted to terminate the thinking process early, allowing for precise control over the CoT length.

[47:00] Test Time Scaling

Thinking Mode Fusion allows for flexible test-time scaling. By setting a maximum thinking budget, Qwen 3 can achieve graceful performance degradations as the budget is reduced. This provides a control knob for balancing inference cost and performance.

[47:30] Composition of the Different Stages

Qwen 3 provides a detailed ablation study, showing the performance at each stage of its pipeline. - **Reasoning RL:** Improves performance on general tasks and instruction following. - **Thinking Mode Fusion:** Further enhances performance, especially for math and coding. - **General RL:** RLHF improves performance across the board.

A notable observation is a potential trade-off: General RL (RLHF) seems to hurt performance on math and coding tasks in thinking mode, while improving it in non-thinking mode. This suggests a tension between general instruction following and specialized reasoning.

[48:00] Recap for Today

- **Overoptimization is a problem:** RL in narrow domains (RLVR) is one solution.
- **GRPO is simple (but with some flaws):** It's an easy-to-implement algorithm that enables RLVR.

- **Lots of successful recipes in the wild:** DeepSeek R1, Kimi 1.5, and Qwen 3 demonstrate the effectiveness of RLVR.

Practical Takeaways

- **RLHF is powerful but tricky:** While RLHF can align models with human preferences, it's prone to overoptimization and can lead to uncalibrated models.
- **RLVR offers a scalable alternative:** By using verifiable, objective rewards, RLVR can overcome the limitations of human feedback and enable more robust and efficient training.
- **Simplicity can be powerful:** GRPO demonstrates that effective RL algorithms don't necessarily need complex components like value functions, making them easier to implement and scale.
- **Data curation is key:** The quality and diversity of the training data, especially for SFT, significantly impact the performance of RLVR models.
- **Control over inference is crucial:** Techniques like Thinking Mode Fusion in Qwen 3 allow for fine-grained control over CoT generation, balancing performance and inference cost.
- **Infrastructure matters:** Efficient RL training requires robust infrastructure to handle rollouts, model updates, and data transfer between workers.

Open Questions / Things to Remember

- **Generalizability of RLVR:** How well do models trained with RLVR on narrow, verifiable domains generalize to broader, more open-ended tasks?
- **Trade-offs between general purpose and specialized reasoning:** Is there an inherent trade-off between optimizing for general instruction following (via RLHF) and specialized reasoning (via RLVR)?
- **The "black magic" of RL hyperparameters:** How to optimally combine different reward signals (accuracy, formatting, consistency) remains an empirical challenge.
- **The role of CoT length:** While longer CoTs can improve reasoning, they also increase inference cost. Finding the optimal balance and controlling CoT generation is an active area of research.
- **The future of RL infrastructure:** As RL for LLMs becomes more prevalent, robust and efficient infrastructure will be critical for scaling research and deployment.

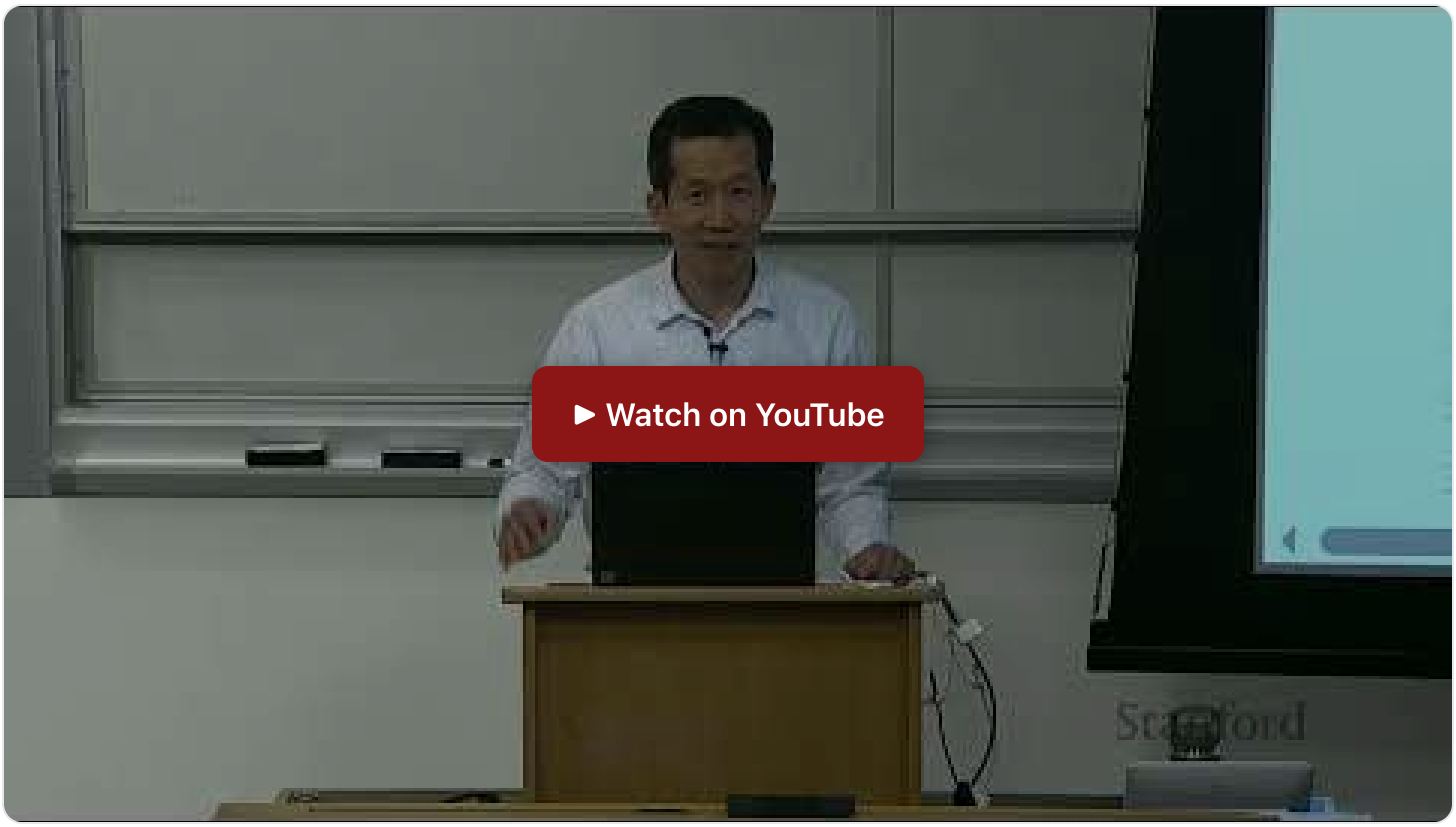
← Lec 15

Lec 17 →

Lecture 17: Alignment — RL 2

Stanford CS336 · Spring 2025 · 1:16:09 · Watch on YouTube ↗

📝 Auto-generated notes from the full lecture video (slides, equations, and live code) and transcript. Verify critical details against the source.



These notes provide a comprehensive overview of Policy Gradient methods in Reinforcement Learning, with a focus on their application to Language Models.

TL;DR

- Reinforcement Learning (RL) is key to surpassing human abilities in Language Models (LMs).
- Policy Gradient (PG) frameworks are conceptually clear, but practical implementation requires careful handling of high variance and changing data distributions.
- Baselines are crucial for reducing variance in PG updates without introducing bias. A common heuristic is to use the expected reward given the state, which relates to the

advantage function.
- GRPO (Group Relative Policy Optimization) leverages the natural group structure in LM settings (multiple responses per prompt) to provide a natural baseline, simplifying PPO by removing the critic.
- Building and scaling RL systems for LMs is more complex than pre-training due to inference workloads, managing multiple models (policy, old policy, reference model, critic), and distributed execution.

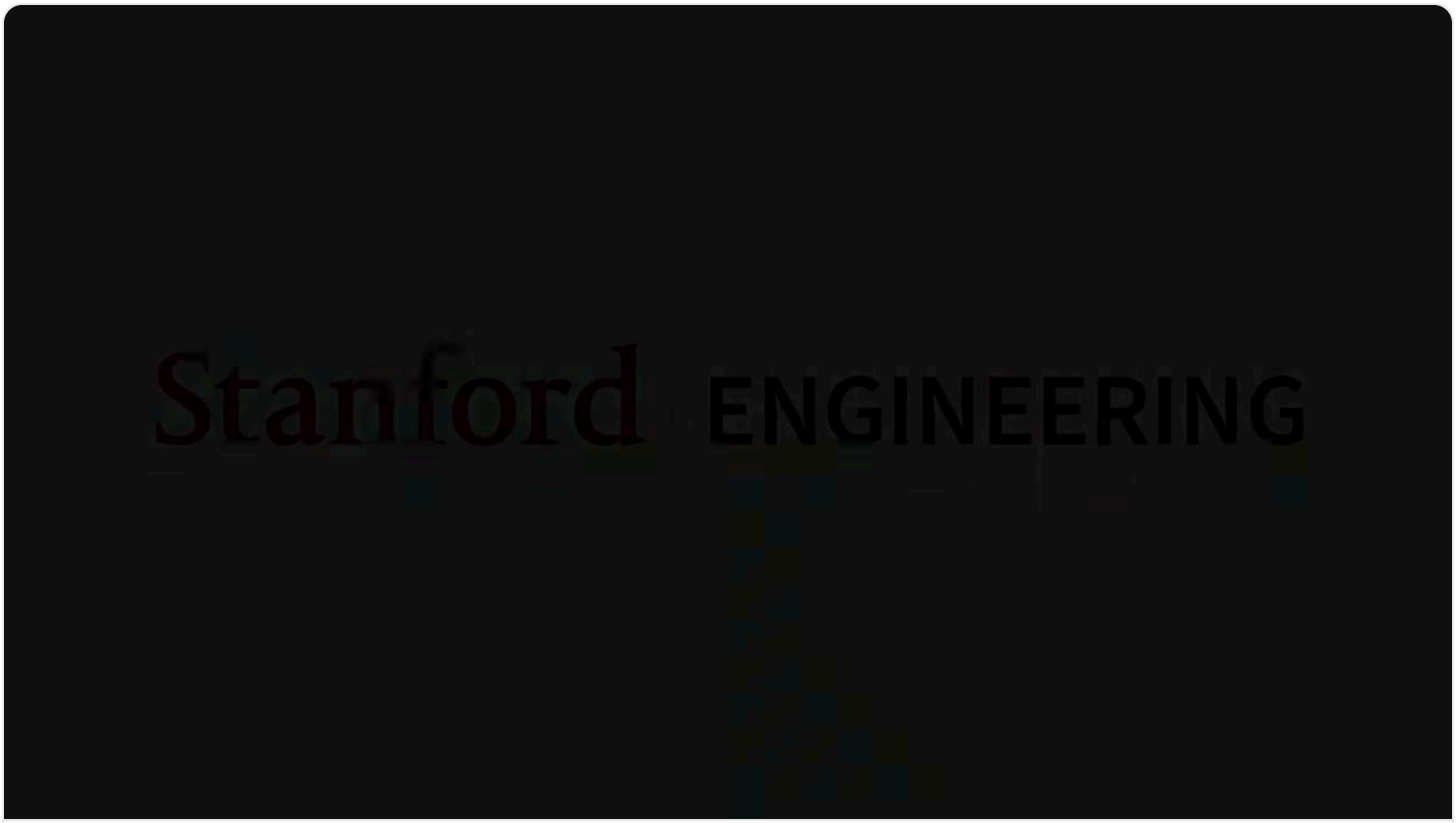
Key Concepts

- Policy Gradient (PG)
- Reinforcement Learning for Language Models (RLHF)
- State, Action, Reward definitions in LMs
- Sparse Rewards
- High Noise/Variance in PG
- Baselines for variance reduction
- Advantage Function
- Group Relative Policy Optimization (GRPO)
- Importance Sampling
- KL Penalty
- Freezing Parameters

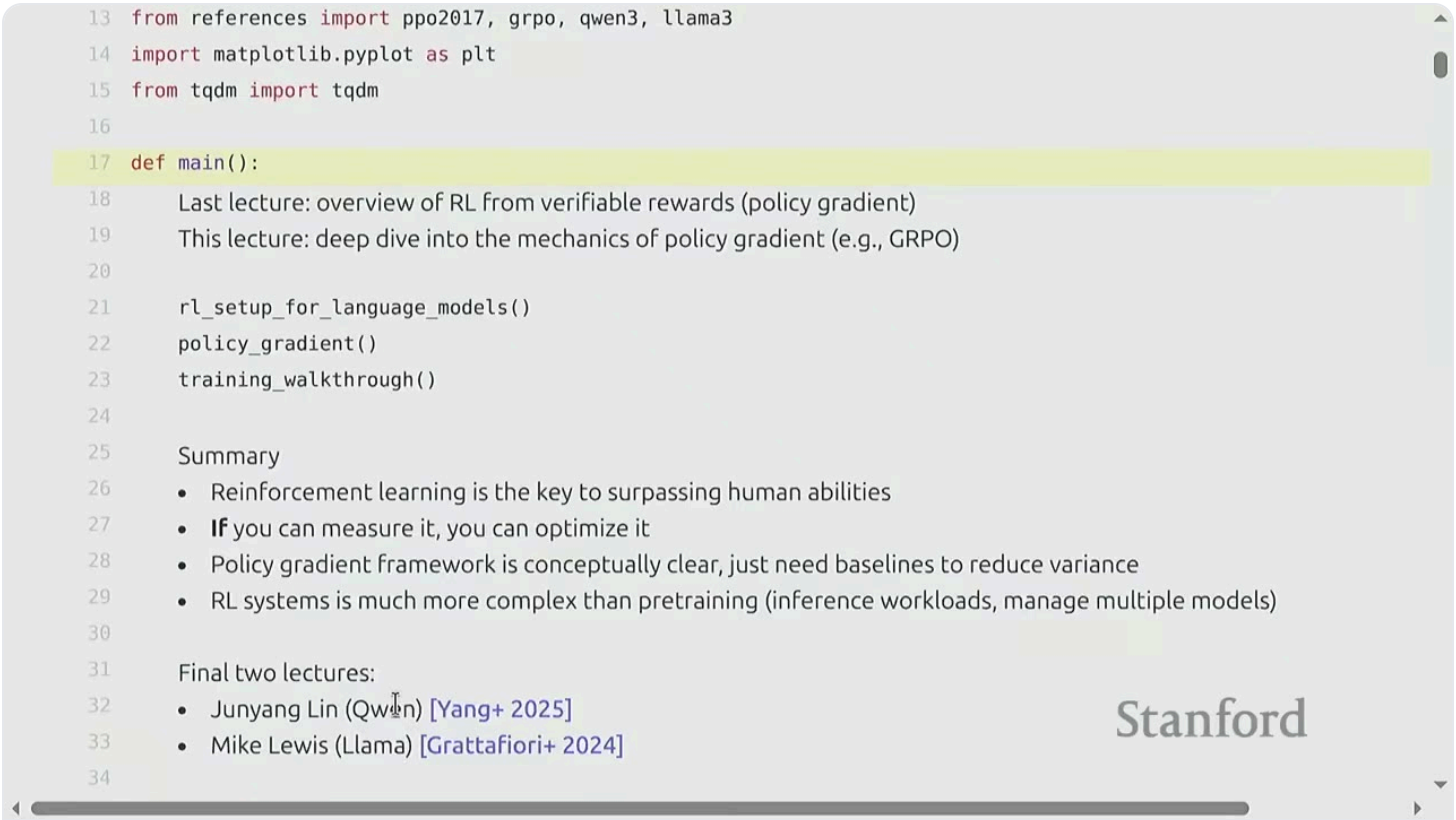
[00:00] Introduction and Course Context

The lecture begins by contextualizing its position within the course, noting it's one of the final lectures.

[00:20] Recap of Previous Lecture and Today's Focus



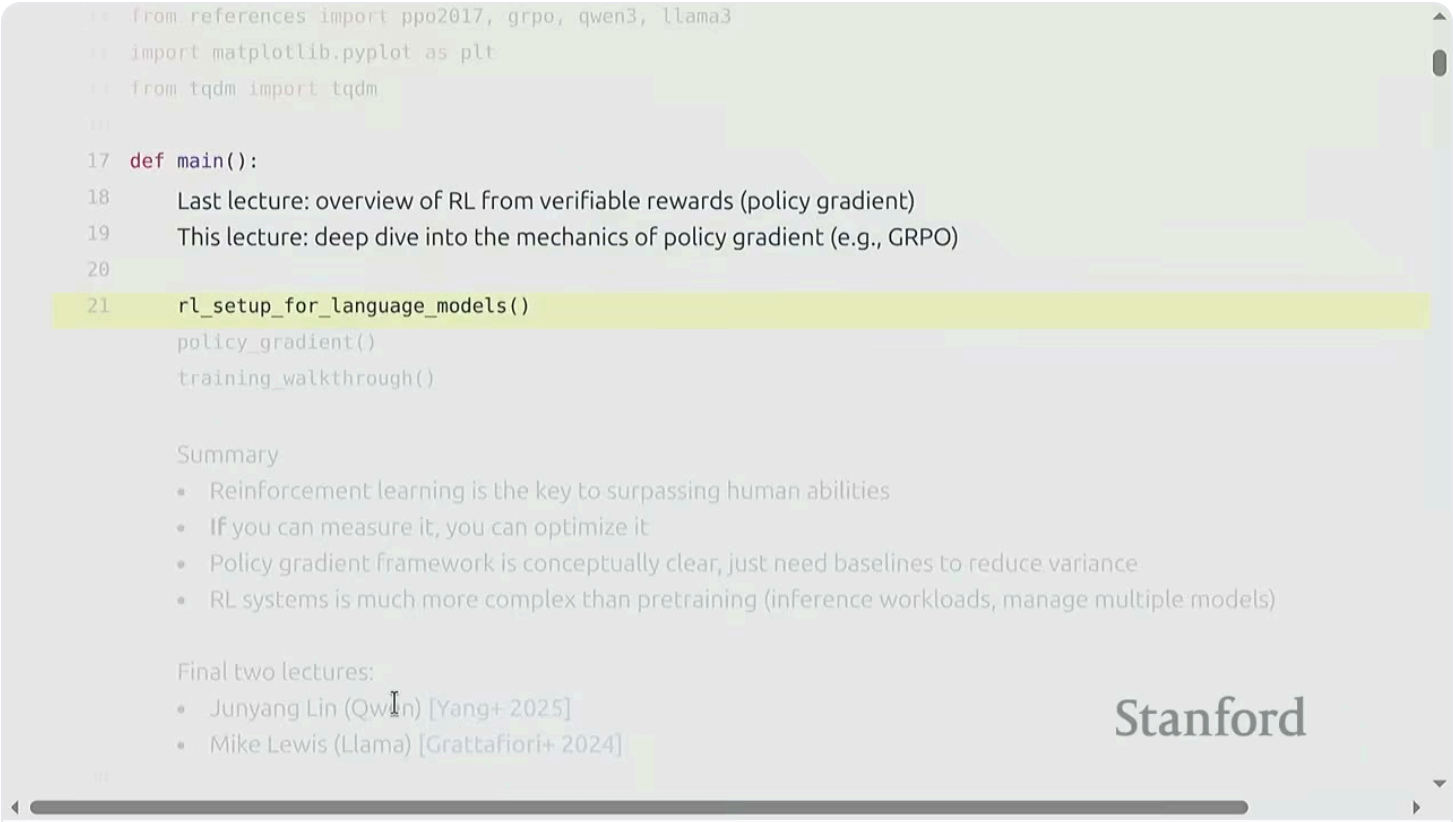
0:05 Stanford Engineering title slide.



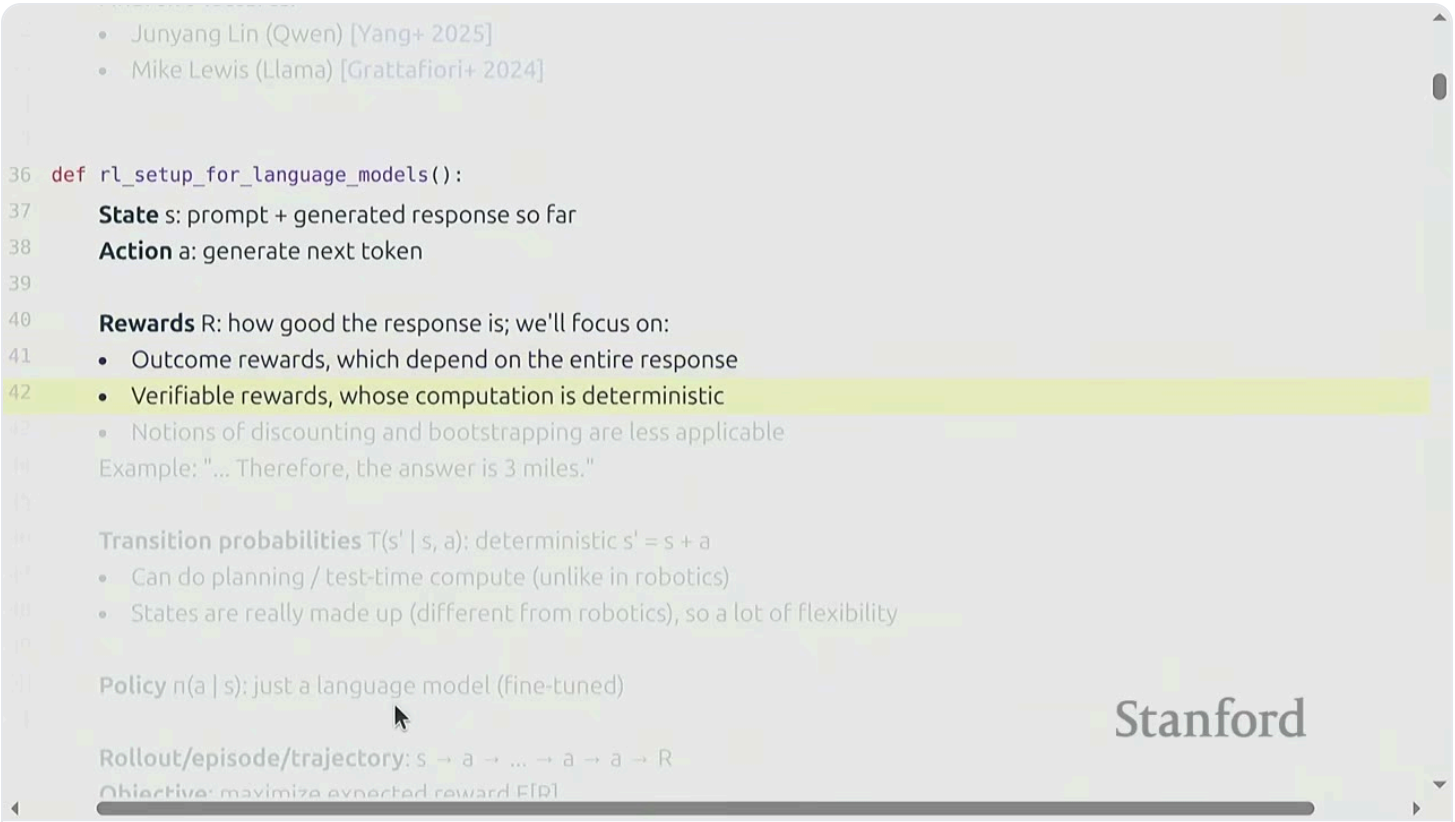
0:20 Python code: main function, lecture summary, and final lectures.

The previous lecture covered an overview of RL from verifiable rewards, including Policy Gradient (PG) algorithms like PPO and GRPO. Today's lecture will deep dive into the mechanics of PG, specifically GRPO. The goal is to go deeper into existing material, show code, and explain some of the underlying math.

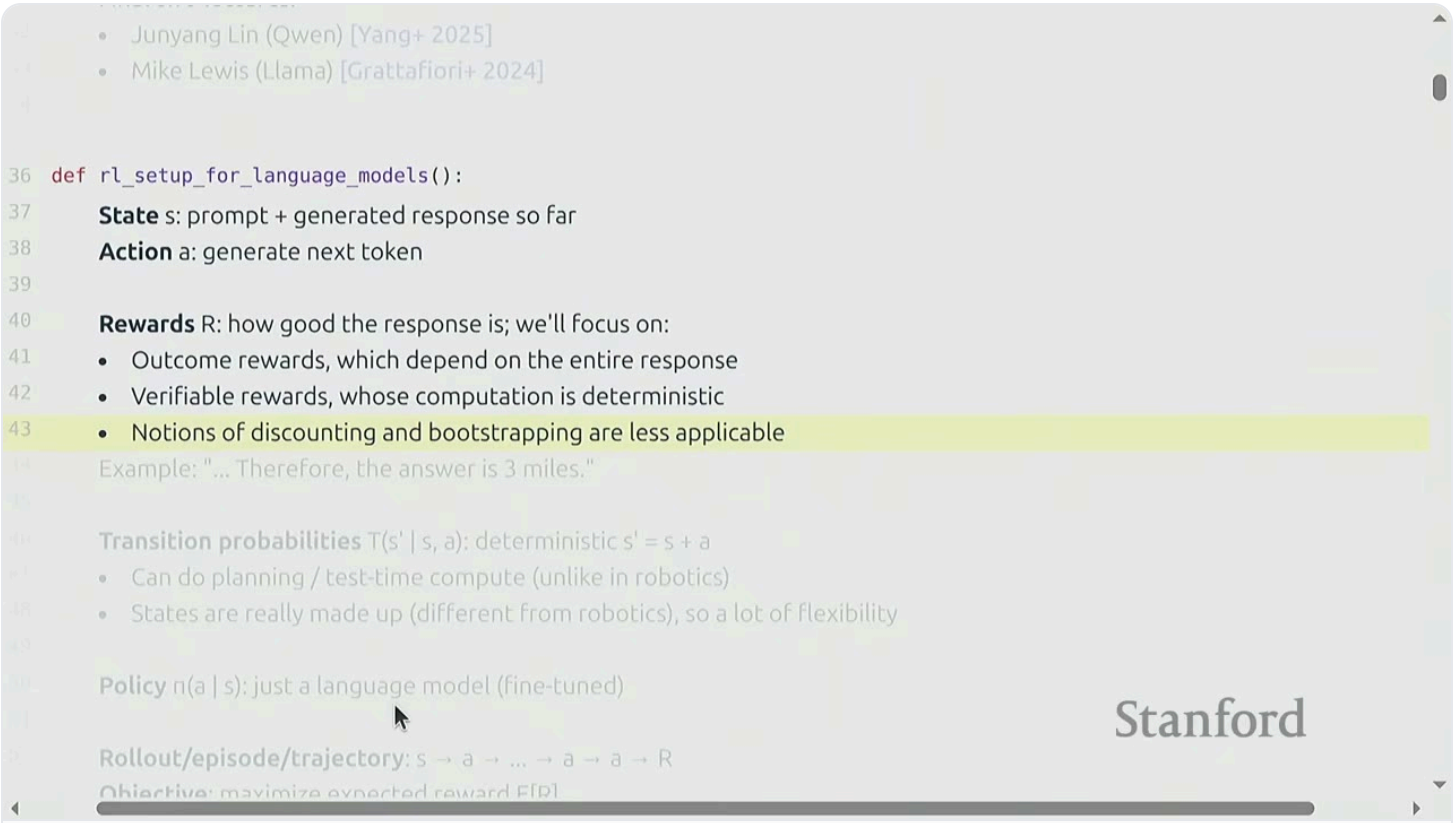
[00:52] RL Setup for Language Models



0:57 Python code: highlighting the rl_setup_for_language_models() function call.



1:40 Python code: defining state, action, and verifiable rewards for language models.



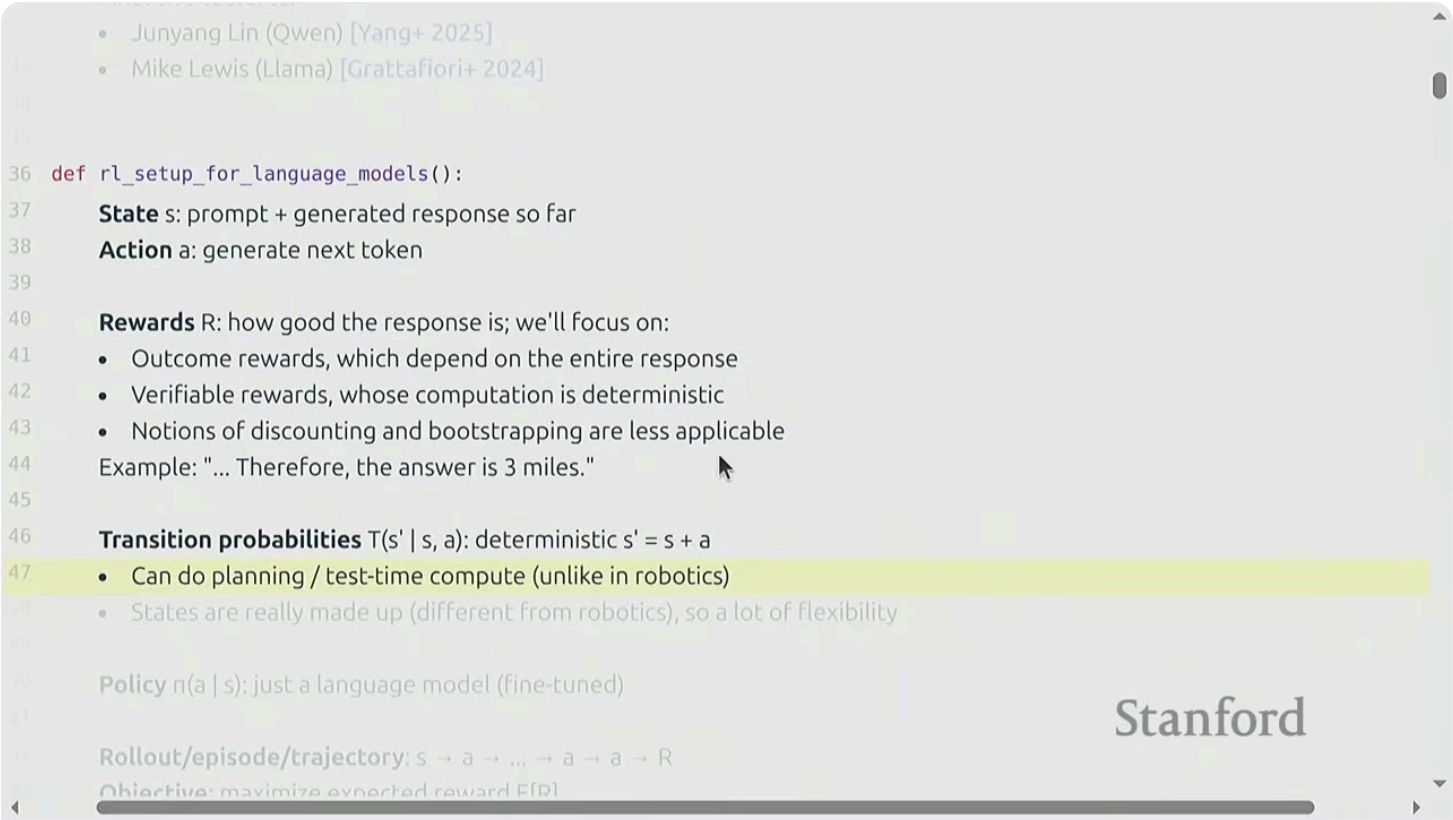
2:03 Python code: highlighting that discounting and bootstrapping are less applicable.

To clarify the RL framework in the context of Language Models:

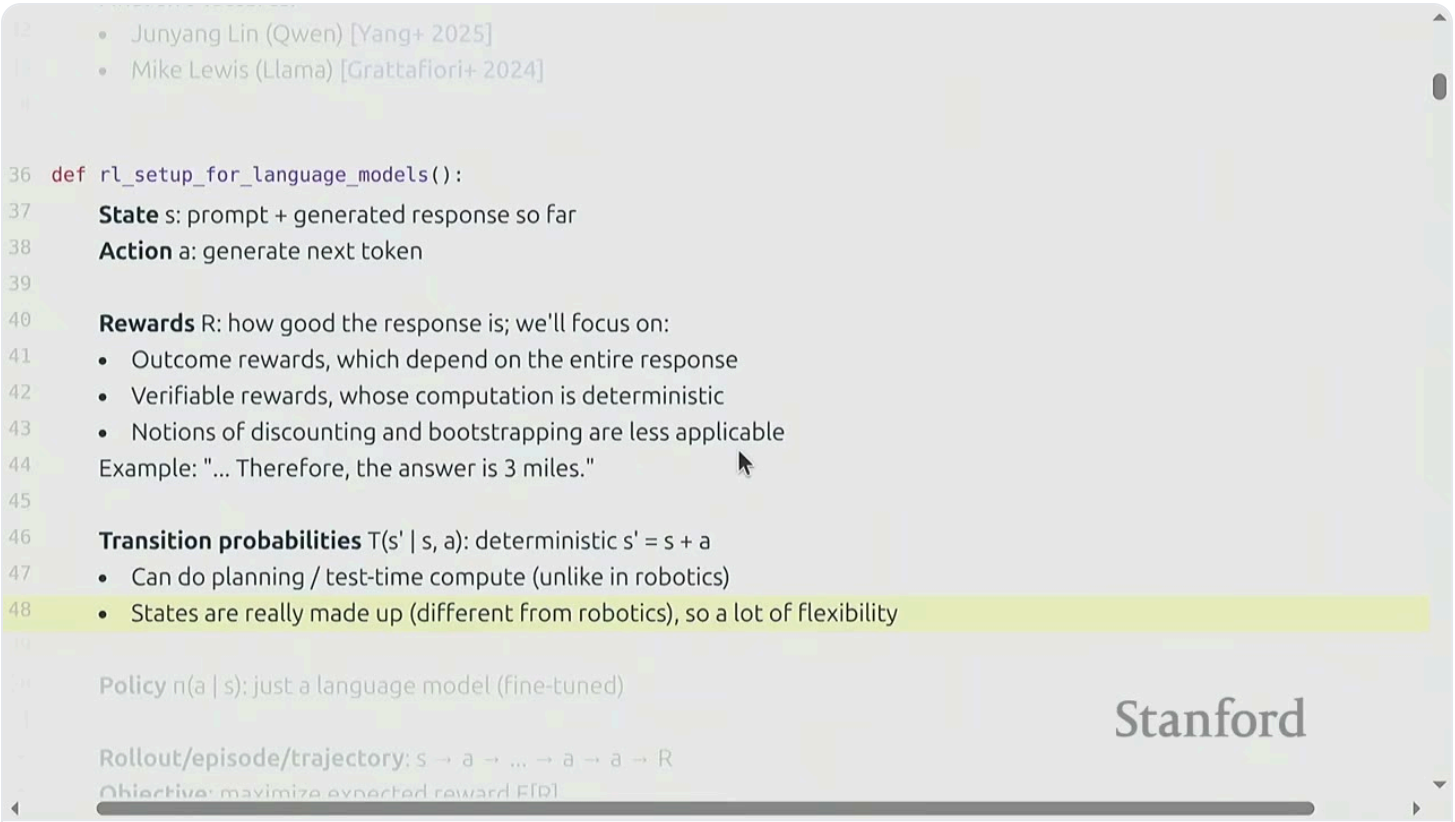
- **State (s):** The prompt + the generated response so far.
- **Action (a):** Generating the next token.
- **Rewards (R):** How good the response is.
    - **Outcome Rewards:** Depend on the entire response.
    - **Verifiable Rewards:** Computation is deterministic (e.g., a function returns a reward, no human labeling).
    - **Discounting and Bootstrapping:** Less applicable here because rewards are typically given only at the end of a sequence of actions (sparse reward).
        - *Example:* For a math problem, the reward is given only after the final answer is generated and checked against a ground truth.
    - Sparse and delayed rewards introduce difficulties but simplify conceptual thinking.
- **Transition Probabilities (T(s'|s, a)):** Deterministic: s' = s + a.
    - This is a key difference from robotics, where transition dynamics are complex and often unknown. In LMs, the transition is simply appending the action (next token) to the current state.
    - This determinism allows for planning or "test-time compute," which is a significant advantage over typical RL settings.
- **States are "made up":** Unlike robotics where states are grounded in the physical world (joint angles, images), in LMs, states are just sequences of tokens.
    - This introduces a lot of flexibility; the LM can generate its own "scratchpad" to derive answers. The challenge is not reaching a state, but ensuring the generated tokens lead to a correct answer.
    - The speaker emphasizes that while it's all RL, the intuitions and challenges change significantly in the LM context.
- **Policy (π(a|s)):** Just a language model (fine-tuned).
    - The policy takes the current state (prompt + generated response) and outputs a distribution over possible next tokens.
- **Rollout/Episode/Trajectory:** $s \to a \to a \to \cdots \to a \to R$.
    - A sequence of states and actions culminating in a single reward.

- **Objective:** Maximize expected reward $E[R]$.
    - The expectation is taken over prompts (given by the environment) and response tokens (generated by the policy).

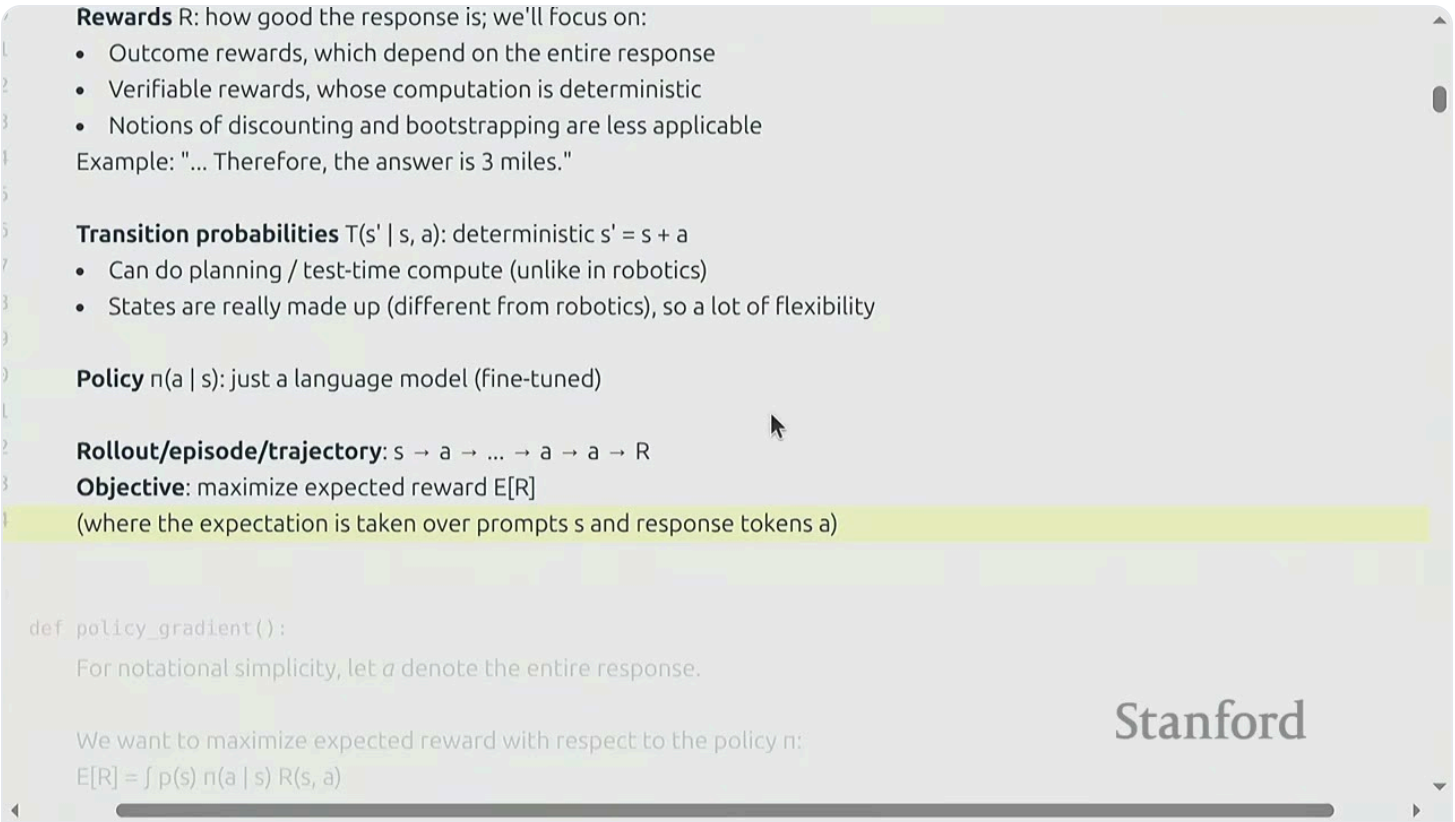
[03:09] Policy Gradient



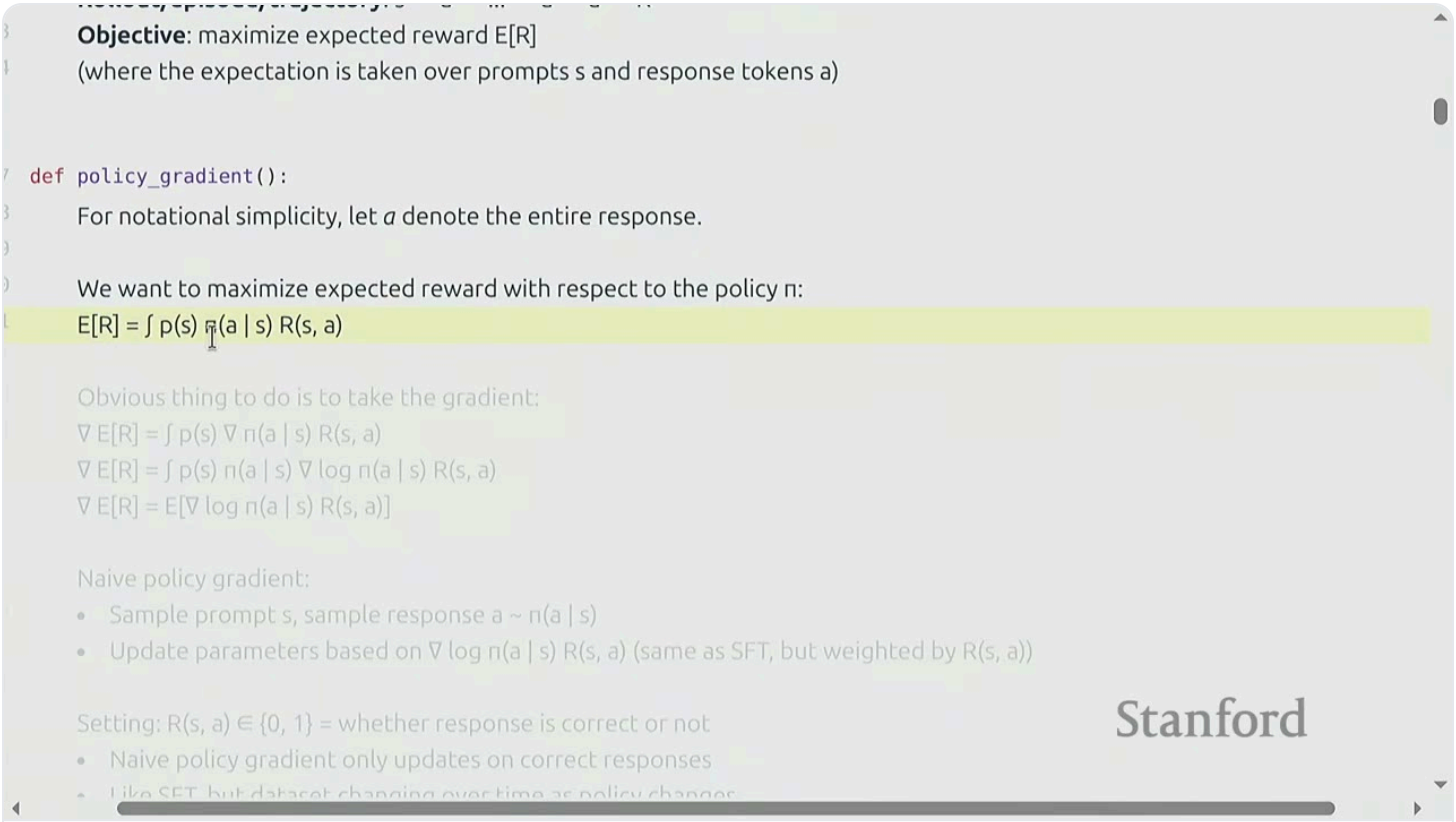
3:08 Python code: highlighting that planning/test-time compute is possible.



4:00 Python code: highlighting that states are made up, offering flexibility.



5:28 Python code: highlighting the objective to maximize expected reward E[R].



6:20 Python code: highlighting the equation for expected reward E[R].

Policy Gradient (PG) refers to a class of methods for learning a policy by iteratively improving it using gradient ascent.

For notational simplicity, let a denote the entire response (sequence of actions/tokens). This is valid in the outcome reward setting where the reward is given only at the end.

The objective is to maximize the expected reward with respect to the policy π:

$$E[R] = \int p(s)\pi(a|s)R(s,a)$$

The "obvious thing to do" is to take the gradient:

$$\nabla E[R] = \nabla \int p(s)\pi(a|s)R(s,a)$$

$$\nabla E[R] = \int p(s)\nabla\pi(a|s)R(s,a)$$

Using the identity $\nabla\pi(a|s) = \pi(a|s)\nabla \log \pi(a|s)$:

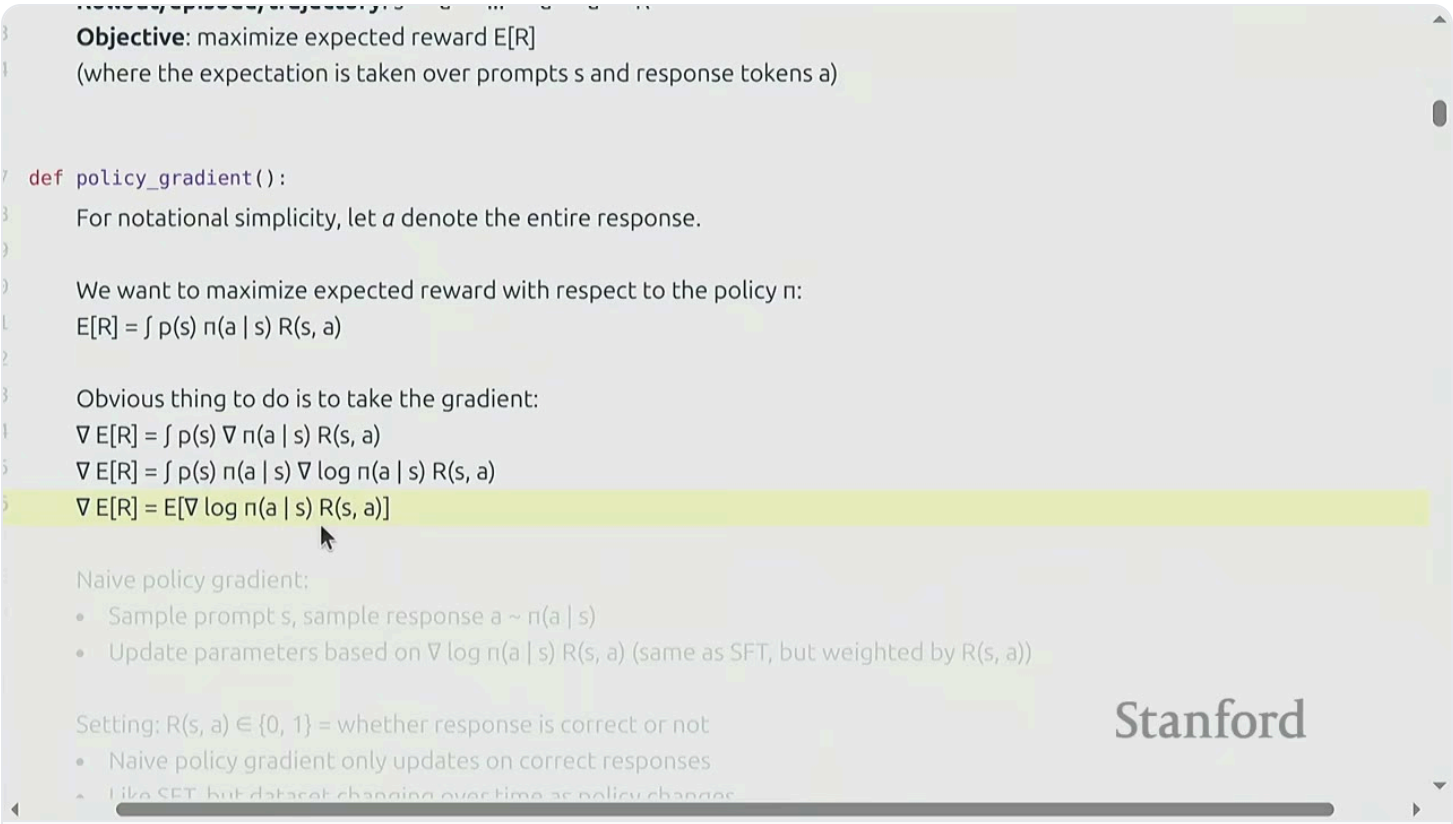
$$\nabla E[R] = \int p(s)\pi(a|s)\nabla \log \pi(a|s)R(s,a)$$

This can be rewritten as an expectation:

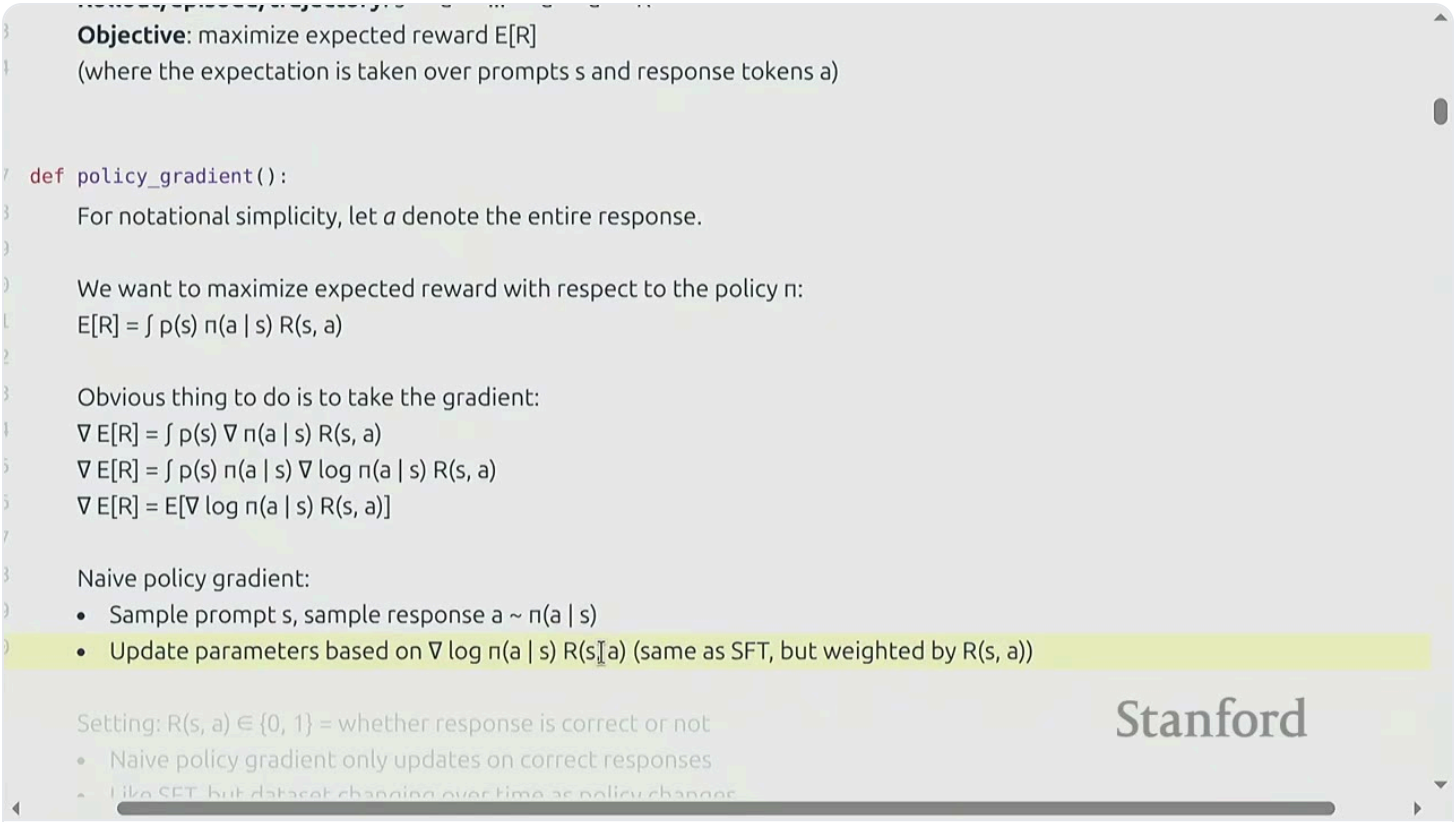
$$\nabla E[R] = E[\nabla \log \pi(a|s)R(s,a)]$$

This form should be familiar from standard PG derivations.

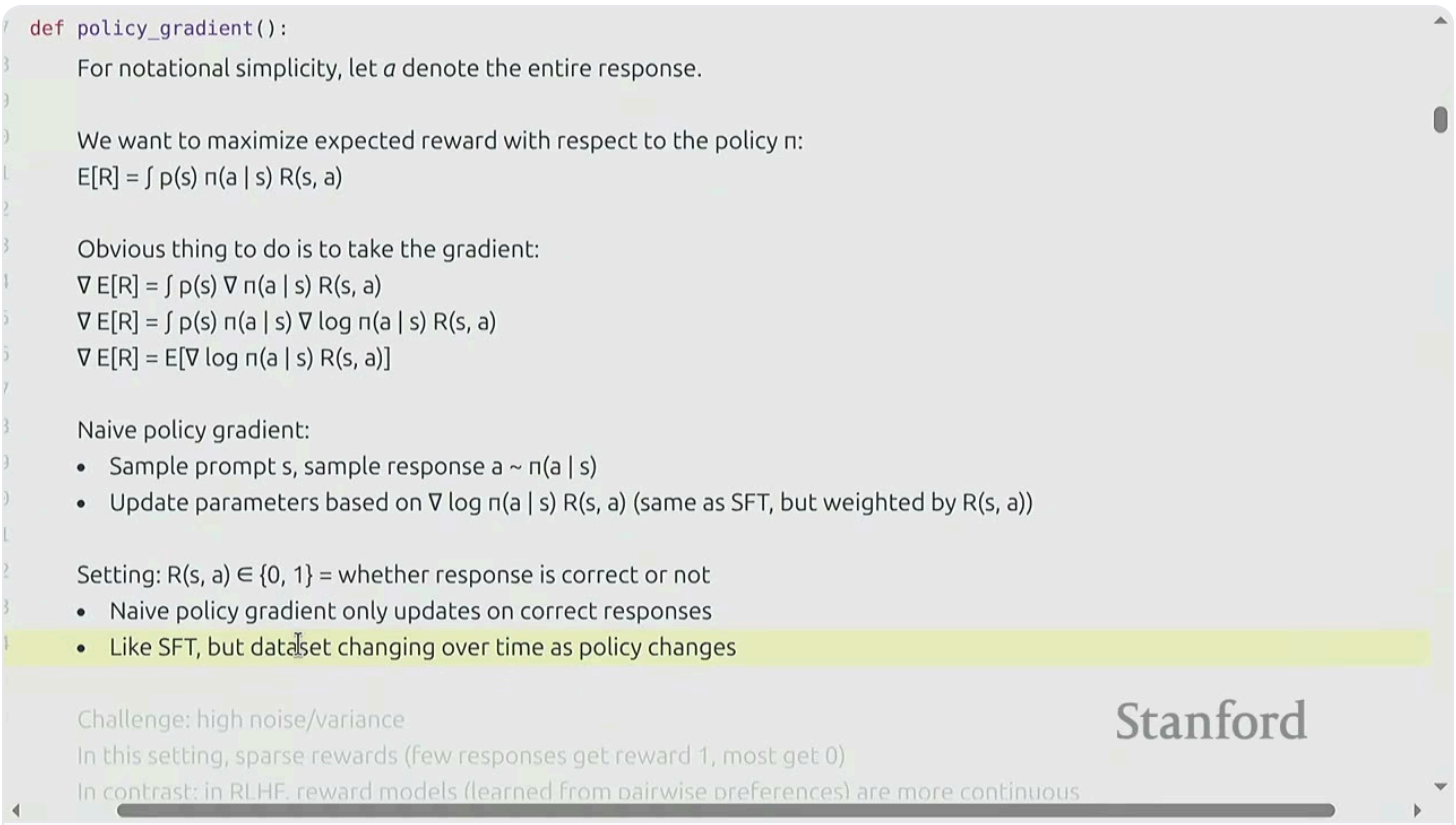
[07:16] Naive Policy Gradient



7:10 Python code: highlighting the gradient equation for expected reward E[R].



8:08 Python code: highlighting parameter updates based on the gradient of log π(a|s).



8:50 Python code: highlighting that the dataset changes over time like policy changes.

The naive policy gradient algorithm proceeds as follows: 1. Sample a prompt $s \sim p(s)$. 2. Sample a response $a \sim \pi(a|s)$. 3. Update parameters based on $\nabla \log \pi(a|s) R(s,a)$.

Interpretation: This is essentially the same as Supervised Fine-Tuning (SFT), but weighted by the reward $R(s,a)$. - In SFT, a human provides the correct response a, and the model is updated to maximize the probability of that a. - In PG, the model generates a, and the update is weighted by the reward.

Setting: Consider a binary reward $R(s,a) \in \{0,1\}$, indicating whether the response is correct or not. - **Naive PG updates only on correct responses:** If $R(s,a) = 0$, the term $\nabla \log \pi(a|s) R(s,a)$ becomes zero, so no update is made. Only responses with $R(s,a) = 1$ contribute to the update, effectively performing SFT on correct responses. - **Data set changing over time:** Unlike traditional SFT, the data distribution from which responses are sampled changes as the policy π evolves.

Challenge: High Noise/Variance - In this sparse reward setting (few responses get reward 1, most get 0), PG suffers from high noise and variance. - If the policy is so bad that it rarely generates correct responses, most updates will be zero, leading to slow or no learning. This is a "stuck" scenario. - In contrast, RLHF models (learned from pairwise preferences) often have more continuous reward models, providing denser feedback.

Question: Why is the data set changing over time? - Each iteration, the policy parameters are updated. The next time a response is sampled, it comes from the *new* policy, which has a different distribution. This means the "data" (trajectories) used for updates changes over time. - Intuitively, if the model can get some easy questions correct, it updates, and hopefully generalizes to slightly harder questions, leading to a gradual increase in reward over time.

Question: Why not make the reward -1 for incorrect responses? - This is a valid idea, and it's implicitly handled by using baselines, which will be discussed next.

[15:43] Baselines

Recall: $\nabla E[R] = E[\nabla \log \pi(a|s) R(s,a)]$ is an unbiased estimate of $\nabla E[R]$, but its variance can be high.

Example: Two States Consider a simplified MDP with two states (prompts) S1 and S2, and two actions A1 and A2 for each. - S1: A1 $\rightarrow$ Reward 11, A2 $\rightarrow$ Reward 9 - S2: A1 $\rightarrow$ Reward 0, A2 $\rightarrow$

Reward 2

The optimal policy would be to choose A1 in S1 (reward 11) and A2 in S2 (reward 2). However, if we only look at local rewards, A2 in S1 (reward 9) might seem better than A2 in S2 (reward 2). A naive PG update might push the policy towards A2 in S1 because 9 > 2, even though A1 is better in S1. This illustrates how local reward signals can be misleading without a broader context.

Idea: Maximize the baselined reward: $E[R - b(s)]$. - This involves subtracting a state-dependent baseline $b(s)$ from the reward. - This is just $E[R]$ shifted by a constant $E[b(s)]$ that doesn't depend on the policy π. - Therefore, maximizing $E[R - b(s)]$ is equivalent to maximizing $E[R]$. - The update rule becomes: $\nabla \log \pi(a|s)(R(s,a) - b(s))$. - This is valid for *any* $b(s)$ that does not depend on the action a.

Question: What $b(s)$ should we use? Let's revisit the two-state example. Assume a uniform distribution over (s, a) and $|\nabla \log \pi(a|s)| = 1$ for simplicity. - **Naive variance:** Using the original rewards (11, 9, 0, 2), the variance is 5.32. - **Define baseline:** $b(S1) = 10$, $b(S2) = 1$. (These are chosen to be close to the average reward for each state). - **Baseline variance:** Subtracting the baseline, the new rewards are (1, -1, -1, 1). The variance is 1.15. - **Variance reduced from 5.32 to 1.15.** This demonstrates the power of baselines in reducing variance.

Optimal $b^*(s)$: The optimal baseline $b^*(s)$ that minimizes variance is given by:

$$b^*(s) = \frac{E[(\nabla \log \pi(a|s))^2 R(s,a)|s]}{E[(\nabla \log \pi(a|s))^2|s]}$$

- This is difficult to compute in practice. - **Heuristic:** A common heuristic is to use the mean reward: $b(s) = E[R|s]$. - This is still hard to compute and must be estimated.

[26:28] Advantage Functions

The choice of $b(s) = E[R|s]$ has connections to advantage functions in RL. - **Value function V(s):** $E[R|s]$ = expected reward from state s. - **Q-function Q(s,a):** $E[R|s,a]$ = expected reward from state s taking action a. - Note: Q and R are the same here because we're assuming outcome rewards and a has all actions. In general, R would be the return from that state, not just a single reward.

- **Definition (Advantage):** $A(s,a) = Q(s,a) - V(s)$.
- **Intuition:** How much better is action a than expected from state s?
- **If $b(s) = E[R|s]$, then the baselined reward is identical to the advantage!**

    - $(R(s,a) - b(s)) = (R(s,a) - E[R|s]) = A(s,a)$.
- **In general:** The update rule is $E[\nabla \log \pi(a|s)\delta]$, where δ is an advantage-like quantity.
- There are multiple choices for δ, as we'll see later. (Refer to CS224R lecture notes for more derivations).

[30:10] Training Walkthrough

The walkthrough will use Group Relative Policy Optimization (GRPO) from Shao+ 2024. - **Simplification to PPO:** GRPO removes the critic (value function). - **Leverages group structure:** The LM setting (multiple responses per prompt) provides a natural baseline $b(s)$. - The ability to generate multiple responses for a single prompt creates a "group" of trajectories. This group structure is leveraged to compute a natural baseline (e.g., the average reward within the group), which helps reduce variance. - If every rollout looked different (no natural group structure), GRPO wouldn't make sense, and a value function would typically be used instead.

[33:00] GRPO Algorithm (Pseudocode)

The pseudocode for GRPO is presented (Algorithm 1: Iterative Group Relative Policy Optimization). 1. Initialize policy model $\pi_{\theta_{init}}$, reward models r_{ϕ_i}, task prompts D, hyperparameters ϵ, β, μ. 2. For iteration $l = 1, \ldots, M$ do: 3. Reference model $\pi_{ref} \leftarrow \pi_{\theta_l}$. 4. For step $s = 1, \ldots, M$ do: 5. Sample a batch D_b from D. 6. Update the old policy model $\pi_{\theta_{old}} \leftarrow \pi_{\theta_l}$ for each question $q \in D_b$. 7. Sample G outputs $o_1^q, \ldots, o_G^q$ for each sampled output o_i^q by running π_{θ_l}. 8. Compute rewards r_i^q for o_i^q through group relative advantage estimation. 9. For GRPO iteration $t = 1, \ldots, \mu$ do: 10. Update the policy model π_{θ_l} by maximizing the GRPO objective (Equation 21). 11. Update π_{θ_l} through continuous training using a replay mechanism. 12. Output π_{θ_l}.

[34:39] Simple Task Definition

A simple task is defined to demonstrate the concepts: **sorting n numbers**. - **Prompt:** n numbers (e.g., `[1, 0, 2]`). - **Response:** n numbers (e.g., `[0, 1, 2]`). - **Reward:** Should capture how close to sorted the response is.

Reward Function Choices: 1. **Exact Match Reward:** Returns 1 if the response is perfectly sorted (matches ground truth), 0 otherwise. - This is a sparse reward, leading to high variance and slow learning if the initial policy is poor. 2. **Partial Credit Reward (Inclusion Reward):** Returns the

number of positions where the response matches the ground truth. - *Example:* `[1, 0, 2]` sorted is `[0, 1, 2]`. - `[0, 1, 2, 3]` → reward 4 (perfect match). - `[3, 1, 0, 2]` → reward 1 (only '2' is in the correct position). - `[0, 3, 1, 2]` → reward 1 (only '0' is correct). - This is still somewhat sparse. 3. **Partial Credit Reward (Inclusion + Ordering Reward):** - Gives 1 point for each token in the prompt that shows up in the response. - Gives 1 point for each adjacent pair in the response that's sorted. - *Example:* `[1, 0, 2]` sorted is `[0, 1, 2]`. - `[0, 1, 2, 3]` → inclusion reward 4, ordering reward 3 → total 7. - `[3, 1, 0, 2]` → inclusion reward 3, ordering reward 0 → total 3. - `[0, 3, 1, 2]` → inclusion reward 4, ordering reward 2 → total 6. - This provides more continuous feedback, which is generally better for RL. The second reward function (inclusion + ordering) will be used for the walkthrough.

[37:28] Simple Model Definition

A simple model is defined to solve the sorting task: - Assumes fixed prompt and response length. - Captures positional information with separate per-position parameters. - Decodes each position in the response independently (not autoregressive). This simplifies the code significantly.

Model Architecture: - `nn.Embedding` for vocabulary. - For each position, a matrix for encoding and a matrix for decoding. - `einsum` is used for transformations.

Forward Pass: 1. Embed the prompts into `[batch, pos, dim]` embeddings. 2. Transform per-prompt position matrix, collapse into one vector: `encoded = einsum(embeddings, self.encode_weights, "batch pos dim1, pos dim1 dim2 -> batch dim2")`. This effectively sums out the position and dim1, resulting in a single vector per batch. 3. Turn into one vector per response position: `decoded = einsum(encoded, self.decode_weights, "batch dim2, batch pos dim1 -> batch pos dim1")`. This applies a position-specific matrix to the encoded vector. 4. Convert to logits (input and output share embeddings): `logits = einsum(decoded, self.embedding.weight, "batch pos dim1, vocab dim1 -> batch pos vocab")`.

[41:16] Generating Responses

The `generate_responses` function takes prompts, the model, and the number of responses to generate. - It samples `num_responses` independently for each `[batch, pos]` from the model's logits using `torch.multinomial`. - The sampled responses are then reshaped to `[batch, trial, pos]`.

[41:16] Computing Rewards and Deltas

- **Compute Rewards:** The `compute_reward` function iterates through each prompt and each generated response, calling the defined reward function (e.g., `sort_inclusion_ordering_reward`).
- **Compute Deltas:** The `compute_deltas` function takes the rewards and converts them into the δ values for updates.
  - **`rewards` mode:** $\delta = R$. This is the naive PG.
  - **`centered_rewards` mode:** $\delta = R - \text{mean}(R)$. The mean is computed over all responses for each prompt.
    - If all rewards for a prompt are the same, δ will be 0, meaning no update. The intuition is that if all generated responses are equally good (or bad), there's no signal to favor one over another.
  - **`normalized_rewards` mode:** $\delta = (R - \text{mean}(R))/\text{std}(R)$. This normalizes the centered rewards by their standard deviation. This makes the updates invariant to the scale of the reward.
  - **`max_rewards` mode:** $\delta = R$ if R is the maximum reward for the batch, 0 otherwise. This is an "all or nothing" approach, focusing only on the best responses.

[47:46] Computing Log Probabilities

The `compute_log_probs` function calculates the log probabilities of the generated responses under the current model. - It passes the prompts through the model to get logits. - Converts logits to log probabilities using `F.log_softmax`. - Indexes into these log probabilities using the generated responses to get the specific log probability for each token in each response.

[49:33] Computing Loss

The `compute_loss` function calculates the loss to update the model parameters. - **`naive` mode:** `loss = -einsum(log_probs, deltas, "batch trial pos, batch trial -> batch trial").mean()`. This is the basic PG update: $\nabla \log \pi(a|s)\delta$. - **`clipped` mode (GRPO/PPO-like):** - This involves computing a ratio of current log probs to old log probs (`unclipped_ratios = log_probs - old_log_probs`). - This ratio is then clipped between `1 - epsilon` and `1 + epsilon`. - The clipped ratio is multiplied by `deltas`. - The minimum of this clipped term and another term (related to the unclipped ratio) is taken. - This ensures that the policy doesn't deviate too much from the old policy in a single update. - The negative sign is for converting reward maximization into loss minimization.

[50:55] Freezing Parameters

Motivation: In GRPO, you'll see ratios like $\pi(a|s)/\pi_{old}(a|s)$. It's important to freeze and not differentiate through π_{old}. - **Toy Example:** If `w` drives both `p` (current policy) and `p_old` (old policy), and you compute `ratio = p / p_old` and then `ratio.backward()`, the gradient `grad.w` will be 0 if `p_old` is not frozen. - **Correct Way:** To treat `p_old` as a constant, use `with torch.no_grad(): p_old = torch.nn.Sigmoid(w)`. This allows getting the value of `p_old` without building a computation graph for it.

[51:59] KL Penalty

Sometimes, an explicit KL penalty is used to regularize the model. - The KL divergence between two distributions P and Q is $KL(P||Q) = E_{x\sim P}[\log(P(x)/Q(x))]$. - This can be rewritten as $E_{x\sim P}[\log(P(x)) - \log(Q(x))]$. - If we want to estimate $KL(\pi_\theta||\pi_{ref})$, we can sample from π_θ and compute $E_{a\sim\pi_\theta}[\log \pi_\theta(a|s) - \log \pi_{ref}(a|s)]$. - An alternative estimate for $KL(\pi_\theta||\pi_{ref})$ is $E_{a\sim\pi_\theta}[\frac{\pi_{ref}(a|s)}{\pi_\theta(a|s)}(\log \pi_\theta(a|s) - \log \pi_{ref}(a|s))]$. This particular estimate can have lower variance. - The math works out because $E_{a\sim\pi_\theta}[\frac{\pi_{ref}(a|s)}{\pi_\theta(a|s)}] = 1$.

[53:38] Summary of Components

So far, the components needed for the GRPO algorithm are: - Generate responses. - Compute rewards R and δ (rewards, centered rewards, normalized rewards, max rewards). - Compute log probs of responses. - Compute loss from log probs and δ (naive, unclipped, clipped). - Freezing parameters.

[53:48] GRPO Algorithm (Full Code)

The full `run_policy_gradient` function is presented, implementing the GRPO algorithm.

Outer Loop (Epochs): - Iterates `num_epochs` times. - If KL penalty is used, the reference model (`ref_model`) is updated every few epochs by cloning the current model. This `ref_model` is used for the KL penalty.

Inside the Epoch Loop: 1. **Sample responses and evaluate their rewards:** - `responses = generate_responses(...)` - `rewards = compute_reward(...)` - `deltas = compute_deltas(...)` (mode can be `rewards`, `centered_rewards`, `normalized_rewards`, `max_rewards`).

2. **Compute log probs under the reference model (if KL penalty is used):**
    - `ref_log_probs = compute_log_probs(..., model=ref_model)` (with `torch.no_grad()` to freeze `ref_model`).
3. **Compute log probs under the current model:**
    - `log_probs = compute_log_probs(...)`
4. **Take a number of steps (`num_steps_per_epoch`) to update the model:**
    - This inner loop performs multiple gradient steps using the same set of generated responses.
    - `loss = compute_loss(...)` (mode can be `naive`, `unclipped`, `clipped`).
    - If KL penalty is used, `loss += kl_penalty * compute_kl_penalty(...)`.
    - Backpropagate and update parameters: `optimizer.zero_grad()`, `loss.backward()`, `optimizer.step()`.

[59:59] Experiments

The code then runs some experiments using the defined task and model.

Example Walkthrough: - The output of a run is shown, detailing the prompt, generated responses, log probabilities, rewards, and deltas at each step. - **Prompt:** `[1, 0, 2]` - **Responses:** Initially, the model generates random sequences. - **Rewards:** The `sort_inclusion_ordering_reward` is used. - **Deltas:** Initially, deltas are 3.0 for all responses (as all rewards are 3.0 in this specific run due to fixed random seed). - **Centering:** When centering is applied, deltas become 0 if all rewards are the same. This means no update is made. - **Normalization:** Normalization makes the updates invariant to the scale of the reward. - **`max_rewards`:** Only the responses with the maximum reward contribute to the update.

Observation: If all generated responses for a prompt have the same reward, `centered_rewards` will be 0, leading to no updates for that prompt. This is a design choice in GRPO, as it assumes there's no signal to favor one response over another within that group.

Freezing Parameters (Important for GRPO/PPO): - In GRPO/PPO, you often see ratios like $\pi(a|s)/\pi_{old}(a|s)$. It's crucial to freeze π_{old} and not differentiate through it. - **Toy Example:** If `w` drives both `p` (current policy) and `p_old`, and you compute `ratio = p / p_old` and `ratio.backward()`, the gradient `grad.w` will be 0 if `p_old` is not frozen. - **Correct Way:** Use

`with torch.no_grad(): p_old = torch.nn.Sigmoid(w)`. This allows getting the value of `p_old` without building a computation graph for it.

KL Penalty: - The KL penalty is used to regularize the model, preventing the new policy from deviating too much from a reference policy. - The `compute_kl_penalty` function estimates $KL(\pi_\theta||\pi_{ref})$.

Training Loop: - The outer loop iterates over epochs. - Inside, responses are generated, rewards and deltas are computed. - If KL penalty is used, a `ref_model` is cloned from the current model every few epochs and used for regularization. - The inner loop iterates `num_steps_per_epoch` times, performing gradient updates using the computed loss. This is where the model learns from the generated responses.

Results: - The training is run for 100 epochs, with 10 steps per epoch. - **Naive PG:** The train loss decreases, and mean reward increases, but the loss curve is very noisy. - **Centered Rewards:** The train loss is less noisy, and mean reward increases more smoothly. - **Normalized Rewards:** Similar to centered rewards, but potentially more stable.

Final Observations from the run: - The model struggles to perfectly sort the numbers, even with partial credit. - The `sort_inclusion_ordering_reward` is a double-edged sword: it provides more signal, but the model might get "stuck" in a local optimum where it's happy generating partially correct answers, but not fully sorted ones. - The model often produces numbers that are present in the prompt but not in the correct order. - The loss curves can be misleading. A decreasing loss doesn't always mean the model is improving in a meaningful way, especially when the data distribution is changing.

[1:13:24] Summary and Practical Takeaways

- **Reinforcement Learning is the key to surpassing human abilities:** Labeled data only allows mimicking existing behavior. RL enables learning beyond human demonstration.
- **If you can measure it, you can optimize it:** The core idea of RL is to define a reward function and then optimize the policy to maximize it.
- **Policy Gradient framework is conceptually clear:** Define objective, subtract baseline to reduce variance, estimate gradient.
- **RL systems are much more complex than pre-training:**

- **Inference workloads:** Generating responses from the LM is computationally expensive.
- **Manage multiple models:** You need to keep track of the current policy, old policy (for importance sampling/clipping), and reference policy (for KL regularization).
- **Distributed execution:** RL often requires parallelizing environment interactions, inference, and training across multiple machines.

Practical Takeaways

- **Variance Reduction is Key:** Baselines are essential for making Policy Gradient methods practical, especially with sparse rewards.
- **Reward Design Matters:** The choice of reward function (sparse vs. dense, exact vs. partial credit) profoundly impacts learning stability and efficiency. Dense, well-shaped rewards are generally preferred.
- **Understanding GRPO/PPO Mechanics:** These algorithms use importance sampling and clipping to stabilize updates and prevent large policy shifts, which is crucial when the data distribution changes.
- **Computational Complexity:** RL for LMs is resource-intensive, requiring careful management of inference, multiple models, and distributed systems.

Open Questions / Things to Remember

- **Credit Assignment Problem:** How do we correctly attribute reward to individual actions in a long sequence, especially when rewards are sparse? This is a fundamental challenge in RL.
- **Reward Hacking:** How to design reward functions that truly capture desired behavior without being "hackable" by the agent (i.e., finding unintended ways to maximize reward)?
- **Balancing Exploration and Exploitation:** How to ensure the policy explores enough to find better strategies while also exploiting known good strategies?
- **Stability of RL Training:** The changing data distribution and high variance make RL training notoriously unstable and sensitive to hyperparameters.
- **The "Lie" of Loss Curves:** In RL, monitoring loss over time can be misleading because the underlying data distribution (generated responses) is constantly changing. Reward metrics are often more indicative of progress.

← Lec 16

🧠 Algorithm Deep Dives

Going beyond the lectures — the genuinely tricky algorithmic questions, worked through with derivations, intuition, and the pitfalls that trip people up.

Screenshots show *what* the slides said; this section is about *why* the algorithms work. Each entry poses a sharp question and answers it in depth.

Jump to: Tokenization & Byte-Pair Encoding · Transformer Architecture Choices · Mixture of Experts · GPU Kernels & FlashAttention · Parallelism · Scaling Laws & compute-optimal training · Inference · Alignment math

Tokenization & Byte-Pair Encoding

> **Q: A naive implementation of BPE training involves repeatedly scanning the entire corpus to find the most frequent pair, which has a terrible time complexity of roughly $O(\text{CorpusSize} \times \text{NumMerges})$. How is BPE training implemented efficiently in practice?**

A thorough and efficient implementation of the BPE training algorithm is critical for handling large corpora. The naive approach is computationally infeasible. The key to an efficient implementation lies in using appropriate data structures to avoid re-scanning the corpus for every single merge.

The core operations in each step of the training loop are: 1. Find the most frequent adjacent pair of tokens. 2. Merge this pair into a new token. 3. Update the corpus and pair counts for the next iteration.

An efficient algorithm optimizes these steps as follows:

1. Initial Corpus Representation and Pair Counting

Instead of treating the corpus as a monolithic flat array of bytes, we first process it into a more structured format. A common approach is to split the text into "words" (e.g., by whitespace) and

represent each word as a sequence of tokens. This is not strictly necessary for the algorithm's efficiency but aligns with common practice (like in GPT-2).

The most important step is to perform **a single pass** over the entire corpus at the beginning to build a complete frequency count of all adjacent token pairs. This can be stored in a hash map.

```
# Pseudocode for initial counting
def get_initial_pair_counts(corpus_words):
    pair_counts = {}
    for word in corpus_words:
        tokens = list(word.encode('utf-8')) # or some other initial tokenization
        for i in range(len(tokens) - 1):
            pair = (tokens[i], tokens[i+1])
            pair_counts[pair] = pair_counts.get(pair, 0) + 1
    return pair_counts
```

This initial counting step has a complexity of $O(C)$, where C is the size of the corpus.

2. Efficiently Finding the Best Pair

With the `pair_counts` map, finding the most frequent pair is a simple iteration over the map's values. We don't need a priority queue for this, as we can just find the max in each step.

3. Efficiently Updating Counts After a Merge

This is the most complex part. When we merge a pair, say `(A, B)` into a new token `Z`, we cannot afford to rescan the corpus to update counts. The merge only affects the local context. Consider a sequence `... X, A, B, Y ...`. - The pair `(A, B)` is eliminated. - The pairs `(X, A)` and `(B, Y)` are destroyed. - New pairs `(X, Z)` and `(Z, Y)` are created.

To perform this update efficiently, we need to know where all occurrences of `(A, B)` are. A naive search is too slow. The efficient solution is to maintain an **inverted index**: a map from each pair to a list of locations (e.g., word index and position within the word) where it occurs.

However, maintaining a full inverted index is complex. A more practical and widely used approach (e.g., in `huggingface/tokenizers`) is to represent each word as a **doubly linked list** of tokens. This allows for $O(1)$ modification of the sequence during a merge.

The refined algorithm looks like this:

1. **Initialization:**

    - Represent the corpus as a list of "words," where each word is a doubly linked list of its initial tokens (bytes).
    - Perform one pass to compute initial `pair_counts`.

2. **Merge Loop:** For `i` from 1 to `num_merges`: a. **Find Best Pair:** Find the pair `(A, B)` with the highest count in `pair_counts`. b. **Update Corpus and Counts:** Iterate through each word in the corpus. Within each word (linked list), iterate through its nodes. - If a node and its successor form the pair `(A, B)`: i. Let the preceding node be `X` and the succeeding node be `Y`. ii. **Decrement old counts:** - `pair_counts[(X.value, A)] -= 1` - `pair_counts[(B, Y.value)] -= 1` iii. **Merge in the linked list:** Change the current node's value to the new token `Z` and delete the next node (which held `B`). This is an $O(1)$ pointer operation. iv. **Increment new counts:** - `pair_counts[(X.value, Z)] += 1` - `pair_counts[(Z, Y.value)] += 1` c. **Store Merge:** Add the merge rule `(A, B) -> Z` to our list of learned merges.

Complexity Analysis:

- **Initial Count:** $O(C)$
- **Merge Loop:** The loop runs $V - V_0$ times, where V is the final vocabulary size and V_0 is the initial one (256).
- Inside the loop, finding the best pair takes $O(P_i)$, where P_i is the number of unique pairs at step i.
- The crucial update step still requires iterating through the locations of the pair being merged. While we don't rescan the *entire* corpus, we do have to find all occurrences of the pair `(A, B)`. The linked list structure makes the *local modification* $O(1)$, but we still need to find where to perform the modification.

The key insight is that we can update the `pair_counts` dictionary directly without re-scanning the raw text. The cost of each merge step is proportional to the number of occurrences of the pair being merged, which is much smaller than the total corpus size. This makes the overall process tractable, though still computationally intensive for large vocabularies and corpora. The final complexity is much better than the naive $O(C \times V)$, closer to $O(C + \sum_{i=1}^{V-V_0} \text{occurrences}(\text{pair}_i) \log P_i)$.

> **Q: The BPE encoding process involves applying a learned list of merges greedily. What happens if a string contains multiple, overlapping candidate pairs? For example,**

> **if our vocabulary has merges for `('a', 'b')` -> `'ab'` and `('b', 'c')` -> `'bc'`, how is the string `'abc'` tokenized?**

This is an excellent question that probes a subtle but critical aspect of the BPE algorithm: ensuring deterministic encoding. If the string `'abc'` could be tokenized as either `['ab', 'c']` or `['a', 'bc']`, the tokenizer would be ambiguous and unusable.

The standard BPE encoding algorithm resolves this ambiguity with a simple, deterministic, greedy procedure. The key is that merges are not chosen based on what's possible, but are ranked by the order in which they were learned during training.

The Algorithm for Deterministic Encoding:

1. **Initialization:** Convert the input string into an initial sequence of tokens (e.g., UTF-8 bytes). `'abc'` -> `[97, 98, 99]`
2. **Iterative Merging:** Repeatedly scan the *current* token sequence and perform the *single best possible merge*. The "best" merge is defined as the one whose pair has the **lowest rank** (i.e., was learned earliest during training). If there's a tie in rank (which shouldn't happen with a proper merge list) or multiple occurrences of the same highest-priority pair, the **leftmost occurrence** is chosen.

Let's trace the example. Assume our training process learned the following merges in order: - ... - Merge #50: `(97, 98)` -> `256` (representing `'ab'`) - ... - Merge #100: `(98, 99)` -> `257` (representing `'bc'`) - ...

Here, the pair `(97, 98)` has a lower rank (50) than `(98, 99)` (100).

Encoding `'abc'`:

- **Initial tokens:** `[97, 98, 99]`
- **Iteration 1:**
  - Scan the sequence for possible merges. We find two: `(97, 98)` at index 0 (rank 50) and `(98, 99)` at index 1 (rank 100).
  - The best pair is `(97, 98)` because it has the lower rank (50 < 100).
  - Perform this merge: replace `[97, 98]` with the new token `256`.
  - The sequence becomes: `[256, 99]`.

- **Iteration 2:**
    - Scan the new sequence `[256, 99]`.
    - The only adjacent pair is `(256, 99)`. Let's assume this pair was never learned as a merge during training.
    - Since no more merges from our learned rules can be applied, the process terminates.
- **Final Tokenization:** `[256, 99]`

The result is unambiguous. The greedy, rank-based, left-to-right priority ensures that for any given string and set of merge rules, there is only one possible output.

Pseudocode for Encoding:

```
def get_pairs(tokens):
    """Helper to get all adjacent pairs in a sequence."""
    return set(zip(tokens, tokens[1:]))

def encode_bpe(text, merge_ranks):
    """
    Encodes text using BPE.
    merge_ranks: A dict mapping (tok1, tok2) -> rank (int, lower is better)
    """
    if not text:
        return []

    tokens = list(text.encode('utf-8'))

    while True:
        pairs = get_pairs(tokens)
        if not pairs:
            break

        # Find the best pair to merge based on rank
        best_pair = min(pairs, key=lambda p: merge_ranks.get(p, float('inf')))

        # If no mergeable pairs are found, we're done
        if best_pair not in merge_ranks:
            break

        # Find the first occurrence of the best pair and merge it
        first_idx = -1
        for i in range(len(tokens) - 1):
            if (tokens[i], tokens[i+1]) == best_pair:
                first_idx = i
                break

        # Perform the merge
        new_token_id = merge_ranks[best_pair] # In a real implementation, you'd map pair to new
```

```
            tokens = tokens[:first_idx] + [new_token_id] + tokens[first_idx+2:]

        return tokens
```

Common Pitfall: A frequent misconception is to iterate through the list of merge rules and apply each one to the entire string. This is incorrect and would lead to ambiguity. The correct method is to repeatedly find the single highest-priority merge *in the current state of the token sequence* and apply only that one before re-evaluating.

> **Q: The lecture notes mention that GPT-2's BPE doesn't run on the raw byte stream of the entire corpus, but first splits text into "words" and runs BPE *within* each word. Why add this extra pre-tokenization step, and what are its algorithmic and practical consequences?**

This design choice in the original GPT-2 tokenizer is a crucial heuristic that significantly shapes the nature of the resulting vocabulary and the tokenizer's behavior. It represents a trade-off between a "pure" statistical approach and a linguistically-motivated one.

Motivation: Preventing "Meaningless" Cross-Word Merges

The primary reason for this pre-tokenization step is to preserve the integrity of words as fundamental semantic units. A pure byte-level BPE, operating on a continuous stream of text, would frequently find statistically common pairs that cross word boundaries.

For example, in the text "this is a sentence.", the pair `('s', ' ')` (from "this is") might be very common. Merging this would create a token `s` that combines the end of one word with a space. While statistically valid, this token is semantically messy. It conflates morphology (the plural 's') with syntax (word separation).

By first splitting the text into "word-like" chunks (e.g., using a regular expression that splits on whitespace and punctuation) and running BPE *only within these chunks*, the algorithm is constrained. It can learn to merge `t` and `h` to `th`, and `th` and `e` to `the`, but it can never merge the `e` from `the` with the space that follows it.

Algorithmic and Practical Consequences:

This design has profound consequences, creating both benefits and drawbacks.

1. Pro: A "Cleaner," More Interpretable Vocabulary The resulting vocabulary is more intuitive. It consists of: - Complete common words (e.g., `" the"`, `" and"`, `" in"`). - Common sub-words and morphemes (e.g., `"ing"`, `"ation"`, `" pre"`). This arguably creates a better inductive bias for the language model, as the tokens it operates on correspond more closely to semantic building blocks.

2. Con: The "Leading Space" Problem This is the most significant drawback. Because pre-tokenization splits on whitespace, a word at the beginning of a sentence is treated as a different initial chunk from the same word in the middle of a sentence. - `"Tokenization is tricky."` -> `["Tokenization", " is", " tricky", "."]` - `" is"` and `"is"` are different strings and will be processed independently by BPE.

This means `" world"` and `"world"` will likely have different token IDs, effectively doubling the vocabulary required for many words. This is inefficient and can be a source of subtle bugs when manipulating text for a model. For example, `model.generate("Hello" + " world")` might produce a different result from `model.generate("Hello world")`.

3. Con: Dependence on Heuristic Regex The entire tokenization process becomes dependent on the specific regular expression used for the initial split. The GPT-2 regex is famously complex:

```
's|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+
```

This pattern hard-codes rules about English contractions and how to group letters, numbers, and symbols. This introduces a strong, language-specific bias and makes the tokenizer less universal. A different regex would produce a completely different vocabulary and tokenization scheme.

4. The Modern Approach: A More Principled Handling of Whitespace Recognizing these issues, many modern tokenizers like **SentencePiece** (used in Llama, T5) and the `huggingface/tokenizers` library take a more elegant approach. Instead of using pre-tokenization to prevent cross-word merges, they treat whitespace as a normal part of the sequence.

- **The Meta-Character Trick:** Whitespace is often replaced with a special meta-character, like ▁ (U+2581), before BPE training. The string `"Hello world"` becomes `"Hello▁world"`.
- Now, BPE can operate on the full sequence. It might learn to merge `" w"`, `"o"`, `"r"`, `"l"`, `"d"` into a single `world` token. It is free to merge across the ▁ character if it's statistically frequent, but the space information is preserved within the token itself.

This avoids the "leading space" problem (a word is always preceded by a ` ` or is at the start of the sequence) and eliminates the need for a complex, heuristic pre-tokenization regex, resulting in a more robust and consistent system.

Transformer Architecture Choices

> **Q: The lecture claims RoPE makes the query-key inner product depend only on relative position. How is this property, $\langle f(q_m, m), f(k_n, n) \rangle = g(q_m, k_n, m-n)$, mathematically derived?**

This is the core insight of Rotary Position Embeddings (RoPE). The derivation relies on the properties of 2D rotations. Let's build it up from first principles.

1. The Goal in 2D

Imagine our query and key vectors, q and k, are just 2D vectors (or complex numbers). We want to apply a transformation f that depends on a vector's absolute position, say m, such that the inner product of two transformed vectors at positions m and n only depends on their relative position, $m - n$.

A rotation is a perfect candidate. The inner product between two vectors is preserved under a common rotation. What if we rotate them by different amounts? Let q_m and k_n be vectors at positions m and n. We apply rotations R_m and R_n respectively. The new inner product is:

$$\langle R_m q_m, R_n k_n \rangle$$

Using the property that for any rotation matrix R, $\langle Rx, Ry \rangle = \langle x, y \rangle$, we can write:

$$\langle R_m q_m, R_n k_n \rangle = \langle q_m, R_m^T R_n k_n \rangle = \langle q_m, R_{-m} R_n k_n \rangle = \langle q_m, R_{n-m} k_n \rangle$$

This shows the inner product now depends on the original vectors and a rotation by their relative position, $n - m$. This is exactly the property we want.

2. The Math using Complex Numbers

The derivation is cleaner with complex numbers. Let a 2D vector (x_1, x_2) be represented as $x = x_1 + ix_2$. A rotation by an angle θ is equivalent to multiplication by $e^{i\theta}$.

Let's define our position-encoding function f as rotating a vector x at position m by an angle $m\theta$:

$$f(x, m) = x \cdot e^{im\theta}$$

Now, let's compute the inner product (real part of the conjugate product) between a query q at position m and a key k at position n:

$$\begin{aligned} \mathrm{Re}[f(q, m)\overline{f(k, n)}] &= \mathrm{Re}[(qe^{im\theta})\overline{(ke^{in\theta})}] \\ &= \mathrm{Re}[qe^{im\theta}\bar{k}e^{-in\theta}] \\ &= \mathrm{Re}[q\bar{k}e^{i(m-n)\theta}] \end{aligned}$$

This result elegantly shows that the interaction between the query and key depends on their original values ($q\bar{k}$) and is modulated by a complex exponential that is a function of only their relative distance, $m - n$.

3. Generalizing to High Dimensions

A Transformer's query and key vectors live in a high-dimensional space, $\mathbb{R}^{d_{head}}$. RoPE applies the 2D rotation idea by splitting the d_{head} dimensions into $d_{head}/2$ pairs. Each pair is treated as a 2D vector (or a complex number) and is rotated independently.

Let a vector $x \in \mathbb{R}^{d_{head}}$ be partitioned as $x = [x_0, x_1, x_2, x_3, \ldots, x_{d-2}, x_{d-1}]$. We form pairs $(x_0, x_1), (x_2, x_3), \ldots$.

To allow the model to distinguish between different relative distances, each pair is rotated by a different frequency. The rotation angle for the j-th pair at position m is $m\theta_j$, where $\theta_j = 10000^{-2j/d_{head}}$. This creates a spectrum of rotational speeds, from high-frequency (for nearby tokens) to low-frequency (for distant tokens).

The rotation of a vector x at position m can be written as a block-diagonal matrix multiplication, $f(x, m) = R_{m,d}x$, where:

$$R_{m,d} = \begin{pmatrix} \cos(m\theta_0) & -\sin(m\theta_0) & 0 & 0 & \ldots \\ \sin(m\theta_0) & \cos(m\theta_0) & 0 & 0 & \ldots \\ 0 & 0 & \cos(m\theta_1) & -\sin(m\theta_1) & \ldots \\ 0 & 0 & \sin(m\theta_1) & \cos(m\theta_1) & \ldots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

The inner product between a query q at position m and a key k at position n becomes:

$$\langle R_{m,d}q, R_{n,d}k\rangle = \langle q, R_{m,d}^T R_{n,d}k\rangle = \langle q, R_{n-m,d}k\rangle$$

This confirms the relative property holds in high dimensions.

Implementation (Pseudocode)

In practice, we don't form the matrix. We compute the sines and cosines and apply them directly.

```
def apply_rope(x, pos_ids):
    # x: [batch, seq_len, num_heads, head_dim]
    # pos_ids: [batch, seq_len]
    head_dim = x.shape[-1]

    # Create the theta frequencies
    # theta_j = 10000^(-2j/d) for j in [0, 1, ..., d/2 - 1]
    thetas = 1.0 / (10000.0 ** (torch.arange(0, head_dim, 2) / head_dim))

    # Create the m*theta angles for each position
    # angles: [seq_len, head_dim/2]
    angles = pos_ids.unsqueeze(-1) * thetas

    # Compute cosines and sines
    # cos_m, sin_m: [seq_len, head_dim/2]
    cos_m = torch.cos(angles)
    sin_m = torch.sin(angles)

    # Repeat cos and sin to match the full head_dim
    # cos, sin: [seq_len, head_dim]
    cos = torch.repeat_interleave(cos_m, 2, dim=-1)
    sin = torch.repeat_interleave(sin_m, 2, dim=-1)

    # Reshape x to apply rotation to pairs of dimensions
    # x_paired: [..., head_dim/2, 2]
    x_paired = x.reshape(*x.shape[:-1], -1, 2)

    # Rotate the pairs
    x_rotated = torch.stack(
        [
            x_paired[..., 0] * cos - x_paired[..., 1] * sin,
            x_paired[..., 1] * cos + x_paired[..., 0] * sin,
        ],
        dim=-1
    ).reshape_as(x)

    return x_rotated

# In attention block:
# q_proj = self.q_proj(x)
# k_proj = self.k_proj(x)
```

```
# q_rotated = apply_rope(q_proj, positions)
# k_rotated = apply_rope(k_proj, positions)
# attention_scores = q_rotated @ k_rotated.transpose(-1, -2)
```

This direct application is far more efficient and achieves the desired relative positional encoding within the attention mechanism itself.

> **Q: Why is dropping mean-centering from LayerNorm to get RMSNorm acceptable, and can you quantify the performance difference?**

The lecture notes correctly state that RMSNorm is faster due to reduced memory movement, not FLOPs. Let's analyze this trade-off and the algorithmic implications.

1. Performance Analysis: Memory is the Bottleneck

First, let's write down the operations for a vector $x \in \mathbb{R}^D$:

- **LayerNorm:** $y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \cdot \gamma + \beta$
- **RMSNorm:** $y = \frac{x}{\sqrt{\frac{1}{D}\sum_i x_i^2 + \epsilon}} \cdot \gamma$

A naive implementation of LayerNorm requires two passes over the input tensor `x`: one to compute the mean $\mathbb{E}[x]$, and a second to compute the variance and normalize. RMSNorm only requires one pass to compute the root mean square.

```
# Simplified pseudocode for one vector
# Naive LayerNorm (2 passes over x)
mean = x.sum() / D
# Pass 1 over x is done
var_sum = 0
for val in x:
    var_sum += (val - mean)**2
variance = var_sum / D
# Pass 2 over x is done
x_norm = (x - mean) / sqrt(variance + eps)
output = gamma * x_norm + beta

# RMSNorm (1 pass over x)
rms_sum = 0
for val in x:
    rms_sum += val**2
rms = sqrt(rms_sum / D + eps)
# Pass 1 over x is done
```

```
    x_norm = x / rms
    output = gamma * x_norm
```

On modern hardware like GPUs, operations like normalization are **memory-bound**. The time taken is dominated by reading data from high-bandwidth memory (HBM) into the much faster on-chip SRAM, not by the arithmetic computations (FLOPs).

- **Memory Traffic (LayerNorm):** A naive implementation reads `x` twice. A highly optimized, fused kernel (e.g., in Triton or cuDNN) can compute the mean and variance in a single pass, but it's more complex. It must read `x` (D elements), `gamma` (D elements), and `beta` (D elements), and write the output (D elements). Total memory I/O is at least $4D$ elements.
- **Memory Traffic (RMSNorm):** A fused kernel reads `x` (D elements) and `gamma` (D elements), and writes the output (D elements). Total memory I/O is $3D$ elements (no `beta`).

This reduction in memory traffic and kernel complexity is the primary source of the speedup. While normalization is a tiny fraction of total model FLOPs (~0.2%), it can be a significant portion of the non-GEMM (General Matrix Multiply) runtime. The lecture's claim that it can be 25% of runtime highlights how impactful memory-bound operations can be. The 3-7% end-to-end speedup reported in the original RMSNorm paper is a direct result of this efficiency gain.

2. Algorithmic Impact: Why is Dropping the Mean Okay?

Losing mean-centering means LayerNorm's "re-centering invariance" ($LayerNorm(x + c) = LayerNorm(x)$ for a scalar shift c) is lost. Why is this acceptable?

1. **Primary Goal is Scale Control:** The most critical function of normalization in deep networks is to control the *magnitude* of activation vectors. This prevents values from exploding or vanishing, which stabilizes training by keeping gradients in a reasonable range. RMSNorm achieves this by scaling the L2-norm (or RMS value) of the activation vector to a target value determined by the learnable gain γ. This variance control is the dominant factor for stability.

2. **Implicit Bias Adaptation:** Subsequent linear layers can adapt to a non-zero mean. A standard linear layer is $y = xW + b$. If its input x has a consistent non-zero mean, the network can learn to adjust the bias term b to compensate. While many modern Transformers drop explicit bias terms, the FFN and attention projections are still capable of learning transformations that are robust to input shifts.

3. **Pre-Norm Architecture:** In a pre-norm Transformer, normalization is applied *before* the main weight matrices of the attention and FFN blocks.

$$x_{l+1} = x_l + \text{Attention}(\text{Norm}(x_l))$$

The key is that the *input* to the attention and FFN blocks has a controlled magnitude. This ensures that the matrix multiplications within these blocks do not produce wildly large outputs, which in turn stabilizes the gradients with respect to the weights. RMSNorm is sufficient for this purpose.

4. **Empirical Success:** Ultimately, the strongest argument is empirical. Models like LLaMA, PaLM, and Chinchilla use RMSNorm and achieve state-of-the-art performance. This demonstrates that for the Transformer architecture, the benefits of LayerNorm's mean-centering are not critical and are outweighed by the computational efficiency of RMSNorm.

> **Q: The lecture notes that SwiGLU FFNs use a $d_{ff} = \frac{8}{3}d_{model}$ ratio. Can you derive this and explain the intuition behind why input-dependent gating is more effective than a static activation like ReLU?**

1. Derivation of the $\frac{8}{3}$ Ratio

The goal is to match the parameter count of a SwiGLU-based Feed-Forward Network (FFN) with that of a standard ReLU-based FFN, which typically uses a 4x expansion factor. Let's count the parameters in the weight matrices (ignoring biases, as is common).

- **Standard ReLU FFN:** The formula is $FFN(x) = \text{ReLU}(xW_1)W_2$.
  - $x \in \mathbb{R}^{d_{model}}$
  - $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}$
  - $W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$ The total number of parameters is the sum of the sizes of W_1 and W_2:

$$\text{Params}_{\text{ReLU}} = (d_{model} \cdot d_{ff}) + (d_{ff} \cdot d_{model}) = 2 \cdot d_{model} \cdot d_{ff}$$

Using the standard rule of thumb $d_{ff} = 4d_{model}$:

$$\text{Params}_{\text{ReLU}} = 2 \cdot d_{model} \cdot (4d_{model}) = 8 \cdot d_{model}^2$$

- **SwiGLU FFN:** The formula is $FFN_{SwiGLU}(x) = (\text{Swish}(xW_1) \otimes xV)W_2$, where $\otimes$ is element-wise multiplication. This variant requires three weight matrices. Let the intermediate dimension be d'_{ff}.
    - $x \in \mathbb{R}^{d_{model}}$
    - $W_1 \in \mathbb{R}^{d_{model} \times d'_{ff}}$ (for the Swish path)
    - $V \in \mathbb{R}^{d_{model} \times d'_{ff}}$ (for the gate path)
    - $W_2 \in \mathbb{R}^{d'_{ff} \times d_{model}}$ (the down-projection) The total number of parameters is the sum of the sizes of W_1, V, and W_2:

    $$\text{Params}_{\text{SwiGLU}} = (d_{model} \cdot d'_{ff}) + (d_{model} \cdot d'_{ff}) + (d'_{ff} \cdot d_{model}) = 3 \cdot d_{model} \cdot d'_{ff}$$

- **Equating Parameters:** To have a similar parameter budget, we set $\text{Params}_{\text{ReLU}} \approx \text{Params}_{\text{SwiGLU}}$:

    $$8 \cdot d_{model}^2 = 3 \cdot d_{model} \cdot d'_{ff}$$

    Solving for the SwiGLU intermediate dimension d'_{ff}:

    $$d'_{ff} = \frac{8}{3} d_{model} \approx 2.66 \cdot d_{model}$$

    This derivation confirms the $\frac{8}{3}$ ratio used by many modern models like LLaMA to balance the parameter count when adopting a Gated Linear Unit (GLU) structure.

2. Intuition for Gating

The superiority of SwiGLU over ReLU stems from its **dynamic, input-dependent nature**.

- **ReLU is Static:** The ReLU activation, $\max(0, z)$, is a fixed, non-linear function. A neuron is either "off" (output 0) or "on" (output z). The decision for this on/off state is based solely on the sign of its input, $z = (xW_1)_i$. It's a hard switch.

- **SwiGLU is Dynamic:** The SwiGLU activation has two components: the main data path $\text{Swish}(xW_1)$ and a gating path xV. The final activation passed to W_2 is the element-wise product of these two.

    $$\text{Activation}_i = \text{Swish}((xW_1)_i) \cdot (xV)_i$$

    This structure can be interpreted as an **input-dependent filter**.

- The term xW_1 projects the input into a higher-dimensional space, proposing a set of potential features.
- The term xV acts as a **gate**. It's another projection of the *same input* x that learns to control *how much* of each proposed feature from the first path should be allowed to pass through.
- If a neuron in the gate path $(xV)_i$ outputs a value near 0, it effectively "shuts off" the corresponding feature from the data path, regardless of its value. If it outputs a large value, it amplifies that feature.

This mechanism allows the FFN to be much more expressive. Instead of a simple on/off decision, the network can make a nuanced, continuous choice about which neurons to fire and by how much, based on the entirety of the input token's representation. This is a form of rapid adaptation or soft attention *within* the FFN layer itself, enabling the model to select and combine features more flexibly for each specific token, which has been shown empirically to lead to better performance.

> **Q: Can you provide a detailed analysis of the KV cache size and memory bandwidth requirements for MHA, MQA, and GQA during autoregressive decoding?**

Yes, this analysis is crucial for understanding why MQA and GQA are so important for efficient inference, especially with long contexts.

Setup: * B: Batch size * L: Number of layers * N: Current sequence length (context length) * d: Model dimension * h: Number of query heads * d_k: Dimension of key/value vectors per head (typically $d_k = d/h$) * g: Number of key/value groups in GQA ($1 \le g \le h$) * `sizeof(fp16)`: 2 bytes (a common data type for the cache)

The KV cache stores the key and value vectors for all N tokens in the context, for every layer. When generating the $(N+1)$-th token, the model's query for this new token must attend to all N keys and values stored in the cache.

1. Multi-Head Attention (MHA) In MHA, every query head has its own unique key and value head. So, $g = h$.

- **KV Cache Size:** For each layer, we store N keys and N values for all h heads.
  - Size per layer = (keys + values) = $(B \cdot N \cdot h \cdot d_k) + (B \cdot N \cdot h \cdot d_k)$
  - Since $h \cdot d_k = d$, this simplifies to: $2 \cdot B \cdot N \cdot d$ elements.

- **Total Cache Size (Bytes) =** $L \cdot 2 \cdot B \cdot N \cdot d \cdot \text{sizeof}(\text{fp16})$

- **Example (LLaMA-7B):** $L = 32, d = 4096, h = 32$. For a single sequence ($B = 1$) with a context of $N = 4096$:
  - Total Size = $32 \cdot 2 \cdot 1 \cdot 4096 \cdot 4096 \cdot 2 \text{ bytes} \approx 2.15 \text{ GB}$. This is a huge amount of memory just for one sequence.

2. Multi-Query Attention (MQA) In MQA, all h query heads share a single key and value head. This is the special case where $g = 1$.

- **KV Cache Size:** For each layer, we only store one set of keys and values.
  - Size per layer = $(B \cdot N \cdot 1 \cdot d_k) + (B \cdot N \cdot 1 \cdot d_k) = 2 \cdot B \cdot N \cdot d_k$
  - **Total Cache Size (Bytes) =**
    $L \cdot 2 \cdot B \cdot N \cdot d_k \cdot \text{sizeof}(\text{fp16}) = L \cdot 2 \cdot B \cdot N \cdot (d/h) \cdot \text{sizeof}(\text{fp16})$
- **Comparison:** The MQA cache is $1/h$ the size of the MHA cache.
- **Example (LLaMA-7B params with MQA):** With $h = 32$, the cache size would be $2.15 \text{ GB}/32 \approx 67 \text{ MB}$. This is a dramatic reduction.

3. Grouped-Query Attention (GQA) GQA is the general case. The h query heads are split into g groups, with each group sharing a KV head.

- **KV Cache Size:** For each layer, we store g sets of keys and values.
  - Size per layer = $(B \cdot N \cdot g \cdot d_k) + (B \cdot N \cdot g \cdot d_k) = 2 \cdot B \cdot N \cdot g \cdot d_k$
  - **Total Cache Size (Bytes) =**
    $L \cdot 2 \cdot B \cdot N \cdot g \cdot d_k \cdot \text{sizeof}(\text{fp16}) = L \cdot 2 \cdot B \cdot N \cdot g \cdot (d/h) \cdot \text{sizeof}(\text{fp16})$
- **Comparison:** The GQA cache is g/h the size of the MHA cache. This provides a tunable knob between model quality (higher g) and inference efficiency (lower g). For example, Mistral-7B uses GQA with $h = 32$ and $g = 8$, resulting in a cache size that is $8/32 = 1/4$ that of an equivalent MHA model.

Memory Bandwidth Impact

During autoregressive decoding, the generation of each new token is a separate step. The main bottleneck is not the FLOPs, but the time it takes to load the entire KV cache from the GPU's slow HBM into its fast on-chip SRAM for the attention computation.

- **Bytes Read per Token (per layer):** The amount of data that must be read from HBM is proportional to the cache size for the keys and values.
    - **MHA:** Reads $O(B \cdot N \cdot d)$ bytes for K/V.
    - **MQA:** Reads $O(B \cdot N \cdot d/h)$ bytes for K/V.
    - **GQA:** Reads $O(B \cdot N \cdot g \cdot d/h)$ bytes for K/V.

Conclusion: By drastically reducing the size of the KV cache, MQA and GQA directly reduce the memory bandwidth required at each generation step. This is the primary reason for their significant improvement in inference throughput (tokens per second), especially for models with many heads and long context lengths. The small degradation in model quality is often a very acceptable price for the massive gains in serving efficiency.

Mixture of Experts: routing & load balancing

> **Q: The notes say Top-K routing is "non-differentiable," so how are the router's parameters trained at all? What gradient signal does the router receive?**

This is a crucial point. The `TopK` operation, which selects the indices of the K largest values from a vector of scores, is a discrete selection. Its gradient is zero almost everywhere, and undefined where ties occur. A naive implementation would block gradients from flowing back to the router parameters for any expert that wasn't selected, providing no signal on how to improve.

The solution is to use a **Straight-Through Estimator (STE)**. In the forward pass, we perform the hard, non-differentiable `TopK` selection. In the backward pass, we "pretend" this selection was a simple multiplication by a 0/1 mask and allow gradients to pass through the selected "gates" as if they were always open.

Let's break down the math.

1. Forward Pass:

For a single token with hidden state $x \in \mathbb{R}^H$, the router computes scores for N experts. The router has a weight matrix $W_g \in \mathbb{R}^{H \times N}$.

1. **Logits:** Compute raw scores (logits) for each expert.

$$z = xW_g \quad (z \in \mathbb{R}^N)$$

2. **Probabilities:** Apply softmax to get normalized probabilities.

$$p = \text{Softmax}(z) \quad (p \in \mathbb{R}^N, \sum p_i = 1)$$

3. **Gating Mask:** Identify the top K probabilities and create a sparse binary mask.

$$M = \text{TopK\_Mask}(p, K) \quad (M \in \{0, 1\}^N, \sum M_i = K)$$

4. **Gating Weights:** The final weight for each expert is its probability, but only if it was selected. This is the STE part: we use the "soft" probability value p_i for the "hard" choice $M_i = 1$.

$$g_i = M_i \cdot p_i$$

5. **Final Output:** The token's output is a weighted sum of the outputs from the selected experts.

$$y = \sum_{i=1}^{N} g_i \cdot \text{Expert}_i(x) = \sum_{i \in \text{TopK}} p_i \cdot \text{Expert}_i(x)$$

```
# Simplified forward pass for one token
def forward_moe_layer(x, experts, router_weights, K):
    # x: [H], router_weights: [H, N]
    logits = x @ router_weights  # [N]
    probs = softmax(logits)      # [N]

    # Non-differentiable step
    top_k_probs, top_k_indices = torch.topk(probs, K)

    # Create a mask for the backward pass (for clarity)
    mask = torch.zeros_like(probs)
    mask.scatter_(0, top_k_indices, 1) # mask is [0, 1, 0, 0, 1, ...] for K=2

    # Gating weights (STE: use soft probs for hard choices)
    gating_weights = probs * mask # This is where the "lie" happens for backprop
    # In a real implementation, you'd just use top_k_probs and top_k_indices
    # to avoid computing on all experts.

    # Compute expert outputs
    final_output = 0
    for i in top_k_indices:
        # The gradient will flow back through here
        expert_output = experts[i](x)
        # And through here, scaled by the probability
```

```
            final_output += probs[i] * expert_output

        return final_output
```

2. Backward Pass (The "Lie"):

Let L be the final model loss. We need to compute $\frac{\partial L}{\partial W_g}$. Using the chain rule, the gradient flows "through" the selected experts. For a single expert i that was in the Top-K set (i.e., $M_i = 1$):

$$\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z_i} = \frac{\partial L}{\partial y}\frac{\partial}{\partial z_i}(p_i \cdot \text{Expert}_i(x))$$

Assuming the expert's computation $\text{Expert}_i(x)$ does not depend on the router's logits z, this simplifies to:

$$\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial y} \cdot \text{Expert}_i(x) \cdot \frac{\partial p_i}{\partial z_i}$$

The crucial part is that for any expert j *not* in the Top-K set (i.e., $M_j = 0$), the gating weight g_j was 0, so its contribution to the output y was 0. Therefore, $\frac{\partial L}{\partial z_j} = 0$.

The Pitfall: The router receives **zero gradient signal** for the experts it did not select. It gets feedback on how to adjust probabilities *among the winners*, but it never learns that it should have picked a different expert entirely. This can lead to a "rich get richer" dynamic where a few experts are always chosen, while others "die" from lack of training signals. This is precisely why auxiliary load balancing losses are not just helpful, but essential for stable MoE training.

> **Q: How does the Switch Transformer's load balancing loss, $L_{aux} = \alpha \sum_{i=1}^{N} f_i P_i$, mathematically encourage experts to be used more evenly?**

This auxiliary loss is a clever heuristic that pushes the router away from imbalanced expert assignments. To understand how, we must derive its gradient with respect to the router's logits.

First, let's define the terms for a batch of T tokens and N experts: - $p_{i,t}$: The softmax probability from the router for expert i and token t. - $P_i = \frac{1}{T}\sum_{t=1}^{T} p_{i,t}$: The average router probability assigned to expert i over the batch. This is the router's *intention* to use expert i. - $c_{i,t} \in \{0, 1\}$: An indicator that is 1 if expert i was chosen for token t (i.e., was in the Top-K), and 0 otherwise. -

$f_i = \frac{1}{T} \sum_{t=1}^{T} c_{i,t}$: The fraction of tokens in the batch that were *actually dispatched* to expert i. This is the expert's *utilization*.

The auxiliary loss is:

$$L_{aux} = \alpha \cdot N \cdot \sum_{i=1}^{N} f_i P_i$$

(Note: The original paper includes a factor of N for scaling, so we'll include it here.)

The key trick is that during backpropagation for this specific loss term, the expert utilization f_i (which comes from the non-differentiable `TopK` choice) is treated as a **constant**. The gradient is only computed with respect to the differentiable probabilities P_i.

Let's find the gradient of L_{aux} with respect to a single router logit $z_{k,t}$ (for expert k, token t). We use the chain rule:

1. **Gradient w.r.t. average probability P_j**:

$$\frac{\partial L_{aux}}{\partial P_j} = \alpha N f_j$$

2. **Gradient of P_j w.r.t. single probability $p_{k,t}$**:

$$\frac{\partial P_j}{\partial p_{k,t}} = \frac{\partial}{\partial p_{k,t}} \left(\frac{1}{T} \sum_{t'=1}^{T} p_{j,t'} \right) = \frac{1}{T} \delta_{jk}$$

   where δ_{jk} is the Kronecker delta (1 if $j = k$, 0 otherwise).

3. **Gradient w.r.t. single probability $p_{k,t}$**:

$$\frac{\partial L_{aux}}{\partial p_{k,t}} = \sum_{j=1}^{N} \frac{\partial L_{aux}}{\partial P_j} \frac{\partial P_j}{\partial p_{k,t}} = \frac{\partial L_{aux}}{\partial P_k} \frac{\partial P_k}{\partial p_{k,t}} = (\alpha N f_k) \left(\frac{1}{T} \right) = \frac{\alpha N f_k}{T}$$

4. **Gradient w.r.t. logit $z_{k,t}$**: The gradient of a loss with respect to softmax logits z_k can be expressed in terms of the gradient with respect to the softmax outputs p_k:

$$\frac{\partial L}{\partial z_k} = \sum_{j=1}^{N} \frac{\partial L}{\partial p_j} \frac{\partial p_j}{\partial z_k} = \sum_{j=1}^{N} \frac{\partial L}{\partial p_j} p_j (\delta_{jk} - p_k) = p_k \left(\frac{\partial L}{\partial p_k} - \sum_{j=1}^{N} \frac{\partial L}{\partial p_j} p_j \right)$$

Substituting our gradient $\frac{\partial L_{aux}}{\partial p_{j,t}} = \frac{\alpha N f_j}{T}$:

$$\frac{\partial L_{aux}}{\partial z_{k,t}} = p_{k,t} \left(\frac{\alpha N f_k}{T} - \sum_{j=1}^{N} \frac{\alpha N f_j}{T} p_{j,t} \right) = \frac{\alpha N p_{k,t}}{T} \left(f_k - \sum_{j=1}^{N} f_j p_{j,t} \right)$$

Intuition: The gradient on logit $z_{k,t}$ is proportional to $f_k - \mathbb{E}_{p_t}[f]$. - $\boldsymbol{f_k}$ is the actual utilization of expert k. - $\sum_{j=1}^{N} \boldsymbol{f_j p_{j,t}}$ is the expected utilization for this specific token, weighted by the router's current probabilities.

If expert k is over-utilized (f_k is high), then $f_k > \mathbb{E}_{p_t}[f]$, making the gradient positive. During gradient *descent*, the optimizer will push the logit $z_{k,t}$ **down**. This makes the router less likely to assign high probability to the over-utilized expert k for this token in the future.

Conversely, if an expert is under-utilized (f_k is low), the gradient term becomes negative, and the optimizer pushes its logit $z_{k,t}$ **up**, encouraging the router to use it more. This simple, elegant mechanism nudges the router towards a state where all f_i are roughly equal, ensuring all experts are trained.

> **Q: What is the "Router Z-loss" and why does penalizing the log-sum-exp of logits improve training stability?**

The Router Z-loss is an auxiliary loss term designed to prevent the router's logits from growing too large, which is a common source of numerical instability, especially when training with low-precision formats like `bfloat16`.

The loss is defined for each token t as the squared log-sum-exp of its router logits z_t, scaled by a small coefficient β:

$$L_z = \beta \cdot \left(\log \sum_{i=1}^{N} e^{z_{i,t}} \right)^2$$

This is then averaged across all tokens in a batch.

Why does this help?

1. **Connection to Softmax and Overflow:** The softmax function, $p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$, is sensitive to large logit values. If any z_i becomes very large, e^{z_i} can exceed the maximum representable

value for a given floating-point type (e.g., `bfloat16` has a max value of ~3.4e38), causing an `inf` (infinity) result. This leads to `NaN` (Not a Number) in the softmax output and corrupts the gradients, causing the training to "blow up".

2. **Log-Sum-Exp as a Smooth Maximum:** The function $\mathrm{LSE}(z) = \log \sum_i e^{z_i}$ is a well-known smooth approximation of the maximum function, i.e., $\mathrm{LSE}(z) \approx \max_i(z_i)$. Therefore, the Z-loss is effectively penalizing $(\max_i z_{i,t})^2$. By adding this penalty to the total loss, the optimizer is incentivized to keep the magnitude of all logits, and especially the largest one, under control.

Mathematical Derivation of the Gradient:

To see how it works, let's derive the gradient of L_z with respect to a single logit z_k. Let $S = \sum_i e^{z_i}$. Then $L_z = \beta(\log S)^2$.

Using the chain rule:

$$\frac{\partial L_z}{\partial z_k} = \frac{\partial L_z}{\partial S}\frac{\partial S}{\partial z_k}$$

$$\frac{\partial S}{\partial z_k} = \frac{\partial}{\partial z_k}\left(\sum_i e^{z_i}\right) = e^{z_k}$$

$$\frac{\partial L_z}{\partial S} = \beta \cdot 2(\log S) \cdot \frac{1}{S}$$

Combining these:

$$\frac{\partial L_z}{\partial z_k} = \beta \cdot 2(\log S) \cdot \frac{1}{S} \cdot e^{z_k} = 2\beta\left(\log \sum_i e^{z_i}\right)\left(\frac{e^{z_k}}{\sum_i e^{z_i}}\right)$$

Recognizing the final term as the softmax probability p_k, we get:

$$\frac{\partial L_z}{\partial z_k} = 2\beta \cdot (\mathrm{LSE}(z)) \cdot p_k$$

Intuition: The gradient for logit z_k is always positive (since LSE and p_k are positive). During gradient descent, the optimizer will always push the logits **down**. The magnitude of this "push" is proportional to two things: 1. $\mathrm{LSE}(z)$: The current magnitude of the (smooth) max logit. If the

logits are already large, the push is stronger. 2. p_k: The router's confidence in expert k. The logit for the expert the router is already confident in gets pushed down the most.

This acts as a gentle regularizer, preventing any logit from growing without bound and keeping the router's internal values within a safe numerical range, thus ensuring stable training. It's a simple but highly effective trick for taming MoE models.

> **Q: How do "fine-grained experts" increase the parameter count for the same computational cost (FLOPs)? A quantitative analysis seems in order.**

This is a key insight behind modern MoE architectures like DeepSeek. The trick is to trade the *size* of each expert for the *number* of experts, which allows for a dramatic increase in the total parameter count while keeping the number of *activated* parameters (and thus FLOPs) constant.

Let's analyze this by comparing a dense FFN layer to a fine-grained MoE layer with the same FLOPs budget.

Definitions: - H: Model hidden dimension. - I: Intermediate dimension of a standard FFN (e.g., $I = 4H$). - T: Number of tokens in a sequence. - FLOPs are approximated as $2 \times \text{MACs}$ (Multiply-Accumulate operations).

1. Baseline: Dense FFN Layer A standard FFN consists of an up-projection ($W_{up} : H \to I$) and a down-projection ($W_{down} : I \to H$). - **Parameters:** The total number of weights is $(H \times I) + (I \times H) = 2HI$. - **FLOPs:** For each token, we perform two matrix-vector multiplications. The total FLOPs are $T \times (2 \cdot H \cdot I + 2 \cdot I \cdot H) \approx 4THI$.

2. Fine-Grained MoE Layer (at constant FLOPs) Now, we replace the dense FFN with an MoE layer. We want to keep the FLOPs the same as the dense baseline. - Let the MoE have N_{fg} "fine-grained" experts. - Each expert has a smaller intermediate dimension, I_{fg}. - The router activates the Top-K experts for each token.

- **FLOPs:** The computation for one token involves K experts. The FLOPs per token are $K \times (2HI_{fg} + 2HI_{fg}) = 4KHI_{fg}$. The total FLOPs are $T \times 4KHI_{fg}$.
- **Equating FLOPs:** To match the dense model's computational cost, we set:

$$4THI = 4TKHI_{fg} \implies I = K \cdot I_{fg}$$

This is the core constraint: the intermediate dimension of the dense model must equal the *sum* of the intermediate dimensions of the activated experts.

- **Parameters:** The total parameter count of the MoE layer is the sum of parameters across all N_{fg} experts (ignoring the small router).

$$\text{Params}_{MoE} = N_{fg} \times (2HI_{fg})$$

Substituting $I_{fg} = I/K$ from our FLOPs constraint:

$$\text{Params}_{MoE} = N_{fg} \times \left(2H\frac{I}{K}\right) = \frac{N_{fg}}{K} \cdot (2HI)$$

Recall that the dense model's parameters were $\text{Params}_{Dense} = 2HI$. Therefore:

$$\text{Params}_{MoE} = \frac{N_{fg}}{K} \cdot \text{Params}_{Dense}$$

The Payoff: For the same computational cost as a dense layer, an MoE layer has $\frac{N_{fg}}{K}$ times more parameters.

Concrete Example: Let's design an MoE to replace a dense FFN where $H = 4096$ and $I = 4H = 16384$. - **Dense FFN:** - FLOPs per token $\propto I = 16384$. - Parameters $\propto 2HI = 2 \cdot 4096 \cdot 16384 \approx 134M$.

- **Fine-Grained MoE:** Let's use $K = 2$ active experts (a common choice).
- To match FLOPs, we need $K \cdot I_{fg} = I \implies 2 \cdot I_{fg} = 16384 \implies I_{fg} = 8192$. Each expert is half the size of the original dense FFN.
- Now, let's say we use $N_{fg} = 8$ total experts (like Mixtral 8x7B).
- **Total Parameters:** $\text{Params}_{MoE} = \frac{N_{fg}}{K} \cdot \text{Params}_{Dense} = \frac{8}{2} \times 134M = 4 \times 134M = 536M$.

By replacing the dense layer with an 8-expert MoE (activating 2 per token), we have **quadrupled the parameter count** of that layer while keeping the inference FLOPs **identical**. This allows the model to have a much larger capacity for storing knowledge without increasing the computational cost of a forward pass.

GPU Kernels & FlashAttention

> **Q: The "online softmax" is key to FlashAttention, but how does it work without knowing the global max and sum for normalization? Derive the update rule that makes this possible and explain its numerical stability.**

A: This is a fantastic question that gets to the heart of FlashAttention's innovation. Standard softmax is a global operation, which is a major obstacle for tiling. The "online softmax" algorithm reframes the problem to be computed in a streaming or block-wise fashion.

1. The Problem with Naive and Standard Softmax

The softmax function for a vector $\boldsymbol{x} = [x_1, \ldots, x_N]$ is defined as:

$$\text{softmax}(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

The issue is numerical stability. If any x_i is large, e^{x_i} can overflow to `inf`. If all x_i are large and negative, e^{x_i} can underflow to `0`, making the denominator zero.

The standard solution is the "log-sum-exp" trick. We find the maximum value of the vector, $m = \max_j(x_j)$, and rewrite the formula:

$$\text{softmax}(\boldsymbol{x})_i = \frac{e^{x_i - m}}{\sum_{j=1}^{N} e^{x_j - m}}$$

This is numerically stable because the largest exponent is now 0 (since $e^{m-m} = e^0 = 1$), preventing overflow. At least one term in the denominator is 1, preventing underflow of the sum.

However, this still requires a full pass over the vector $\boldsymbol{x}$ to find the global maximum m *before* we can compute the numerators and the final sum. This is incompatible with a tiled approach where we only have a small block of $\boldsymbol{x}$ in fast memory at any given time.

2. Deriving the Online Update Rule

The online algorithm maintains running statistics. Let's say we have processed a block of inputs $\boldsymbol{x}^{(1)} = [x_1, \ldots, x_k]$ and computed their partial softmax statistics: - Running maximum: $m^{(1)} = \max(x_1, \ldots, x_k)$ - Running denominator: $l^{(1)} = \sum_{j=1}^{k} e^{x_j - m^{(1)}}$

Now, a new block of inputs $\boldsymbol{x}^{(2)} = [x_{k+1}, \ldots, x_N]$ arrives. We can compute its local statistics: - Local maximum: $m^{(2)} = \max(x_{k+1}, \ldots, x_N)$ - Local denominator: $l^{(2)} = \sum_{j=k+1}^{N} e^{x_j - m^{(2)}}$

The goal is to combine $(\boldsymbol{x}^{(1)}, m^{(1)}, l^{(1)})$ and $(\boldsymbol{x}^{(2)}, m^{(2)}, l^{(2)})$ to get the statistics for the full vector $\boldsymbol{x} = [\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}]$.

The new global maximum is simply:

$$m^{\text{new}} = \max(m^{(1)}, m^{(2)})$$

The new denominator, normalized by m^{new}, is:

$$l^{\text{new}} = \sum_{j=1}^{N} e^{x_j - m^{\text{new}}}$$

We can split this sum:

$$l^{\text{new}} = \sum_{j=1}^{k} e^{x_j - m^{\text{new}}} + \sum_{j=k+1}^{N} e^{x_j - m^{\text{new}}}$$

Now, let's re-introduce our old running statistics, $l^{(1)}$ and $l^{(2)}$, by factoring out the appropriate terms:

$$l^{\text{new}} = \sum_{j=1}^{k} e^{(x_j - m^{(1)}) + (m^{(1)} - m^{\text{new}})} + \sum_{j=k+1}^{N} e^{(x_j - m^{(2)}) + (m^{(2)} - m^{\text{new}})}$$

$$l^{\text{new}} = e^{m^{(1)} - m^{\text{new}}} \sum_{j=1}^{k} e^{x_j - m^{(1)}} + e^{m^{(2)} - m^{\text{new}}} \sum_{j=k+1}^{N} e^{x_j - m^{(2)}}$$

Substituting in $l^{(1)}$ and $l^{(2)}$ gives us the final update rule for the denominator:

$$l^{\text{new}} = e^{m^{(1)} - m^{\text{new}}} \cdot l^{(1)} + e^{m^{(2)} - m^{\text{new}}} \cdot l^{(2)}$$

3. The Algorithm in FlashAttention

In FlashAttention, this is applied iteratively. A thread block computes attention for a block of queries Q_i against a block of keys K_j.

```
# Pseudocode for one query row's online softmax
# Initialize running statistics
m_i = -float('inf') # running max
l_i = 0.0           # running sum of exps
O_i = zeros(d_v)    # running output vector
```

```
    # Outer loop over blocks of keys/values
    for j in range(num_key_blocks):
        # 1. Load block of keys K_j and values V_j from HBM to SRAM
        K_j, V_j = load_from_hbm(j)

        # 2. Compute attention scores for current block
        S_ij = Q_i @ K_j.T

        # 3. Find local max and compute local softmax components
        m_ij_new = max(S_ij)
        P_ij = exp(S_ij - m_ij_new)
        l_ij = sum(P_ij)

        # 4. Update global max and running statistics
        m_i_old = m_i
        m_i = max(m_i, m_ij_new)

        # Rescale old sum and old output vector to the new global max
        scale = exp(m_i_old - m_i)
        l_i = l_i * scale + l_ij
        O_i = O_i * scale

        # 5. Update output vector with current block's contribution
        O_i += P_ij @ V_j

    # Final normalization
    O_i = O_i / l_i
    # Write O_i back to HBM
```

The crucial insight is that we also need to update the running output vector O_i. If the global maximum m_i changes, the previous partial outputs were normalized with a stale maximum. The line `O_i = O_i * scale` corrects for this, effectively "down-weighting" the previous output to be consistent with the new, larger maximum. This ensures that when the final division by l_i happens, all components have been correctly scaled relative to the true global maximum.

This algorithm allows each block of the QK^T matrix to be computed, used, and then discarded, avoiding the $O(N^2)$ memory cost of storing the full matrix.

> **Q: The FlashAttention backward pass recomputes the attention matrix instead of storing it. Quantify the memory-compute trade-off of this decision and explain why it's a huge performance win on modern GPUs.**

A: This is a classic example of trading compute for memory bandwidth, a winning strategy on modern hardware where compute scales much faster than memory.

1. Standard Attention Backward Pass

Let's analyze the memory requirements for a standard backward pass. The attention output O is computed as $O = PV$, where $P = \text{softmax}(S)$ and $S = \frac{QK^T}{\sqrt{d_k}}$. To compute the gradients with respect to the inputs (dQ, dK, dV), the chain rule requires the intermediate attention matrix P. - $dV = P^T dO$ - $dP = dOV^T$ - $dS = \text{grad\_softmax}(P, dP)$ - $dQ = dSK/\sqrt{d_k}$ - $dK = dS^T Q/\sqrt{d_k}$

The key bottleneck is that to compute dP, dS, dQ, dK, we need to have the $N \times N$ attention matrix P (where N is sequence length) that was computed during the forward pass.

Memory Cost of Storing P: - The matrix P has dimensions $N \times N$. - Storing it in FP32 requires $4N^2$ bytes. - For a typical context length of $N = 4096$, this is $4 \times (4096)^2 \approx 67$ MB. - For $N = 16384$, this is $4 \times (16384)^2 \approx 1.07$ GB per attention head, per layer! This is clearly not scalable and quickly exhausts GPU HBM.

2. FlashAttention's Recomputation Strategy

FlashAttention avoids this $O(N^2)$ memory cost by not storing the full attention matrix P. Instead, during the backward pass, it recomputes the necessary blocks of P on-the-fly.

The Backward Pass Algorithm: 1. The gradients dO and the original inputs Q, K, V are read from HBM. 2. The backward pass iterates through blocks of Q, K, V in the same way as the forward pass. 3. For each block, it recomputes the corresponding block of the score matrix $S_{ij} = Q_i K_j^T$. 4. Using the recomputed S_{ij} and the saved online softmax statistics (m_i, l_i from the forward pass, which only cost $O(N)$ memory), it can exactly reconstruct the corresponding block of P_{ij}. 5. With this on-the-fly block of P_{ij} and the corresponding block of dO_i, it computes the local gradients dS_{ij}, and accumulates the global gradients dQ, dK, dV. 6. The recomputed block of P_{ij} is then discarded.

3. Quantifying the Trade-off

Let's analyze the costs for a sequence of length N and head dimension d_k.

Aspect	Standard Attention (Backward)	FlashAttention (Backward)
HBM Memory	$O(N^2)$ to store P.	$O(N)$ to store softmax stats.
HBM Reads	Read P once: $O(N^2)$	Read Q, K again: $O(Nd_k)$

Aspect	Standard Attention (Backward)	FlashAttention (Backward)
Compute (FLOPs)	Uses stored P.	Recomputes $S = QK^T$: $O(N^2 d_k)$

Analysis: - **Memory Savings:** FlashAttention reduces the HBM memory requirement for intermediate activations from $O(N^2)$ to $O(N)$. For $N = 4k, d_k = 64$, this is a reduction from ≈ 67MB to ≈ 32KB—a >2000x reduction. This is the primary win. - **Memory Bandwidth Trade-off:** Instead of one large $O(N^2)$ read of matrix P, FlashAttention performs $O(Nd_k)$ reads of Q and K. Since typically $d_k \ll N$ (e.g., $64 \ll 4096$), the amount of data read from HBM is significantly less: $O(Nd_k) \ll O(N^2)$. - **Compute Cost:** The price is re-doing the QK^T matrix multiplication, which adds $O(N^2 d_k)$ FLOPs.

Why is this a win? The Roofline model tells us that performance is limited by either compute throughput or memory bandwidth. - **Standard Attention:** The need to read the massive $N \times N$ matrix P from slow HBM makes the backward pass severely **memory-bound**. The GPU's powerful compute units spend most of their time idle, waiting for data. - **FlashAttention:** By recomputing, the algorithm trades a massive HBM read for extra on-chip computation. Since modern GPUs have a very high compute-to-memory-bandwidth ratio, this is an excellent trade. The additional FLOPs can be executed quickly by the SMs using data that is already in registers or SRAM. The algorithm becomes much more **compute-bound**, allowing it to run closer to the GPU's peak theoretical throughput.

In essence, FlashAttention's backward pass avoids the single slowest operation—transferring a gigantic matrix from HBM—at the cost of more of the fastest operation—on-chip arithmetic.

> **Q: A CUDA kernel maps each thread to a single data element using `threadIdx`, while a Triton kernel operates on a vector of `offsets`. Why does Triton's block-level, vectorized programming model often lead to high performance with less effort, especially regarding memory access?**

A: This question highlights a fundamental shift in the programming abstraction for GPUs. While both CUDA C++ and Triton compile down to similar machine code (PTX), Triton's model is designed to automate common but tricky GPU optimizations, particularly memory coalescing.

1. The CUDA C++ Model: Thread-centric

In a typical CUDA C++ kernel for an element-wise operation, the programmer writes code from the perspective of a *single thread*.

```
// CUDA C++: A single thread's perspective
__global__ void add_kernel(float* x, float* y, float* out, int N) {
    // Each thread computes its own global index
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Boundary check
    if (i < N) {
        // Each thread performs one operation on one element
        out[i] = x[i] + y[i];
    }
}
```

- **Responsibility:** The programmer is responsible for the index arithmetic (`blockIdx.x * ...`). This logic maps the 3D hierarchy of grids, blocks, and threads to a linear index in the tensor.
- **Memory Access:** Each thread issues its own load (`x[i]` , `y[i]`) and store (`out[i]`). For the hardware to perform a *coalesced read* (efficiently loading a contiguous chunk of memory for a whole warp of 32 threads), the programmer must ensure that `threadIdx.x` , `threadIdx.x+1` , ..., `threadIdx.x+31` map to contiguous memory addresses. While simple for this `add_kernel` , it becomes complex for strided, transposed, or tiled access patterns. The compiler has limited visibility; it only sees that a single thread `i` is accessing `x[i]` .

2. The Triton Model: Block-centric and Vectorized

Triton elevates the abstraction. The programmer writes code from the perspective of a *program* (a thread block) that operates on *vectors* of data.

```
# Triton: A block's perspective
@triton.jit
def add_kernel(x_ptr, y_ptr, out_ptr, N, BLOCK_SIZE: tl.constexpr):
    # This program gets a unique ID (pid)
    pid = tl.program_id(axis=0)

    # This program is responsible for a whole block of offsets
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)

    # Boundary check on the vector of offsets
    mask = offsets < N
```

```
    # Load, compute, and store entire vectors of data
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    out = x + y
    tl.store(out_ptr + offsets, out, mask=mask)
```

- **Responsibility:** The programmer defines a block of work. The `tl.arange` call creates a symbolic vector of offsets. The programmer thinks "my program handles the chunk of data from `block_start` to `block_start + BLOCK_SIZE`".
- **Memory Access:** The key is `tl.load(x_ptr + offsets, ...)`. The Triton compiler sees that the program intends to load a contiguous block of `BLOCK_SIZE` elements. This high-level, declarative information is a gift to the compiler.

Why Triton's Model is a Win:

1. **Automated Coalescing:** Because the compiler sees the intent to load a whole vector (`offsets`), it can analyze the access pattern. It knows the threads in a warp will access contiguous addresses and can therefore generate the most efficient wide, vectorized load/store instructions in PTX (e.g., `ld.global.v4.b32` to load four 32-bit values at once). The programmer doesn't need to manually ensure their `threadIdx` arithmetic results in coalesced access; they just declare the block of data they want, and Triton handles the optimal fetching.

2. **Simplified Logic:** The programmer is freed from writing complex indexing code, especially for multi-dimensional tensors. Instead of `(b * H + h) * W + w`, you might just define block pointers and offsets. This makes the code cleaner, less error-prone, and more focused on the algorithm's logic.

3. **Compiler Optimizations:** The block-level perspective gives the compiler more scope for optimization. It can reason about data reuse within the block, manage shared memory for reductions (`tl.sum`), and schedule instructions to hide latency, all of which would require expert-level manual coding in CUDA C++.

In short, the CUDA model says, "Here's what one thread does; figure out how to run 32 of them efficiently." The Triton model says, "Here's a whole chunk of data I want to process; you, the compiler, are the expert on the hardware, so figure out the best way to load, store, and compute it." This higher-level abstraction allows the compiler to make informed, hardware-specific

decisions that lead to high performance without burdening the programmer with low-level hardware details.

Parallelism: DP / TP / PP / ZeRO / FSDP

> **Q: FSDP (ZeRO-3) is often preferred for its memory savings, but it seems to have 1.5x the communication volume of standard Data Parallelism (`all_reduce`). Why is it still considered highly efficient in practice?**

This is a crucial observation. While FSDP (Fully Sharded Data Parallel, equivalent to ZeRO-3) does transfer more data, its efficiency comes from **overlapping communication with computation**. The raw communication volume is a misleading metric for wall-clock time if the communication can be hidden behind computation.

Let's first confirm the communication volume. Let Ψ be the number of model parameters and N_d be the number of devices. * **Standard DDP:** Performs one `all_reduce` on the gradients per step. An `all_reduce` is equivalent to a `reduce_scatter` followed by an `all_gather`. The total data sent and received by each GPU is approximately $2 \times |\Psi|$ bytes. * **FSDP (ZeRO-3):** 1. **Forward Pass:** One `all_gather` for each layer's parameters before its computation. Total over all layers: $|\Psi|$ bytes. 2. **Backward Pass:** One `all_gather` for each layer's parameters, followed by a `reduce_scatter` for its gradients. Total over all layers: $|\Psi| + |\Psi| = 2|\Psi|$ bytes.

```
The total communication volume is approximately $3 \times |\Psi|$ bytes, which is indeed 1.5x t
```

The key is that FSDP's communication is not a single blocking call at the end of the backward pass. Instead, it's broken into smaller, per-layer chunks that can be scheduled intelligently.

Intuition: The FSDP Execution Timeline

Modern GPU programming uses streams to execute operations asynchronously. We can have a `compute_stream` for calculations (like matrix multiplies) and a `comm_stream` for data transfers (like `all_gather`).

Consider the execution for two sequential layers, `Layer L` and `Layer L+1`, in an FSDP forward pass:

1. **Initiate Comm for Layer L:** On the `comm_stream`, issue a non-blocking `all_gather` for the parameters of `Layer L`.
2. **Wait and Compute Layer L:** The `compute_stream` waits for the `comm_stream` to finish gathering the parameters for `Layer L`. Once they arrive, it performs the forward computation for `Layer L`.
3. **Initiate Comm for Layer L+1 & Free Layer L:** Crucially, *as soon as the computation for `Layer L` begins*, we can immediately issue the non-blocking `all_gather` for the parameters of `Layer L+1` on the `comm_stream`. After `Layer L`'s computation is done, its (now full-sized) parameters can be freed.
4. **Wait and Compute Layer L+1:** The `compute_stream` finishes `Layer L`, then waits for the `all_gather` of `Layer L+1`'s parameters (which was already in flight) and computes `Layer L+1`.

This creates a pipelined effect:

```
Time ->
-------------------------------------------------------------------------
Compute Stream: | Wait L | Compute L | Wait L+1 | Compute L+1 | ...
Comm Stream:    | Gather L | Gather L+1 | Gather L+2 | ...
-------------------------------------------------------------------------
```

The communication for layer `L+1` is overlapped with the computation for layer `L`.

Formal Condition for Perfect Overlap

The communication latency is hidden if the time to compute is greater than or equal to the time to communicate.

$$T_{\text{compute}}(L) \geq T_{\text{comm}}(L+1)$$

Let's model these times: * **Communication Time:** A common model for collective communication is $T_{\text{comm}}(M) = \alpha + \beta M$, where α is the latency overhead, M is the message size in bytes, and β is the reciprocal of the effective bandwidth. For an `all_gather` of a layer with $|\Psi_L|$ parameters (at 2 bytes/param), $M = 2|\Psi_L|$. * **Computation Time:** This is the number of

floating-point operations (FLOPs) for the layer, divided by the GPU's effective TFLOPS. $T_{\text{compute}}(L) = \frac{\text{FLOPs}(L)}{\text{TFLOPS}_{\text{effective}}}$.

For a Transformer layer, the FLOPs are dominated by matrix multiplications and are proportional to `batch_size * seq_len * hidden_dim^2`. The parameters $|\Psi_L|$ are proportional to `hidden_dim^2`. For large hidden dimensions, the computation time grows faster than the communication time, making it easier to hide the latency.

Pseudocode with Overlap

```
# Simplified FSDP forward pass with overlapping
compute_stream = torch.cuda.Stream()
comm_stream = torch.cuda.Stream()

# Prefetch parameters for the first layer
with torch.cuda.stream(comm_stream):
    param_l = all_gather(param_shard_l) # Non-blocking

for l in range(num_layers):
    # Wait for the current layer's params to be ready
    compute_stream.wait_stream(comm_stream)

    with torch.cuda.stream(compute_stream):
        # Once params are ready, start computation for layer l
        activations = compute_forward(activations, param_l)
        # We no longer need the full params for layer l
        del param_l

    # As soon as computation for layer l starts, prefetch params for layer l+1
    if l < num_layers - 1:
        with torch.cuda.stream(comm_stream):
            param_l_plus_1 = all_gather(param_shard_l_plus_1) # Non-blocking

# Final synchronization
torch.cuda.synchronize()
```

The same logic applies to the backward pass, where the `reduce_scatter` of layer `L` 's gradients can be overlapped with the computation of layer `L-1` 's gradients.

Pitfall: This overlap is not guaranteed. If a model has many small layers (e.g., small `hidden_dim`), the `T_compute` for each layer might be too small to hide the `T_comm` (especially the latency component α). In such cases, the GPU will spend time waiting for data, and FSDP's performance will degrade, exposing the higher communication volume as increased wall-clock time.

> **Q: How does Megatron-style Tensor Parallelism handle a sequence of matrix multiplications, like in an MLP block, without requiring an expensive `all_gather` of the intermediate activation?**

This is a clever algorithmic trick at the heart of tensor parallelism. It pairs a **column-parallel** linear layer with a subsequent **row-parallel** linear layer. This structure ensures the intermediate activation can remain sharded (distributed) across GPUs, avoiding a communication-heavy `all_gather` step.

Let's analyze a standard Transformer MLP block:

$$Y = \text{GELU}(XW_{up})W_{down}$$

Here, $X \in \mathbb{R}^{s \times h}$, $W_{up} \in \mathbb{R}^{h \times 4h}$, and $W_{down} \in \mathbb{R}^{4h \times h}$.

Let's parallelize this across T GPUs.

The Strategy: Column-Parallel followed by Row-Parallel

1. **Split `W_up` by columns:** Each GPU i gets a shard $W_{up,i} \in \mathbb{R}^{h \times (4h/T)}$. This is a **column-parallel** split.
2. **Split `W_down` by rows:** Each GPU i gets a shard $W_{down,i} \in \mathbb{R}^{(4h/T) \times h}$. This is a **row-parallel** split.

Forward Pass Derivation

Let's assume the input X is replicated on all T GPUs (this is true if the preceding layer, e.g., attention, ended with an `all_reduce`).

1. **First Matmul (Column-Parallel):** Each GPU i computes its part of the first matrix multiplication:

   $$Z_i = XW_{up,i}$$

   The result Z_i has shape $s \times (4h/T)$. If we were to concatenate these results, we would get the full intermediate tensor $Z = [Z_1, Z_2, \ldots, Z_T]$. But we don't! The tensor Z remains sharded across the GPUs.

2. **Activation Function:** The activation function (GELU) is applied element-wise on the local shard.

$$G_i = \text{GELU}(Z_i)$$

The sharded tensor $G = [G_1, G_2, \ldots, G_T]$ is still distributed. **No communication has occurred yet.**

3. **Second Matmul (Row-Parallel):** The full mathematical operation is $Y = GW_{down}$. Let's expand this with our sharded tensors:

$$Y = [G_1, G_2, \ldots, G_T] \begin{bmatrix} W_{down,1} \\ W_{down,2} \\ \vdots \\ W_{down,T} \end{bmatrix} = \sum_{i=1}^{T} G_i W_{down,i}$$

This is a beautiful result. Each GPU i can compute a partial result $Y_i = G_i W_{down,i}$ using only its local data. The final result Y is simply the sum of these partial results.

4. **Final Communication:** To get the final sum, we perform a single `all_reduce` operation on the partial results Y_i.

$$Y = \text{all\_reduce}(\{Y_1, Y_2, \ldots, Y_T\})$$

The output Y is now replicated on all GPUs, ready to be the input for the next Transformer block.

Pseudocode of the Forward Pass

```
def mlp_forward(X, W_up_shard, W_down_shard, tp_group):
    # X is replicated on all GPUs in tp_group
    # W_up_shard is a column-wise shard of W_up
    # W_down_shard is a row-wise shard of W_down

    # 1. Column-parallel matmul. Output is sharded.
    # f_identity function in forward pass
    Z_shard = matmul(X, W_up_shard)

    # 2. Element-wise activation. Output remains sharded.
    G_shard = gelu(Z_shard)

    # 3. Row-parallel matmul. Output is a partial result.
    # g_all_reduce function in forward pass
    Y_partial = matmul(G_shard, W_down_shard)

    # 4. Sum partial results and replicate across GPUs.
    Y = all_reduce(Y_partial, group=tp_group)
```

```
    return Y
```

Backward Pass

The backward pass mirrors this logic. The gradient `dY` is replicated. The backward pass of an `all_reduce` is an identity operation, so each GPU starts with `dY`. 1. The gradient for the row-parallel layer (`dG_shard`) will be sharded, which is exactly what's needed as input for the backward pass of the GELU and the column-parallel layer. 2. The gradient with respect to the input, `dX`, will be a partial sum on each GPU, requiring a final `all_reduce` to be correctly accumulated.

This elegant pairing avoids materializing the full $s \times 4h$ intermediate tensor on any single GPU and collapses the communication into a single `all_reduce` at the end of the block.

> **Q: In Pipeline Parallelism, increasing the number of microbatches `m` reduces the "bubble" overhead. However, very small microbatches can underutilize the GPU. How can we formally analyze this trade-off to find the optimal number of microbatches?**

This is a classic systems trade-off between minimizing pipeline stalls and maximizing single-device performance. We can find the optimal number of microbatches, m_{opt}, by creating a simple cost model for the total execution time and minimizing it.

Let's define our variables: * p: Number of pipeline stages (GPUs). * B: Global batch size. * m: Number of microbatches. * b: Microbatch size, where $b = B/m$.

1. Modeling Single-GPU Microbatch Time

The time to process one microbatch of size b on a single GPU, $T_{step}(b)$, is not perfectly linear. It includes fixed overheads (kernel launches, memory allocation) and a compute component. A simple but effective model is:

$$T_{step}(b) = T_{ov} + T_{comp} \cdot b$$

* T_{ov}: The fixed overhead per microbatch, independent of its size. * T_{comp}: The compute time per single example in the microbatch.

2. Modeling Total Pipeline Execution Time

Using the standard 1F1B (one forward, one backward) schedule, the pipeline has a "bubble" at the beginning (filling the pipe) and end (draining the pipe). The total time to process a global batch is the sum of the time for all microbatches to pass through the steady-state portion of the pipe, plus the bubble time.

The total duration consists of $(m + p - 1)$ "ticks," where each tick is the time to process one microbatch on one stage.

$$T_{wall}(m) = (m + p - 1) \cdot T_{step}(b)$$

Note: This is a slight simplification. A more precise model for 1F1B would be `(m+p-1) * (T_f + T_b)`, but this simpler model captures the core trade-off.

Now, substitute our models for $T_{step}(b)$ and $b = B/m$:

$$T_{wall}(m) = (m + p - 1) \left(T_{ov} + T_{comp} \frac{B}{m} \right)$$

Let's expand this expression:

$$T_{wall}(m) = mT_{ov} + mT_{comp} \frac{B}{m} + (p - 1)T_{ov} + (p - 1)T_{comp} \frac{B}{m}$$

$$T_{wall}(m) = mT_{ov} + T_{comp}B + (p - 1)T_{ov} + \frac{(p - 1)T_{comp}B}{m}$$

The term $T_{comp}B$ is the total compute time for the batch, which is constant. The term $(p - 1)T_{ov}$ is also constant with respect to m. The parts that vary with m are: * mT_{ov}: Total overhead cost. This increases linearly with m (more, smaller microbatches means more overhead). * $\frac{(p-1)T_{comp}B}{m}$: Total bubble cost. This is the compute work of $(p - 1)$ microbatches that is not parallelized. It decreases as m increases.

3. Finding the Optimal `m`

To find the value of m that minimizes $T_{wall}(m)$, we can take the derivative with respect to m and set it to zero. We only need to consider the terms that depend on m.

$$\frac{d}{dm} \left(mT_{ov} + \frac{(p - 1)T_{comp}B}{m} \right) = 0$$

$$T_{ov} - \frac{(p - 1)T_{comp}B}{m^2} = 0$$

$$T_{ov} = \frac{(p-1)T_{comp}B}{m^2}$$

Solving for m:

$$m^2 = \frac{(p-1)B \cdot T_{comp}}{T_{ov}}$$

$$m_{opt} = \sqrt{(p-1)B\frac{T_{comp}}{T_{ov}}}$$

Interpretation of the Result

This formula provides powerful intuition:

- **Pipeline Depth (`p-1`):** A deeper pipeline (larger p) requires more microbatches to hide the larger bubble.
- **Global Batch Size (`B`):** A larger global batch allows for more microbatches.
- **Arithmetic Intensity (`T_comp / T_ov`):** This ratio is key.
  - If computation is expensive relative to overhead ($T_{comp} \gg T_{ov}$), the ratio is large. It's worth it to have many small microbatches to reduce the bubble, as the overhead of each is negligible.
  - If overhead is high relative to computation ($T_{ov} \gg T_{comp}$), e.g., for small layers or fast GPUs, the ratio is small. It's better to use fewer, larger microbatches to amortize the high fixed cost, even if it means accepting a larger pipeline bubble.

In practice, T_{ov} and T_{comp} must be empirically measured for a given model and hardware. However, this derivation proves that simply "making microbatches smaller to reduce the bubble" is not always the right answer. There is a sweet spot that optimally balances the two competing costs.

Scaling Laws & compute-optimal training

> **Q: The notes highlight that Chinchilla's main correction to earlier work (like Kaplan et al.) was properly accounting for the cosine learning rate schedule. Why, precisely,**

> **does using intermediate checkpoints from a single long training run with a cosine schedule lead to incorrect scaling law estimates?**

This is a subtle but critical point that gets to the heart of why fitting scaling laws is so computationally demanding. The error stems from a mismatch between the training dynamics of a truncated run and a run that was *designed* to be short from the outset.

The Goal & The Flawed Shortcut

The goal of this analysis is to find the loss `L(N, D)` as a function of model size `N` and dataset size `D`. To do this, we need to measure the final loss for many different pairs of `(N, D)`.

The flawed shortcut, used in some early analyses, is to train one large model for a very long time (many tokens) and save checkpoints along the way. The assumption is that a checkpoint at `D_i` tokens represents the final state of a model trained on a dataset of size `D_i`. This would be a huge computational saving, as one long run could provide data for dozens of `(N, D)` points.

Why the Cosine Schedule Breaks This Shortcut

Modern LLMs are almost universally trained with a cosine annealing learning rate schedule. The learning rate `η` at step `t` is given by:

$$\eta(t) = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{\pi t}{T_{total}}\right)\right)$$

The crucial variable here is `T_{total}`, the total number of training steps planned for the entire run. The entire *shape* of the learning rate curve depends on this final horizon.

Let's compare two scenarios:

1. **Full Run:** We train a model for `T_{total} = 100,000` steps. The LR schedule is a slow, gradual decay over all 100k steps.
2. **Truncated Run:** We take the model from Scenario 1 at step `t = 20,000`. The flawed assumption is that this model is representative of a model trained optimally for just 20,000 steps.
3. **Correct Short Run:** We train a new model from scratch with `T_{total} = 20,000`. Its cosine schedule will be much more aggressive, decaying from `η_max` to `η_min` over just 20k steps.

The Mismatch in Dynamics

The model from the truncated run (Scenario 2) is "undertrained" relative to its learning rate. At step 20,000, its learning rate is still very high because it's "expecting" another 80,000 steps of training. In contrast, the model from the correct short run (Scenario 3) has already gone through its entire high-to-low LR cycle. It has spent more of its (short) life at lower learning rates, allowing it to settle into a better minimum for that specific training duration.

Therefore, `Loss(Scenario 2)` will almost certainly be higher than `Loss(Scenario 3)`. Using the loss from the truncated run as a data point for `L(N, D=20k_steps)` would be an overestimate of the model's true capability for that data budget, biasing the resulting scaling law.

Chinchilla's Solution (and its cost)

The Chinchilla authors recognized this flaw. Their solution was simple but expensive: for each data point `(N_i, D_i)` they wanted to measure, they trained a separate model from scratch with a full cosine schedule optimized for that specific `D_i` (i.e., for `T_{total} = D_i / \text{batch_size}`). This ensures that every measured loss value represents a properly converged model for its given budget. This increases the computational cost of the analysis quadratically, but yields a much more accurate result.

A More Modern, Efficient Alternative: WSD Schedules

Later work (e.g., MiniCPM, DeepSeek) proposed a compromise using a **Warm-up Stable Decay (WSD)** or trapezoidal learning rate schedule.

```
def wsd_schedule(step, warmup_steps, total_steps, decay_steps):
    stable_steps = total_steps - warmup_steps - decay_steps
    if step < warmup_steps:
        # Linear warmup
        return (step / warmup_steps) * max_lr
    elif step < warmup_steps + stable_steps:
        # Stable phase
        return max_lr
    else:
        # Cosine decay over the last part
        decay_progress = (step - warmup_steps - stable_steps) / decay_steps
        return min_lr + 0.5 * (max_lr - min_lr) * (1 + math.cos(math.pi * decay_progress))
```

The key insight is that during the long "stable" phase, the training dynamics are consistent. One can perform a single long run, and then for each desired data budget `D_i`, rewind to the

checkpoint at the end of the stable phase for that budget and apply the *final, short decay phase*. This largely avoids the cosine schedule mismatch problem while being much cheaper than training dozens of models from scratch.

> **Q: Chinchilla's "Method 2" uses "IsoFLOP curves" to find the optimal model and data size. Can you provide a step-by-step algorithmic derivation of this method and the intuition behind it?**

The IsoFLOPs method is an elegant and robust way to determine the optimal trade-off between model size and data size for a fixed compute budget. It directly answers the question: "If I have a budget of `C` FLOPs, what is the best model size `N` and number of training tokens `D` to use?"

The Core Constraint: The Compute Equation

First, we need a model for the total training compute `C` (in FLOPs). A widely used approximation for a Transformer is:

$$C \approx 6 \times N \times D$$

Where: - `N` is the number of non-embedding parameters. - `D` is the number of training tokens. - The factor of 6 comes from: `2` for the matrix multiply in the forward pass (`2 * N`), `4` for the corresponding backward pass (`4 * N`), totaling `6N` FLOPs per token.

An "IsoFLOP curve" (iso meaning "same") is the set of all pairs `(N, D)` that result in the same total compute `C` . On a plot of `D` vs. `N` , this is a hyperbola: `D = C / (6N)` .

The IsoFLOP Algorithm

The algorithm proceeds in two main phases: (1) Finding the optimal point for several fixed budgets, and (2) Fitting a scaling law to those optimal points.

Phase 1: Finding Optimal `(N, D)` for Fixed Budgets `C`

1. **Select Compute Budgets:** Choose a set of exponentially spaced compute budgets to explore, e.g., `C_1 = 10^20` , `C_2 = 10^21` , `C_3 = 10^22` FLOPs.
2. **For each budget `C_k` :** a. **Generate IsoFLOP Points:** Choose several points `(N_i, D_i)` that lie on the IsoFLOP curve for `C_k` . A good strategy is to vary `N` exponentially. For example, for `C_k = 10^{21}` :

- Point 1 (Small Model, Lots of Data): `N_1 = 70M`, so `D_1 = 10^{21} / (6 * 70M) ≈ 2.4T` tokens.
- Point 2 (Medium Model, Medium Data): `N_2 = 350M`, so `D_2 = 10^{21} / (6 * 350M) ≈ 480B` tokens.
- Point 3 (Large Model, Less Data): `N_3 = 1.75B`, so `D_3 = 10^{21} / (6 * 1.75B) ≈ 95B` tokens.

b. **Train Models:** Train a separate model from scratch for each point `(N_i, D_i)`. Each model of size `N_i` is trained for `D_i` tokens. Record the final validation loss `L_i`.

c. **Find the Minimum:** Plot the resulting losses `L_i` against the model sizes `N_i`. This will produce a characteristic U-shaped curve. The bottom of the "U" represents the best trade-off. Find the model size `N_{opt}(C_k)` that minimizes the loss for this compute budget `C_k`. The corresponding optimal number of tokens is `D_{opt}(C_k) = C_k / (6 \times N_{opt}(C_k))`.

Phase 2: Fitting the Scaling Law

3. **Collect Optimal Points:** After repeating Phase 1 for all chosen budgets, you will have a set of optimal allocations:
   - `(C_1, N_{opt}(C_1), D_{opt}(C_1))`
   - `(C_2, N_{opt}(C_2), D_{opt}(C_2))`
   - `...`
4. **Fit Power Laws:** The hypothesis is that the optimal model size and data size scale as a power law of the compute budget:

$$N_{opt}(C) = K_N \cdot C^a$$

$$D_{opt}(C) = K_D \cdot C^b$$

To find the exponents `a` and `b`, we perform a linear regression in log-log space:

$$\log(N_{opt}) = a \cdot \log(C) + \log(K_N)$$

$$\log(D_{opt}) = b \cdot \log(C) + \log(K_D)$$

The slopes of these lines give us the desired scaling exponents `a` and `b`.

Pseudocode for the entire process:

```
budgets = [1e20, 1e21, 1e22, 1e23] # FLOPs
optimal_points = []

for C in budgets:
    # Phase 1: Find optimum for this budget C
    model_sizes_to_test = [70e6, 350e6, 1.75e9, 8e9] # Example N values
    results_for_C = []
    for N in model_sizes_to_test:
        D = C / (6 * N)
        # Train model of size N on D tokens to convergence
        final_loss = train_model(N, D)
        results_for_C.append({'N': N, 'loss': final_loss})

    # Find the N that resulted in the minimum loss for this C
    # (In practice, you'd fit a quadratic or other curve to the points)
    best_result = min(results_for_C, key=lambda x: x['loss'])
    N_opt = best_result['N']
    D_opt = C / (6 * N_opt)
    optimal_points.append({'C': C, 'N_opt': N_opt, 'D_opt': D_opt})

# Phase 2: Fit scaling law to the optimal points
log_C = [math.log(p['C']) for p in optimal_points]
log_N_opt = [math.log(p['N_opt']) for p in optimal_points]
log_D_opt = [math.log(p['D_opt']) for p in optimal_points]

# Perform linear regression: log_N_opt = a * log_C + const
a = linear_regression(log_C, log_N_opt).slope
# Perform linear regression: log_D_opt = b * log_C + const
b = linear_regression(log_C, log_D_opt).slope

print(f"Optimal scaling exponents: a={a:.2f}, b={b:.2f}")
```

Intuition and Result

The U-shape of the loss curve for a fixed budget makes intuitive sense: - **Too small `N` (left side of U):** The model lacks the capacity to learn from the vast amount of data (`D` is huge). It becomes the bottleneck. - **Too large `N` (right side of U):** The model is too large for the small amount of data it's shown (`D` is small). It is severely under-trained and cannot converge to a good solution.

The IsoFLOPs method systematically finds the "sweet spot" at the bottom of this U for various scales. Chinchilla's key finding using this method was that `a ≈ 0.5` and `b ≈ 0.5`. This implies that as you increase your compute budget `C`, you should scale both your model size and your

dataset size proportionally to `sqrt(C)`. This was a major departure from previous work which suggested scaling `N` much more aggressively than `D`.

> **Q: The notes state that Chinchilla-optimal training may not be what you want for deployment, leading to a trend of "over-training". Can you formalize the trade-off between training and inference costs to show why a smaller, "over-trained" model can be economically superior?**

This is a crucial insight that bridges scaling law research with real-world production economics. The "compute-optimal" point found by Chinchilla minimizes the FLOPs required during **training** to reach a certain loss. However, for any successful model, the total compute spent on **inference** over its lifetime can dwarf the one-time training cost.

Formalizing the Costs

Let's define two cost components:

1. **Training Cost (`Cost_train`):** This is the one-time compute budget, which we model as `C = 6ND`.
2. **Inference Cost (`Cost_inference`):** This is the ongoing cost. For each forward pass, the cost is proportional to the model size `N`. If we expect a model to serve a total of `K` tokens over its lifetime, the total inference cost is:

   $$\text{Cost}_{inference} \approx \gamma \cdot K \cdot N$$

   where `γ` is a constant representing FLOPs per parameter per token for inference (roughly 2). The key is that inference cost is directly proportional to model size `N`.

The **Total Cost of Ownership (TCO)** of the model is:

$$\text{TCO}(N, D) = \text{Cost}_{train} + \text{Cost}_{inference} = 6ND + \gamma KN$$

The Chinchilla-Optimal Strategy

Chinchilla's analysis provides a recipe to achieve a target loss `L_target` with the minimum training compute `C_train`. Let's say this recipe calls for a model of size `N_{opt}` trained on `D_{opt}` tokens. - `C_{opt} = 6 N_{opt} D_{opt}` - The TCO for this strategy is: `TCO_{opt} = C_{opt} + \gamma K N_{opt}`

The "Over-training" Strategy

Now, consider an alternative. What if we choose a **smaller model**, `N' < N_{opt}`? - **Benefit:** The inference cost will be lower: `Cost'_{inference} = \gamma K N' < \gamma K N_{opt}`. - **Problem:** To reach the *same target loss* `L_target` with a smaller model, we must compensate for its lower capacity by training it on much more data. Let's call this `D'`.

We can use the Chinchilla scaling law for loss to find the required `D'`:

$$L(N, D) = A \cdot N^{-\alpha} + B \cdot D^{-\beta} + E_{ir}$$

We set `L(N_{opt}, D_{opt}) = L(N', D')` and solve for `D'`:

$$A \cdot N_{opt}^{-\alpha} + B \cdot D_{opt}^{-\beta} = A \cdot (N')^{-\alpha} + B \cdot (D')^{-\beta}$$

$$(D')^{-\beta} = \frac{A}{B}(N_{opt}^{-\alpha} - (N')^{-\alpha}) + D_{opt}^{-\beta}$$

Since `N' < N_{opt}`, the term `(N_{opt}^{-\alpha} - (N')^{-\alpha})` is negative, but this seems wrong. Let's re-examine the equation. `N' < N_{opt}` means `(N')^{-α} > N_{opt}^{-α}`. So the term is positive. This means `(D')^{-β}` will be larger, which implies `D'` will be smaller. This is the opposite of what we expect.

Let's re-think the logic. The IsoFLOP curves show that for a *fixed* compute budget, moving away from `N_opt` increases loss. To get back to the same loss level with a smaller model, we must *increase* the total compute budget by training for longer.

Let's use a simpler framing. The loss `L` is on a contour line in the `(N, D)` plane. The Chinchilla-optimal point `(N_{opt}, D_{opt})` is the point on this contour that minimizes `C = 6ND`. Any other point `(N', D')` on the same loss contour will have `C' = 6N'D' > C_{opt}`.

So, the "over-training" strategy is: 1. Pick a smaller model size `N' < N_{opt}`. 2. Find the data `D'` required to hit the same loss `L_target`. This point `(N', D')` will lie on the same loss iso-line as `(N_{opt}, D_{opt})`. 3. This new training plan will have a higher training cost: `C' = 6N'D' > C_{opt}`. 4. The TCO for this new strategy is: `TCO' = C' + \gamma K N'`.

When is Over-training Worth It?

We prefer the over-training strategy if `TCO' < TCO_{opt}`:

$$C' + \gamma K N' < C_{opt} + \gamma K N_{opt}$$

$$(C' - C_{opt}) < \gamma K (N_{opt} - N')$$

Let's analyze this inequality: - **Left side (`C' - C_{opt}`):** This is the **increase in training cost**. It's a positive, one-time cost. - **Right side (`γK(N_{opt} - N')`):** This is the **total savings in inference cost** over the model's lifetime. It's also positive.

The inequality shows that over-training is economically superior if the lifetime inference savings outweigh the upfront increase in training cost.

Numerical Example

- Suppose Chinchilla-optimal for loss `L` is `N_{opt}=70B`, `D_{opt}=1.4T` tokens. `C_{opt} \propto 70 \times 1.4 = 98`.
- Suppose we can achieve the same loss `L` with a smaller model `N'=50B` by "over-training" on `D'=2.5T` tokens.
- The new training cost is `C' \propto 50 \times 2.5 = 125`. The training cost increased by `125 - 98 = 27` units.
- The inference cost is proportional to `N`. The savings are proportional to `N_{opt} - N' = 70 - 50 = 20`.
- The trade-off becomes: `27 < (\text{Inference Cost Factor}) \times 20`.

If the model is a research prototype used a few times, the inference factor is small, and the Chinchilla-optimal model is cheaper overall. But if it's a production model like Llama 3 serving billions of queries (`K` is huge), the inference factor is massive, the inequality holds, and it is far more economical to pay the extra training cost to get a smaller, cheaper-to-run final model. This is why we see models like Llama 3 being trained on over 15T tokens, far exceeding the original Chinchilla ratio of 20 tokens/parameter.

Inference: KV cache, batching, speculative decoding

> **Q: The lecture notes claim that during the token generation phase, the arithmetic intensity of attention is approximately 1 and that batching doesn't help, unlike for**

> **MLP layers. Can you provide a rigorous derivation for this and explain the core intuition behind why the batch size `B` cancels out?**

This is a crucial observation that explains why the generation step of autoregressive decoding is fundamentally memory-bound. Let's break it down by analyzing the MLP and Attention layers separately.

Recall: Arithmetic Intensity (AI) is the ratio of floating-point operations (FLOPs) to bytes of data moved between high-bandwidth memory (HBM) and the processor.

$$\text{AI} = \frac{\text{Total FLOPs}}{\text{Total Bytes Transferred}}$$

A workload is compute-bound if its AI is higher than the accelerator's intensity (e.g., ~295 for an H100); otherwise, it's memory-bound.

1. MLP Layers during Generation

In the generation phase, we process a batch of `B` sequences, each generating one new token. So, the input to each MLP layer is a tensor of shape `(B, 1, D)`, where `D` is the model dimension. An MLP block typically consists of three matrix multiplications (up-projection, gate, down-projection). Let's analyze one, `Y = X @ W_up`, where `X` is `(B, D)` and `W_up` is `(D, F)`.

- **FLOPs:** The number of operations for a matrix multiplication is `2 * B * D * F`.
- **Bytes Transferred:**
  - Read input `X`: `B * D * 2` bytes (assuming BF16).
  - Read weights `W_up`: `D * F * 2` bytes.
  - Write output `Y`: `B * F * 2` bytes.
  - Total Bytes: `2 * (BD + DF + BF)`.

The arithmetic intensity is:

$$\text{AI}_{\text{MLP}} = \frac{2 \cdot BDF}{2 \cdot (BD + DF + BF)} = \frac{BDF}{BD + DF + BF}$$

In a typical large model, the hidden dimensions `D` and `F` are much larger than the batch size `B` (e.g., `D, F` are in the thousands, `B` is tens or hundreds). Therefore, the `DF` term in the denominator dominates:

$$\text{AI}_{\text{MLP}} \approx \frac{BDF}{DF} = B$$

Intuition: The largest piece of data moved is the weight matrix `W_up`, which is `O(DF)`. This single matrix is read from HBM and then re-used for all `B` sequences in the batch. The computation `O(BDF)` grows linearly with `B`, but the largest memory cost `O(DF)` is amortized. Thus, increasing the batch size `B` directly increases the arithmetic intensity, helping to make the operation compute-bound.

2. Attention Layers during Generation

The situation is starkly different for attention. During generation, for each of the `B` sequences, the new query vector `q` (shape `(1, D)`) must attend to all `S` previous tokens in its own history, which are stored in its personal Key-Value (KV) cache.

Let's analyze the first major computation: the dot-product of queries with keys, `AttentionScores = Q @ K^T`. * `Q` is the batch of new queries: shape `(B, 1, D)`. * `K_{cache}` is the batch of cached keys: shape `(B, S, D)`.

- **FLOPs:** We perform `B` independent matrix multiplications of a `(1, D)` matrix with a `(D, S)` matrix.
  - Total FLOPs: `B \times (2 \cdot 1 \cdot S \cdot D) = 2 \cdot B \cdot S \cdot D`.
- **Bytes Transferred:**
  - Read queries `Q`: `B * 1 * D * 2` bytes.
  - Read key cache `K_{cache}`: `B * S * D * 2` bytes.
  - Write attention scores: `B * 1 * S * 2` bytes.
  - Total Bytes (ignoring the small output): `2 \cdot (BD + BSD) = 2 \cdot BD(S+1)`.

The arithmetic intensity is:

$$\text{AI}_{\text{Attention}} = \frac{2 \cdot BSD}{2 \cdot BD(S+1)} = \frac{S}{S+1}$$

As the context length `S` grows, this value approaches 1. Notice that the batch size `B` has completely canceled out.

Intuition: Unlike the MLP layers where a single large weight matrix is shared across the batch, in attention, **each sequence has its own private KV cache**. When you double the batch size `B`,

you also double the total amount of KV cache data that must be read from HBM (`B * S * D`). Both the FLOPs and the memory traffic scale linearly with `B` , so their ratio—the arithmetic intensity—remains constant.

The same logic applies to the second major computation, `Output = AttentionScores @ V` , where the AI also works out to be independent of `B` .

Conclusion: Because the KV cache is not shared between sequences in a batch, batching does not amortize the memory access cost for attention layers during generation. The arithmetic intensity remains stuck at a very low value (`≈1`), far below the threshold for compute-bound execution on modern accelerators. This makes the generation step of Transformers inherently **memory-bandwidth bound**.

> **Q: Speculative decoding accepts a token `x_i` from a draft model `p` with probability `min(1, q(x_i|H) / p(x_i|H))` , where `q` is the target model. How can we be sure this process generates a sequence with the exact same distribution as sampling directly from the target model `q` ?**

This is a fantastic question, as the correctness of speculative decoding relies on a clever application of rejection sampling. The guarantee that the final sample comes from the target distribution `q` is not immediately obvious. Let's derive it.

Let `H` be the history (prompt + previously generated tokens). We want to sample the next token `x` from the target distribution `q(x|H)` . We use a cheaper draft model that produces a distribution `p(x|H)` .

The algorithm for one step is as follows: 1. Sample a candidate token `x' ~ p(x|H)` . 2. Calculate the acceptance ratio `r = q(x'|H) / p(x'|H)` . 3. Accept `x'` with probability `min(1, r)` . 4. If `x'` is accepted, it becomes the final output for this step. 5. If `x'` is rejected, we must sample a token from a modified "correction" distribution.

The key is to show that the combination of the accepted draft tokens and the corrected rejection samples perfectly reconstructs the original target distribution `q` .

Derivation

Let's find the probability of sampling a specific token `x_k` as the final output, `P(x_{final} = x_k)`. A token can be chosen in two mutually exclusive ways: either it was the draft token and was accepted, or the draft token was rejected and it was chosen via the correction mechanism.

Case 1: The draft token `x_k` is sampled and accepted.

The probability of this joint event is:

$$P(\text{sample } x_k \text{ and accept } x_k) = p(x_k|H) \times \min\left(1, \frac{q(x_k|H)}{p(x_k|H)}\right)$$

This simplifies to:

$$P(\text{sample } x_k \text{ and accept } x_k) = \min(p(x_k|H), q(x_k|H))$$

This is the portion of the probability mass for `x_k` that is "covered" by the acceptance step. However, this is clearly not equal to `q(x_k|H)`. The missing mass must be accounted for by the rejection case.

Case 2: The draft token is rejected, and `x_k` is sampled from the correction distribution.

First, let's define the "missing" probability mass for each token `x_k`. This is the probability that `q` assigns to `x_k` minus the probability that our acceptance step assigns to it.

$$\text{MissingMass}(x_k) = q(x_k|H) - \min(p(x_k|H), q(x_k|H))$$

This can be written more cleanly using the positive part function `(z)^+ = max(0, z)`:

$$\text{MissingMass}(x_k) = (q(x_k|H) - p(x_k|H))^+$$

If `q(x_k|H) <= p(x_k|H)`, the missing mass is 0. If `q(x_k|H) > p(x_k|H)`, the acceptance step only covered `p(x_k|H)`, so we are missing `q(x_k|H) - p(x_k|H)`.

The total probability of rejecting the draft sample is the sum of all missing masses over the entire vocabulary `V`:

$$P(\text{reject}) = \sum_{x \in V} \text{MissingMass}(x) = \sum_{x \in V} (q(x|H) - p(x|H))^+$$

If a rejection occurs, we must sample from the normalized distribution of this missing mass. This is our **correction distribution**, `q_{corr}`:

$$q_{corr}(x_k|H) = \frac{\text{MissingMass}(x_k)}{P(\text{reject})} = \frac{(q(x_k|H) - p(x_k|H))^+}{\sum_{x \in V}(q(x|H) - p(x|H))^+}$$

The probability of sampling `x_k` via this rejection path is:

$$P(\text{reject and then sample } x_k) = P(\text{reject}) \times q_{corr}(x_k|H)$$

$$P(\text{reject and then sample } x_k) = P(\text{reject}) \times \frac{(q(x_k|H) - p(x_k|H))^+}{P(\text{reject})} = (q(x_k|H) - p(x_k|H)$$

Putting It All Together

The total probability of `x_k` being the final sample is the sum of the probabilities from the two cases:

$$P(x_{final} = x_k) = P(\text{sample } x_k \text{ and accept } x_k) + P(\text{reject and then sample } x_k)$$

$$P(x_{final} = x_k) = \min(p(x_k|H), q(x_k|H)) + (q(x_k|H) - p(x_k|H))^+$$

Let's check the two possibilities for the relationship between `q` and `p`: 1. If `q(x_k|H) \le p(x_k|H)`: $P(x_{final} = x_k) = q(x_k|H) + 0 = q(x_k|H)$. 2. If `q(x_k|H) > p(x_k|H)`: $P(x_{final} = x_k) = p(x_k|H) + (q(x_k|H) - p(x_k|H)) = q(x_k|H)$.

In both cases, the final probability is exactly `q(x_k|H)`. The procedure is mathematically guaranteed to be an exact sample from the target model `q`.

```
def speculative_sample_step(history, target_model, draft_model):
    """
    Performs one step of speculative sampling.
    Note: In practice, this is done for a block of K tokens.
    """
    # 1. Draft: Sample a candidate from the cheap model p
    draft_logits = draft_model.get_logits(history)
    draft_probs = softmax(draft_logits)
    candidate_token = sample_from(draft_probs)

    # 2. Verify: Get probabilities for the candidate from both models
    # This is the expensive step, but it's just one forward pass
    target_logits = target_model.get_logits(history)
    target_probs = softmax(target_logits)

    p_prob = draft_probs[candidate_token]
    q_prob = target_probs[candidate_token]

    # 3. Accept/Reject
    acceptance_prob = min(1.0, q_prob / p_prob)
```

```
        if random.random() < acceptance_prob:
            # Accept the draft token
            return candidate_token
        else:
            # 4. Reject and Resample from the correction distribution
            # The correction distribution is proportional to (q - p)^+
            # This can be implemented efficiently without re-calculating softmax
            correction_probs = (target_probs - draft_probs).clamp(min=0)
            norm_factor = correction_probs.sum()
            if norm_factor < 1e-9: # Handle case where p >= q everywhere
                # This can happen if p is a perfect or overly confident model
                # In this rare case, we must resample from q.
                return sample_from(target_probs)

            correction_probs /= norm_factor
            return sample_from(correction_probs)
```

Intuition: The draft model `p` acts as a cheap proposal distribution. We use it to quickly guess a token. If the target model `q` would have been *at least as likely* to pick that token (`q/p >= 1`), we just accept it. If `q` was less likely (`q/p < 1`), we accept it some of the time. The magic is in the rejection step: we precisely calculate the "probability mass" that our acceptance rule missed and sample from that residual. This perfectly patches the distribution back up to match `q` .

> **Q: PagedAttention is said to use a "copy-on-write" mechanism to efficiently handle multiple output sequences sharing a common prefix. How is this implemented at the level of KV cache memory blocks, and what is the precise memory saving compared to a naive implementation?**

This question targets the core systems innovation of PagedAttention, which is crucial for high-throughput serving. The "copy-on-write" analogy from operating systems is powerful but can be slightly misleading; for standard decoding, it's more of a "share-then-append" mechanism.

The Problem: Shared Prefixes in Inference

A common inference scenario involves generating multiple sequences from a single prompt. Examples include: * **Beam Search:** Generating `k` candidate sequences (beams). * **Parallel Sampling:** Generating `N` independent samples for one user query to find the best one. * **Batching identical prompts:** Multiple users asking the same question.

In all these cases, the initial processing of the prompt (the "prefill" stage) results in an identical KV cache for all `N` sequences.

Naive KV Cache Management

Without PagedAttention, an inference server would typically allocate a contiguous block of memory for the KV cache of each sequence, large enough to hold the prompt and the maximum possible generation length.

- **Implementation:** After running prefill on the prompt once, the resulting KV cache data would be physically copied (`memcpy`) into the `N` separate buffers.
- **Memory Cost:** Let `S_p` be the prompt length, `L` be the number of layers, and `D` be the model dimension. The memory for one token's K and V vectors (in MHA) is `2 * L * D * sizeof(dtype)` . Let this be `C_token` . The total memory for the `N` prefixes is:

  $$M_{\text{naive}} = N \times S_p \times C_{\text{token}}$$

  This is extremely wasteful, as `(N-1) * S_p * C_token` bytes are redundant copies. This wasted memory reduces the number of concurrent requests a GPU can handle.

PagedAttention: Virtualizing the KV Cache

PagedAttention solves this by borrowing the concept of virtual memory and paging from operating systems.

1. **Physical Blocks:** The entire GPU memory allocated for the KV cache is divided into fixed-size blocks (or "pages").
2. **Logical Blocks & Page Tables:** Each sequence's KV cache is a *logical* sequence of blocks. A per-sequence **page table** maps these logical block indices to physical block indices in GPU memory. This means a sequence's KV cache can be stored in non-contiguous physical blocks.
3. **Reference Counting:** Each physical block maintains a reference counter.

"Copy-on-Write" in Action (Share-then-Append):

Let's trace the `N` parallel samples scenario:

1. **Prefill:** The prompt is processed once. The resulting KV cache occupies, say, `k` physical blocks.

2. **Sharing:** Instead of copying, PagedAttention creates `N` logical sequences. The page tables for all `N` sequences are initialized to point to the *same* `k` physical blocks. The reference count of each of these `k` blocks is set to `N`.
3. **Generation (Divergence):** Now, we generate the first new token for each of the `N` sequences.
    - For sequence `i`, the model generates a new key and value vector.
    - The PagedAttention memory manager allocates a **new, free physical block** from its pool.
    - The page table for sequence `i` is appended with the index of this new physical block.
    - The reference count of this new block is set to 1.
    - The reference counts of the original `k` shared blocks remain unchanged at `N`.

No data from the prefix was ever copied. The sequences continue to share the read-only prefix blocks while appending their own unique, private blocks for the newly generated tokens.

```
# Conceptual pseudocode for PagedAttention's sharing mechanism
class MemoryManager:
    # Manages a pool of physical blocks and their reference counts
    physical_blocks: dict[int, KVCacheBlock]
    ref_counts: dict[int, int]
    free_block_indices: list[int]

    def allocate_block(self) -> int:
        # ... returns index of a free block

    def free_block(self, block_idx: int):
        # ... decrements ref_count and adds to free pool if zero

class Sequence:
    # Each sequence has a page table mapping logical to physical blocks
    page_table: list[int] # list of physical block indices

# --- Scenario: N=3 parallel samples from one prompt ---

manager = MemoryManager()

# 1. Prefill: Process prompt, store in new physical blocks
prompt_kv_data = prefill(prompt)
prompt_physical_blocks = manager.store_in_new_blocks(prompt_kv_data) # e.g., [101, 102]

# 2. Share: Create 3 sequences pointing to the same physical blocks
sequences = [Sequence() for _ in range(3)]
for seq in sequences:
    seq.page_table = list(prompt_physical_blocks) # All share [101, 102]

# Set reference counts for the shared blocks
```

```
    for block_idx in prompt_physical_blocks:
        manager.ref_counts[block_idx] = 3

    # 3. Generate (Diverge): Generate one token for each sequence
    for i, seq in enumerate(sequences):
        # Generate new KV for this sequence's next token
        new_kv_data = generate_step(seq)

        # Allocate a new physical block for this unique token
        new_block_idx = manager.allocate_block() # e.g., 201 for seq 0, 202 for seq 1, etc.
        manager.store_data(new_block_idx, new_kv_data)
        manager.ref_counts[new_block_idx] = 1

        # Append the new physical block to this sequence's page table
        seq.page_table.append(new_block_idx)

    # After one step:
    # seq[0].page_table = [101, 102, 201]
    # seq[1].page_table = [101, 102, 202]
    # seq[2].page_table = [101, 102, 203]
    # manager.ref_counts = {101: 3, 102: 3, 201: 1, 202: 1, 203: 1}
```

Memory Savings Analysis

Let `S_p` be the prompt length, `S_g` be the generated length per sequence, and `C_token` be the memory cost per token in the KV cache.

- **Naive Memory:** Each of the `N` sequences requires its own buffer for the full length.

$$M_{\text{naive}} = N \times (S_p + S_g) \times C_{\text{token}}$$

- **PagedAttention Memory:** The prefix is stored once, and each of the `N` sequences gets its own memory for the generated part.

$$M_{\text{PagedAttention}} = (S_p \times C_{\text{token}}) + (N \times S_g \times C_{\text{token}})$$

- **Memory Saved:** The difference is the memory saved by not duplicating the prefix.

$$\text{Savings} = M_{\text{naive}} - M_{\text{PagedAttention}} = (N - 1) \times S_p \times C_{\text{token}}$$

This saving is immense, especially for applications with long system prompts or few-shot examples (`S_p` is large) and a high degree of parallelism (`N` is large). By eliminating this redundancy, PagedAttention can dramatically increase the effective batch size and overall throughput of an inference server.

Alignment math: SFT, RLHF/PPO, DPO, GRPO

> **Q: The DPO paper claims to optimize the standard RLHF objective without an explicit reward model or reinforcement learning. How is the DPO loss function derived, and what is the key "non-parametric assumption" that makes this transformation from a KL-constrained RL problem to a simple binary classification loss possible?**

This is a fantastic question that gets to the heart of DPO's elegance. The derivation connects a complex reinforcement learning objective to a simple supervised loss by re-parameterizing the reward function in terms of the policy itself.

Let's walk through the derivation step-by-step.

1. The Standard RLHF Objective

The goal in RLHF is to find a policy π_θ that maximizes the expected reward from a reward model $r_\phi(x, y)$, while not straying too far from an initial reference policy π_{ref} (usually the SFT model). This is formulated as a KL-constrained optimization problem:

$$\max_{\pi_\theta} \mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta(y|x)}[r(x,y)] - \beta D_{KL}(\pi_\theta(y|x)||\pi_{ref}(y|x))$$

Here, β is a hyperparameter that controls the strength of the KL penalty. A higher β means we penalize deviation from π_{ref} more heavily.

2. The Closed-Form Optimal Policy (The "Non-Parametric" Insight)

The key insight from the DPO paper (and prior work) is that for a *given* reward function $r(x, y)$, this optimization problem has a known closed-form solution for the optimal policy, which we'll call $\pi_r(y|x)$. This solution is found by assuming we can optimize over the space of *all possible distributions* (a "non-parametric" assumption), not just those representable by our specific neural network architecture.

The optimal policy $\pi_r(y|x)$ is:

$$\pi_r(y|x) = \frac{1}{Z(x)}\pi_{ref}(y|x)\exp\left(\frac{1}{\beta}r(x,y)\right)$$

where $Z(x) = \sum_y \pi_{ref}(y|x) \exp\left(\frac{1}{\beta} r(x,y)\right)$ is the partition function that ensures the policy probabilities sum to 1 for a given prompt x.

Intuition: This says the optimal policy is the reference policy, re-weighted by the exponentiated reward. High-reward completions y get their probabilities boosted, and low-reward completions get their probabilities suppressed.

3. Inverting the Relationship: From Policy to Reward

The previous step showed how to get the optimal policy from a reward function. DPO's magic is to flip this around. If we have an optimal policy π_r, what reward function *must have* produced it? We can rearrange the equation above to solve for $r(x,y)$:

$$\exp\left(\frac{1}{\beta} r(x,y)\right) = \frac{\pi_r(y|x)}{\pi_{ref}(y|x)} Z(x)$$

$$\frac{1}{\beta} r(x,y) = \log\left(\frac{\pi_r(y|x)}{\pi_{ref}(y|x)}\right) + \log(Z(x))$$

$$r(x,y) = \beta \log\left(\frac{\pi_r(y|x)}{\pi_{ref}(y|x)}\right) + \beta \log(Z(x))$$

This is the crucial step. It establishes a one-to-one mapping: every policy implies a unique reward function (up to a constant that depends only on x). This allows us to replace the unknown reward function $r(x,y)$ with a function of our learnable policy $\pi_\theta(y|x)$ and the fixed reference policy $\pi_{ref}(y|x)$.

4. Modeling Human Preferences with Bradley-Terry

RLHF data consists of pairwise preferences: for a prompt x, a human preferred completion y_w ("winner") over a rejected completion y_l ("loser"). The Bradley-Terry model is a standard way to model the probability of this preference:

$$p(y_w \succ y_l|x) = \sigma(r(x,y_w) - r(x,y_l)) = \frac{1}{1 + \exp(-(r(x,y_w) - r(x,y_l)))}$$

This says the probability of preferring y_w increases as the difference in their rewards grows.

5. Substituting the Implied Reward into the Preference Model

Now, we substitute our policy-defined reward from Step 3 into the Bradley-Terry model. Let's look at the difference in rewards:

$$r(x, y_w) - r(x, y_l) = \left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{ref}(y_w|x)} + \beta \log Z(x)\right) - \left(\beta \log \frac{\pi_\theta(y_l|x)}{\pi_{ref}(y_l|x)} + \beta \log Z(x)\right)$$

$$r(x, y_w) - r(x, y_l) = \beta \left(\log \frac{\pi_\theta(y_w|x)}{\pi_{ref}(y_w|x)} - \log \frac{\pi_\theta(y_l|x)}{\pi_{ref}(y_l|x)}\right)$$

Notice that the unknown partition function $Z(x)$ conveniently cancels out! Let's define the implicit reward estimate for a policy π_θ as $\hat{r}_\theta(x, y) = \beta \log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)}$.

The preference probability is now:

$$p(y_w \succ y_l|x) = \sigma(\hat{r}_\theta(x, y_w) - \hat{r}_\theta(x, y_l))$$

6. The Final DPO Loss

The goal is to find the policy π_θ that maximizes the likelihood of the observed human preferences. This is a standard maximum likelihood estimation problem. We want to maximize the log-probability of the data, which is equivalent to minimizing the negative log-likelihood:

$$\mathcal{L}_{DPO}(\pi_\theta; \pi_{ref}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log \sigma(\hat{r}_\theta(x, y_w) - \hat{r}_\theta(x, y_l))\right]$$

Substituting the definition of $\hat{r}_\theta$:

$$\mathcal{L}_{DPO}(\pi_\theta; \pi_{ref}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{ref}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{ref}(y_l|x)}\right)\right]$$

This is exactly the DPO loss. It's a simple binary cross-entropy loss where we are classifying which response is preferred. The "logits" for this classification are the difference in the implicit rewards.

Implementation and Pitfalls

In practice, the implementation is straightforward and resembles supervised fine-tuning.

```
def dpo_loss(policy_logits_w, policy_logits_l,
             ref_logits_w, ref_logits_l,
             beta=0.1):
    """
    Computes the DPO loss for a batch of winning and losing responses.
```

```
    Logits are assumed to be per-token, shape [batch_size, seq_len, vocab_size].
    """
    # Get log-probabilities of the actual sequences
    # These are sums over the sequence length
    pi_log_probs_w = get_sequence_log_probs(policy_logits_w, y_w) # shape: [batch_size]
    pi_log_probs_l = get_sequence_log_probs(policy_logits_l, y_l) # shape: [batch_size]

    ref_log_probs_w = get_sequence_log_probs(ref_logits_w, y_w) # shape: [batch_size]
    ref_log_probs_l = get_sequence_log_probs(ref_logits_l, y_l) # shape: [batch_size]

    # Calculate implicit rewards
    pi_ratio_w = pi_log_probs_w - ref_log_probs_w
    pi_ratio_l = pi_log_probs_l - ref_log_probs_l

    # The "logits" for the binary classification
    logits = pi_ratio_w - pi_ratio_l

    # Binary cross-entropy loss with a target of 1 (we want to prefer y_w)
    loss = -F.logsigmoid(beta * logits).mean()
    return loss
```

- **Numerical Stability:** The log-probability ratios can be large. However, since the loss is a `logsigmoid`, it's generally stable.
- **Reference Model:** The reference model `pi_ref` is kept frozen during training. This requires having two models in memory (the policy being trained and the reference), but only one requires gradients.
- **Hyperparameter β:** This parameter is crucial. It acts as an inverse temperature. If β is too high, the gradients can become very large, leading to instability. If it's too low, the learning signal is weak. Typical values are around 0.1 to 0.5.

> **Q: The GRPO algorithm simplifies PPO by replacing the learned value function with a z-score normalization of rewards. Why, mathematically, does this z-scoring introduce bias into the policy gradient estimate, and what are the practical consequences of this bias?**

This is a subtle but critical point that highlights the trade-offs between algorithmic simplicity and theoretical correctness. While GRPO's z-scoring is an effective heuristic, it violates a core assumption of policy gradient methods, leading to predictable and sometimes undesirable side effects.

1. The Role of a "Valid" Baseline in Policy Gradients

Let's start with the Policy Gradient Theorem. The gradient of the expected reward $J(\theta)$ is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\nabla_\theta \log \pi_\theta(\tau) \cdot R(\tau) \right]$$

where τ is a trajectory (a full response in our case) and $R(\tau)$ is its total reward.

To reduce the high variance of this estimate, we introduce a baseline $b(s)$ that depends on the state (prompt) but not the action (response). The gradient becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\nabla_\theta \log \pi_\theta(\tau) \cdot (R(\tau) - b(s)) \right]$$

This estimate remains **unbiased** (i.e., its expectation is still the true gradient) if and only if the baseline term has an expectation of zero:

$$\mathbb{E}_{\tau \sim \pi_\theta} \left[\nabla_\theta \log \pi_\theta(\tau) \cdot b(s) \right] = 0$$

Let's prove this condition. We can rewrite the expectation as an integral over trajectories:

$$\int \pi_\theta(\tau|s) p(s) \left(\nabla_\theta \log \pi_\theta(\tau|s) \cdot b(s) \right) d\tau ds$$

Using the log-derivative trick ($\nabla_\theta \pi_\theta = \pi_\theta \nabla_\theta \log \pi_\theta$):

$$= \int p(s) b(s) \left(\nabla_\theta \pi_\theta(\tau|s) \right) d\tau ds$$

$$= \int p(s) b(s) \nabla_\theta \left(\int \pi_\theta(\tau|s) d\tau \right) ds$$

Since $\int \pi_\theta(\tau|s) d\tau = 1$ (probabilities sum to 1), its gradient is zero: $\nabla_\theta(1) = 0$.

$$= \int p(s) b(s) \cdot 0 \cdot ds = 0$$

This holds as long as $b(s)$ does not depend on the trajectory τ (or the actions within it), allowing us to pull it out of the integral over τ.

2. How GRPO Violates the Baseline Condition

GRPO proposes an "advantage" estimate for a response i within a group G of responses to the same prompt:

$$\hat{A}_i = \frac{r_i - \text{mean}(r_1, \ldots, r_{|G|})}{\text{std}(r_1, \ldots, r_{|G|}) + \epsilon}$$

Here, the baseline for response i is not just the mean, but a function of the entire group's rewards. Let's focus on the denominator: $b_i = \text{std}(r_1, \ldots, r_{|G|})$.

The standard deviation is calculated from the rewards of *all* responses $\{r_1, \ldots, r_{|G|}\}$ in the group. Each reward r_j depends on the trajectory τ_j sampled to generate it. Therefore, the denominator $\text{std}(\ldots)$ is a function of the entire set of sampled trajectories $\{\tau_1, \ldots, \tau_{|G|}\}$.

When we compute the gradient for a single trajectory τ_i, the term $\nabla_\theta \log \pi_\theta(\tau_i)$ is multiplied by an "advantage" whose denominator depends on τ_j for all $j \neq i$. This coupling means the baseline is no longer independent of the actions, violating the condition for an unbiased estimate. The gradient for one sample is being scaled by a factor that depends on other random samples in the batch, introducing bias.

3. Practical Consequences of the Biased Gradient

This theoretical flaw has real, practical consequences during training:

- **Length-Based Reward Hacking:** The most significant issue is a bias related to response length. The policy gradient update for response i is roughly proportional to $\frac{r_i - \bar{r}}{\text{std}}$.
    - **If an answer is wrong ($r_i < \bar{r}$):** The model is incentivized to make the response *very long*. A longer response might not improve the reward r_i, but if it's generated with low probability, it can increase the variance of rewards in the group, thus increasing the denominator `std`. A larger denominator shrinks the negative advantage term, reducing the penalty. This encourages the model to "BS" at length when it doesn't know the answer.
    - **If an answer is right ($r_i > \bar{r}$):** The model is incentivized to make the response as *short as possible*. A shorter correct answer is often generated with higher probability, potentially reducing group variance and maximizing the positive advantage.

  This explains the phenomenon observed in models like DeepSeek-Coder, where RL training with GRPO leads to a dramatic and often unbounded increase in the length of Chain-of-Thought (CoT) reasoning, which may be an artifact of the biased objective rather than a sign of "deeper thinking."

- **Distorted Focus:** The `std` term in the denominator causes the algorithm to place more weight on prompts where the model's performance is inconsistent (high variance) and less weight on prompts where the model is either consistently right or consistently wrong (low variance). This can slow down learning on "easy" or "hopeless" problems.

In summary: GRPO's z-scoring is a clever, simple heuristic that provides a powerful adaptive baseline without needing a separate critic network. However, its theoretical flaw introduces a bias that can lead to undesirable behaviors like pathological length increases. Recent work has proposed fixes, such as removing the standard deviation term (recovering a simple baselined policy gradient) or applying length normalization to counteract the bias.

```
# Pseudocode illustrating the coupling in GRPO
def grpo_advantage(rewards_in_group):
    # rewards_in_group is a tensor of rewards for N responses to the same prompt
    mean_reward = rewards_in_group.mean()
    std_reward = rewards_in_group.std() # <-- Problem here!

    # The std_reward depends on ALL rewards in the group.
    # When calculating the gradient for one response, its "advantage"
    # is scaled by a term that depends on other random responses.
    advantages = (rewards_in_group - mean_reward) / (std_reward + 1e-8)
    return advantages

# In the training loop...
# for each prompt:
#   responses = model.generate(prompt, num_responses=N)
#   rewards = compute_rewards(responses)
#   advantages = grpo_advantage(rewards) # Biased!
#   for i in 1..N:
#     log_prob_i = compute_log_prob(responses[i])
#     loss += -log_prob_i * advantages[i].detach() # .detach() doesn't fix the bias
```

> **Q: The KL-divergence term appears in both the overall RLHF objective (e.g., DPO) and within the inner loop of policy gradient algorithms like PPO. What are the distinct roles of the KL term in these two contexts, and how does the choice of reference policy (π_{ref} vs. π_{old}) reflect these different goals?**

This is an excellent question that untangles a common source of confusion. The KL-divergence term is a multi-purpose tool in alignment, serving two distinct functions: one as a **global behavioral constraint** and the other as a **local update stabilizer**.

Role 1: Global Behavioral Constraint

- **Context:** This is the KL term in the main RLHF objective, which is optimized over the entire training run.

$$\max_{\pi_\theta} \mathbb{E}[r(x,y)] - \beta D_{KL}(\pi_\theta(y|x)||\pi_{ref}(y|x))$$

- **Goal:** To prevent the final, optimized policy π_θ from deviating too far from a trusted, high-quality initial policy. Its purpose is to preserve general capabilities and prevent "reward hacking" or "mode collapse." The model should get better at the specific task (e.g., math problems) without forgetting how to write coherent English, follow general instructions, or maintain its safety profile.
- **Reference Policy (π_{ref}):** This is typically the **Supervised Fine-Tuned (SFT) model**. The SFT model is chosen because it has already been trained on a diverse set of high-quality demonstrations and represents a good, general-purpose starting point. It embodies the desired style and broad capabilities we don't want to lose. This π_{ref} is **fixed** throughout the entire RLHF training process.
- **Timescale:** Global, across the entire training procedure.
- **Pitfall:** If the KL coefficient β is too low, the policy may over-optimize for the reward signal, leading to narrow, brittle, or nonsensical behavior that achieves a high score but is not generally useful (e.g., outputting only "The answer is 42." to every math problem). If β is too high, the policy will be too constrained and unable to learn from the reward signal, effectively never improving beyond the SFT model.

Role 2: Local Update Stabilizer

- **Context:** This KL term appears within the inner loop of on-policy RL algorithms like PPO and TRPO. These algorithms are based on importance sampling. They generate a batch of experience (rollouts) using a policy π_{old}, and then use that data to perform one or more gradient updates on the current policy π_θ.
- **Goal:** To ensure the stability of the gradient updates. The importance sampling ratio, $w_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)}$, is used to correct for the fact that the data came from a different policy. If π_θ becomes too different from π_{old}, this ratio can become very large or small, leading to extremely high variance in the gradient estimates and causing the training to diverge. The KL constraint keeps π_θ inside a "trust region" around π_{old} where the importance sampling estimate is reliable.

- **Reference Policy (π_{old}):** This is the **policy from the previous iteration**—the one used to generate the current batch of data. Unlike the global π_{ref}, this π_{old} is **updated frequently** (typically after every batch or set of gradient steps). It's a moving target that tracks the learning progress.
- **Timescale:** Local, within a single batch of updates.
- **Implementation:**
  - **TRPO** enforces this as a hard constraint: $D_{KL}(\pi_\theta || \pi_{old}) \leq \delta$.
  - **PPO** uses a simpler "clipping" mechanism on the probability ratio as a proxy for the KL constraint, which is computationally cheaper and often works just as well.

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min(w_t \hat{A}_t, \text{clip}(w_t, 1-\epsilon, 1+\epsilon)\hat{A}_t) \right]$$

Summary Table

Feature	Role 1: Global Behavioral Constraint	Role 2: Local Update Stabilizer
Purpose	Prevent catastrophic forgetting, preserve general capabilities.	Ensure stable gradient updates via importance sampling.
Algorithm Context	Overall RLHF objective (DPO, PPO's full objective).	Inner loop of on-policy algorithms (PPO, TRPO, GRPO).
Reference Policy	π_{ref} = SFT Model (fixed).	π_{old} = Policy from previous iteration (frequently updated).
Timescale	Global (entire training run).	Local (per batch of updates).
Effect	Keeps the final model "well-behaved" and generalist.	Prevents training from diverging due to high-variance gradients.

In the full PPO objective used by InstructGPT, both terms are actually present: one for the reward, one for the global KL penalty against the SFT model, and the clipping mechanism for local stabilization. DPO cleverly sidesteps the need for local stabilization by being an offline, supervised-style algorithm, so it only needs to consider the global KL constraint, which is baked directly into its loss function.

🛠️ Assignments — Do the Work

Study guides for the 5 implementation-heavy assignments — where the real learning is.

The ~22 hours of lectures are the easy part. The assignments are where you actually learn — and they take far longer. As Andrej Karpathy put it, real learning "is not supposed to be fun — the primary feeling should be that of effort," like a serious session at the gym rather than a 10-minute highlight reel. These notes and the Deep Dives are the warm-up and the map; the assignments are the gym. **So this section gives guidance, not solutions** — pointers to do the work yourself, not a way around it.

Jump to: A1: Basics · A2: Systems · A3: Scaling · A4: Data · A5: Alignment & Reasoning RL

Assignment 1: Basics

Official starter code & handout ↗ · due Apr 15, 2026 (2026 offering) · **No solutions here — by design.**

Before you start — lectures: Lec 01: Overview and Tokenization · Lec 02: PyTorch, Resource Accounting · Lec 03: Architectures, Hyperparameters | deep dives: Tokenization & Byte-Pair Encoding · Transformer Architecture Choices

You will implement the fundamental components of a modern language model from scratch and assemble them into a working system. The deliverables are:

1. **A Byte-Pair Encoding (BPE) Tokenizer:** You'll implement the BPE training algorithm to learn a vocabulary and merge rules from a text corpus. Your tokenizer will have `train`, `encode`, and `decode` methods. The specific style is based on GPT-2, which involves a pre-tokenization step before running BPE on word-level chunks.
2. **A Transformer Model:** You will build the decoder-only Transformer architecture as a `torch.nn.Module`. This involves implementing its constituent parts: multi-head self-attention (MHA) with Rotary Position Embeddings (RoPE), a SwiGLU feed-forward network, and RMSNorm for normalization. You will assemble these into Transformer blocks and stack them to form the full model.

3. **An AdamW Optimizer:** You will implement the AdamW optimizer by creating a custom class that inherits from `torch.optim.Optimizer`. This requires managing the optimizer's state (first and second moment estimates) for each model parameter and implementing the `step()` logic, including bias correction and decoupled weight decay.
4. **A Training Harness:** You will write a script that uses these three components to train a minimal language model on a real dataset (like TinyStories). This involves setting up the data loading, the training loop, calculating the cross-entropy loss, and calling your custom optimizer.

These pieces fit together to form a complete training pipeline: raw text is fed to your tokenizer to create integer sequences, which are batched and passed to your Transformer model. The model's output is used to calculate a loss, and your AdamW optimizer updates the model's weights to minimize this loss.

> **Why this is where the learning happens**

Watching lectures gives you a map, but implementing this assignment is the journey where you actually learn the terrain. Building these components from first principles forces you to move beyond "leaky abstractions" and confront the details that make these systems work. You will internalize how architectural choices (like RMSNorm vs. LayerNorm) are not just theoretical but have real consequences for speed and memory, and how an algorithmic choice in your tokenizer directly impacts the shape of your model's embedding matrix. This struggle is what solidifies the concepts and builds the engineering mindset required to innovate.

> **What makes it hard**

This assignment combines conceptual hurdles with non-trivial engineering. Don't underestimate it.

- **Algorithmic Complexity (BPE):** A naive BPE training implementation that re-scans the corpus for every merge is computationally infeasible ($O(\text{CorpusSize} \times \text{NumMerges})$). You must devise an efficient method to track and update pair frequencies without repeated scans, which requires careful data structure design. The encoding process also has a subtle greedy logic that must be implemented deterministically.
- **Tensor Gymnastics (Transformer):** The logic of multi-head attention involves a flurry of tensor reshaping, transposing, and batch matrix multiplications. Keeping track of

dimensions (`batch` , `seq_len` , `num_heads` , `head_dim`) is a notorious challenge. A single incorrect `transpose` or `view` can introduce silent bugs that are hard to trace. Implementing the causal mask and RoPE correctly requires precise control over tensor indices and dimensions.

- **Numerical Stability:** You are building a deep neural network where numbers can easily explode or vanish. The attention softmax, $softmax(\frac{QK^T}{\sqrt{d_k}})$, can overflow if the logits in the numerator are too large. The $\sqrt{d_k}$ scaling factor is not just a suggestion; it's critical for stable training. Your choice of normalization (pre-norm vs. post-norm) will dramatically affect gradient flow and training stability.
- **Memory Management:** A language model is a memory hog. For every parameter in your model, you must store not only its value (4 bytes for `float32`), but also its gradient (4 bytes), and the AdamW optimizer's state (8 bytes for the two moments). That's 16 bytes per parameter, *before* even considering the memory for activations, which scales with batch size and sequence length. You will quickly learn why techniques like mixed-precision training are essential.
- **Silent Bugs:** The most frustrating challenge is that your code might be "wrong" but still run without errors. An incorrect attention mask, a miscalculated gradient in your optimizer, or a bug in your BPE encoding might not cause a crash, but will simply lead to a model that refuses to learn. The loss will stagnate, and you won't know why.

How to approach it

Tackle this assignment by building and testing components incrementally. Do not try to write everything at once.

1. **Start with the Optimizer:** Implement your AdamW optimizer first. You can test it in isolation on a simple problem, like fitting a linear regression model. You can even write a unit test that performs a single step with your optimizer and asserts that the resulting parameter values match those produced by `torch.optim.AdamW` on the same inputs.
2. **Build the Tokenizer:**
    - First, focus on the `encode` and `decode` methods. Assume you are *given* a vocabulary and a list of learned merges. Can you correctly apply the merges to encode a string and then decode it back? Test your round-trip capability.
    - Next, implement the `train` method. Start with a tiny, one-sentence corpus. Manually trace the first few merges to ensure your pair counting and update logic is correct.

Then, scale up to the full training corpus.

3. **Construct the Transformer, Piece by Piece:**
    - Start with the innermost components. Implement `RMSNorm` and the `SwiGLU` feed-forward block. Write small tests to verify their output shapes.
    - Tackle self-attention. First, implement a *single* attention head without RoPE. Get the masking and scaled dot-product right.
    - Then, wrap this in a `MultiHeadAttention` module. This is mostly about managing shapes and combining the outputs of the individual heads.
    - Now, add `RoPE`. This is a modification to the queries and keys *before* they are used in the attention calculation.
    - Finally, assemble a single `TransformerBlock` using your MHA and FFN modules, along with the residual connections and normalization.
4. **Assemble and Test the Full Model:** Stack your `TransformerBlock`s to create the full model. Your goal is to achieve "loss-goes-down" on a single, tiny batch of data. Set up a minimal training loop with your model, a standard `CrossEntropyLoss`, and your custom AdamW optimizer. If you can overfit a single batch and drive the loss to near-zero, your entire pipeline is likely correct.
5. **Scale Up:** Once you've successfully overfit a single batch, you can move on to training on the full dataset and trying to achieve a competitive perplexity score.

Common pitfalls & debugging tips

- **Incorrect Attention Mask:** A classic bug. Your causal mask must prevent a position from attending to future positions. A common error is an off-by-one in the mask's dimensions or applying it incorrectly (e.g., adding `0` instead of `-inf` before the softmax). **Tip:** After computing your attention scores but before the softmax, extract a single attention matrix from your batch and print it. The upper triangle (above the diagonal) should be `-inf`.
- **Mismatched Tensor Shapes:** Use `assert` statements liberally to check the shapes of your tensors at every step, especially within the attention mechanism. Comments like `# shape: (batch, heads, seq_len, head_dim)` are good, but `assert x.shape == (b, h, s, d)` is better.
- **BPE Merge Ambiguity:** The encoding process must be deterministic. When multiple merges are possible (e.g., in "abc", you could merge 'ab' or 'bc'), the ambiguity is resolved by picking the merge that was learned earliest (has the lowest rank). If there's a tie, the leftmost occurrence in the sequence wins.

- **Weight Decay vs. L2 Regularization:** The 'W' in AdamW stands for decoupled weight decay. It is *not* the same as adding an L2 regularization term to the loss. Weight decay is applied directly to the weights during the optimizer step. Make sure you implement this distinction correctly.
- **Forgetting `.to(device)`:** A simple but common error. Your model, your optimizer's state, and your data must all live on the same device (CPU or GPU).
- **The Single-Batch Test is Your Best Friend:** If your model can't learn to perfectly memorize a single batch of data, it has no hope of learning from the entire dataset. If your loss isn't plummeting towards zero in this test, there is a bug in your model, loss calculation, or optimizer `step` function.
- **Check Gradients:** If you suspect your backward pass is wrong, you can inspect the `.grad` attribute of your model's parameters after `loss.backward()`. Are they `None`? Are they all zero? This can point to issues like detached parts of your computation graph.

Assignment 2: Systems

Official starter code & handout ↗ · due Apr 29, 2026 (2026 offering) · **No solutions here — by design.**

> **Before you start** — lectures: Lec 05: GPUs · Lec 06: Kernels, Triton · Lec 07: Parallelism 1 · Lec 08: Parallelism 2 | deep dives: GPU Kernels & FlashAttention · Parallelism

You will implement three key systems components for large-scale language modeling. First, you'll use profiling tools to benchmark a standard Transformer block and identify performance bottlenecks, particularly in the attention layer. Second, you will write a custom FlashAttention-2 forward and backward pass from scratch in Triton, a Python-based language for GPU programming. This involves implementing tiling, online softmax, and recomputation to minimize slow memory access. Finally, you will use PyTorch's Fully Sharded Data Parallelism (FSDP) to build a memory-efficient, multi-GPU training loop, allowing you to train models that wouldn't fit on a single device.

> **Why this is where the learning happens**

Watching a lecture on GPU architecture is like reading a map; implementing a custom kernel is like navigating the terrain yourself. You will directly confront the "memory wall"—the massive gap

between compute speed and memory bandwidth—that lectures can only describe. By manually managing data movement between slow HBM and fast on-chip SRAM, you will gain a physical intuition for why algorithms like FlashAttention are not just clever, but essential for modern AI. This struggle transforms abstract concepts like tiling and kernel fusion into concrete engineering skills.

What makes it hard

The primary challenge is bridging the gap between a high-level algorithm and a low-level, hardware-aware implementation.

- **Numerical Stability:** The "online softmax" trick in FlashAttention requires maintaining a running maximum and a sum of exponentials. A naive implementation can easily lead to `inf` or `NaN` values. The update rule when the running maximum changes, $l_{new} = e^{m_{old} - m_{new}} l_{old} + e^{s_i - m_{new}}$, involves careful scaling to prevent overflow/underflow.
- **Memory Management & Correctness:** In Triton, you are responsible for calculating memory offsets for loading "tiles" of your Q, K, and V matrices. An off-by-one error in your pointer arithmetic can lead to incorrect data being loaded, silent correctness failures, or hard crashes. You must also use masks correctly to handle sequences whose lengths are not perfect multiples of your tile size.
- **Conceptual Complexity of the Backward Pass:** The backward pass of attention is more complex than the forward pass. To save memory, FlashAttention recomputes attention scores from the forward pass instead of storing them. You must correctly derive and implement the gradients with respect to Q, K, and V, while re-materializing the necessary values on-the-fly within your tiled execution, all without access to the full $N \times N$ attention matrix.
- **Distributed Systems Complexity:** Setting up a distributed training job with FSDP involves managing process groups and understanding how model parameters, gradients, and optimizer states are sharded across devices. A common trap is a deadlock, where one GPU waits for another in a collective operation that never completes, causing the program to hang indefinitely.

How to approach it

Follow this roadmap to build your components incrementally, testing at each stage.

1. **Profile First:** Before optimizing, you must measure.

- Set up a simple benchmark for a single Transformer block using a standard PyTorch implementation.
- Use `torch.profiler` to generate a trace. Identify which operations (e.g., `aten::matmul`, `aten::softmax`) consume the most GPU time. This will confirm that the attention calculation is your bottleneck and motivate the need for a custom kernel.

2. **Implement the FlashAttention Forward Pass in Triton:**

- Start with the simplest case: a "naive" attention kernel in Triton that computes attention for a single row of Q against all of K and V, without tiling the sequence length dimension. Verify its correctness against `torch.nn.functional.scaled_dot_product_attention`.
- Now, implement the full tiled forward pass. Your outer loop will iterate over blocks of the K/V sequence. Your inner logic will process a block of Q against a block of K/V.
- Focus on the core "online softmax" state: what running statistics (the max value `m` and the normalization factor `l`) do you need to carry through the outer loop?
- Think carefully about how to update your output block `O` when you encounter a new maximum value. You'll need to rescale the existing `O` before adding the contribution from the new block.
- Test your tiled implementation for numerical correctness against the PyTorch reference on various matrix sizes, especially those that are not perfectly divisible by your tile size.

3. **Implement the FlashAttention Backward Pass:**

- First, ensure you are saving the correct, minimal set of values from the forward pass to global memory for use in the backward pass (hint: you need the output `O` and the final softmax statistics `L`).
- Structure your backward pass kernel to mirror the tiled structure of the forward pass. You will iterate through blocks of Q, K, and V to recompute the attention scores needed for the gradient calculation.
- Derive or look up the gradients `dO`, `dQ`, `dK`, `dV`. Implement the calculation for `dQ`, `dK`, and `dV` inside your tiled loop. Remember that `dK` and `dV` are accumulated across the blocks of Q.
- Again, test for numerical correctness at every step by comparing your kernel's output gradients to those produced by PyTorch's autograd on the reference attention function. Use `torch.allclose` with a reasonable tolerance.

4. **Build the Distributed Training System:**
    - Start with a standard model (like the one from your previous assignments) and a training loop that runs on a single GPU. Profile its peak memory usage.
    - Wrap your model with `torch.distributed.fsdp.FullyShardedDataParallel`. You will need to set up a `torch.distributed` process group.
    - Run the FSDP-wrapped training loop on multiple GPUs. Profile the peak memory usage *per GPU*. You should see a significant reduction compared to the single-GPU baseline, demonstrating that you've successfully sharded the model's state.

> **Common pitfalls & debugging tips**

- **Benchmarking Errors:** Always use `torch.cuda.synchronize()` before starting and stopping your timer. The CPU and GPU execute asynchronously, and failing to synchronize will result in you timing only the CPU's kernel launch time, not the actual GPU execution. Also, run several warm-up iterations before measuring.
- **Triton Pointer Math:** Use `print()` statements inside your Triton kernel (with `TRITON_INTERPRET=1`) to debug the offsets and pointers you are calculating. It's slow but invaluable for catching indexing errors.
- **Handling Edges:** Your code must handle sequence lengths that aren't a multiple of your `BLOCK_SIZE`. Use the `mask` argument in `tl.load` and `tl.store` to prevent out-of-bounds memory access. Forgetting this will lead to garbage results or crashes.
- **Numerical Stability in Softmax:** When implementing the online softmax, always subtract the current running maximum from your scores *before* calling `tl.exp()`. This prevents the exponentiation from producing `inf`.
- **Check Against a Reference:** The most powerful debugging tool you have is a correct reference implementation. At every stage of your Triton kernel development (forward and backward), compare your output tensor against the output of `torch.nn.functional.scaled_dot_product_attention` using `torch.allclose`. If they don't match, they should be very close.
- **Backward Pass State:** For the backward pass, you only need to have stored the final `O`, `L`, and `dO` in global memory. Everything else (like the `S` matrix) must be recomputed on the fly from `Q`, `K`, and `V`. If you find yourself trying to save the `S` matrix, you've missed the point of recomputation.
- **FSDP Hangs:** If your multi-GPU training script hangs, it's almost always a deadlock. This means one or more processes have entered a collective communication call (like

`all_reduce`) while others have not. Ensure that all ranked processes are executing the exact same code path that involves collective calls.

Assignment 3: Scaling

Official starter code & handout ↗ · due May 6, 2026 (2026 offering) · **No solutions here — by design.**

Before you start — lectures: Lec 09: Scaling Laws 1 · Lec 11: Scaling Laws 2 | deep dives: Scaling Laws & compute-optimal training

You will write a suite of Python scripts to conduct a small-scale replication of the Chinchilla scaling law experiments. Your goal is to determine the compute-optimal way to train a Transformer model. You will not be building the model or trainer itself; instead, you'll use a provided API that lets you train a model of a given size (`N` parameters) on a given amount of data (`D` tokens) and returns the final validation loss.

Your deliverables will be: 1. **An Experiment Runner:** A script that systematically calls the training API for various combinations of model sizes (`N`) and dataset sizes (`D`) to gather loss data. 2. **An IsoFLOP Analyzer:** A tool that implements the "IsoFLOP" method. For a *fixed* compute budget `C`, this tool will train several `(N, D)` configurations that all cost `C` FLOPs and identify the optimal `N` that minimizes loss. 3. **A Scaling Law Fitter:** A script that takes the optimal points from multiple IsoFLOP analyses and fits power laws to them. Specifically, you will find the exponents `a` and `b` in the relationships $N_{opt}(C) \propto C^a$ and $D_{opt}(C) \propto C^b$. 4. **A Predictor:** A final function that uses your fitted scaling laws to predict the optimal model size and number of training tokens for a large, unseen training budget.

These components will work together to transform raw experimental data into a predictive engineering tool.

> **Why this is where the learning happens**

Watching a lecture on scaling laws can make the concepts seem abstract and the results pre-ordained. This assignment forces you to confront the messy reality of generating the data yourself. You will gain a visceral understanding of the trade-off between model size and data by plotting the U-shaped IsoFLOP curves from your own experiments. Actually fitting the power-law

functions to your noisy, hard-won data points will solidify why log-log plots are so essential and how a few small-scale runs can yield powerful predictions about massive models.

What makes it hard

The primary challenge is not in writing complex code, but in correctly orchestrating the experiments and interpreting the results.

- **Conceptual Complexity:** The core of the assignment is the IsoFLOP method (Chinchilla's "Method 2"). You must internalize the compute formula, $C \approx 6ND$, and understand that for a fixed `C`, `N` and `D` are inversely proportional. You are searching for the minimum of a U-shaped loss curve that exists along this hyperbola.
- **Managing Experimental Sweeps:** You will be running dozens of training jobs. Keeping track of the parameters (`N`, `D`, `C`) and their resulting `loss` requires careful organization. A single mislabeled data point can corrupt your analysis.
- **Curve Fitting Instability:** You will likely use a library function like `scipy.optimize.curve_fit` to fit your scaling laws. These numerical optimizers can be sensitive. If you provide a poor functional form or bad initial guesses for the parameters, the fit can fail to converge or produce nonsensical results. The challenge is in understanding the expected shape of the data to guide the fitting process.
- **Correctness of Measurement:** As the notes explain, the way you measure loss is critical. A loss value for a given `(N, D)` pair is only valid if it comes from a model of size `N` trained from scratch for `D` tokens with an appropriate learning rate schedule. You cannot simply take intermediate checkpoints from one long run; this will produce biased data and lead to incorrect scaling exponents.

How to approach it

Follow this roadmap to build your solution incrementally and ensure each part is working before moving to the next.

1. **Master the Training API:** Before anything else, figure out how to run a single experiment. Call the provided training function with a fixed `N` and `D` and verify that you can successfully retrieve the final validation loss.

2. **Run a Simple Scaling Sweep:** To build intuition, start with a one-dimensional analysis.

- Fix a model size `N`. Run experiments for several different data sizes `D`.
- Plot `log(loss)` vs. `log(D)`. Do you see a roughly linear trend? This is your first scaling law!

3. **Implement the IsoFLOP Analysis for a Single Budget:** This is the heart of the assignment.
    - Pick one, medium-sized compute budget `C` (e.g., `1e21` FLOPs).
    - Write a helper function that generates 5-7 pairs of `(N, D)` that all satisfy the compute constraint $C = 6ND$. Choose your `N` values to be spaced logarithmically.
    - Run the training for each of these pairs and save the results.
    - Plot `loss` vs. `N` (with `N` on a log scale). You should see a U-shaped curve.
    - Find the optimal model size, `N_opt`, that corresponds to the minimum of this curve.
4. **Scale Up the Analysis:** Now, repeat the process from step 3 for several different compute budgets `C`. Choose budgets that are an order of magnitude apart (e.g., `1e20`, `1e21`, `1e22` FLOPs). At the end of this step, you should have a list of optimal points: `(C_1, N_opt(C_1), D_opt(C_1))`, `(C_2, N_opt(C_2), D_opt(C_2))`, etc.
5. **Fit the Final Power Laws:** With your list of optimal points, you can now determine the scaling exponents.
    - Plot `log(N_opt)` vs. `log(C)`. The data should look like a line. Perform a linear regression to find the slope, which is your exponent `a`.
    - Do the same for `log(D_opt)` vs. `log(C)` to find the slope `b`.
6. **Make Your Prediction:** Use the fitted laws ($N_{opt} \propto C^a$, $D_{opt} \propto C^b$) to calculate the optimal `N` and `D` for the final, large compute budget specified in the assignment prompt.

Common pitfalls & debugging tips

- **"My IsoFLOP curve isn't U-shaped!"**
    - Check your range of `N` values for a fixed `C`. If they are too close together, you may only see one side of the "U". Your `N` values should span at least two orders of magnitude.
    - Verify your compute calculation. For each `(N, D)` pair, assert that `6 * N * D` is very close to your target `C`. A simple mistake here is a common source of error.

- **Curve fitting is failing or giving weird results.**
    - Don't fit the complex Chinchilla loss function directly unless you have to. For finding the exponents `a` and `b`, it's much more robust to work in log-space and fit a simple line (`y = mx + c`).
    - If you must fit a non-linear function, provide good initial guesses (`p0`) to the optimizer. For example, the irreducible error term `E_ir` must be lower than any loss you've observed.
- **"My predicted exponents are very different from Chinchilla's."**
    - This is possible! Your small-scale experiment may have different characteristics. However, first rule out errors.
    - The most critical error to avoid: **Do not use intermediate checkpoints from a single long run.** Each `(N, D)` data point *must* come from its own independent training run to be valid.
    - To find the minimum of the U-shaped IsoFLOP curve, don't just take the lowest point you sampled. It's more accurate to fit a simple parabola to the `(log(N), loss)` points and calculate the true minimum of that parabola. This smooths out experimental noise.
- **Forgetting the constants.**
    - The compute formula is $C \approx 6ND$. Don't forget the `6`. Write a helper function for this calculation and use it everywhere to ensure consistency.
- **Plotting on the wrong scale.**
    - Scaling laws appear as straight lines on **log-log plots**. If your data looks curved, double-check that both your x and y axes are logarithmic. Use `plt.loglog()` or `np.log()` on your data before passing it to `plt.plot()`.

Assignment 4: Data

Official starter code & handout ↗ · due May 20, 2026 (2026 offering) · **No solutions here — by design.**

Before you start — lectures: Lec 13: Data 1 · Lec 14: Data 2

You will construct a multi-stage data processing pipeline to transform raw, messy web data from Common Crawl into a clean, deduplicated, and filtered dataset suitable for pretraining a language model. This is not a single program, but a series of connected components that act like a factory assembly line for data.

Your pipeline will consist of these core stages: 1. **Extractor:** A tool that reads raw web data in WARC (Web ARChive) format, parses the HTML, and extracts the main textual content, discarding boilerplate like navigation bars, ads, and scripts. 2. **Filter:** A module that applies a series of cleaning and filtering rules. This includes language identification to keep only English documents and quality heuristics to remove low-quality or nonsensical text (e.g., documents with too little content, repetitive phrases, or "lorem ipsum" placeholder text). 3. **Deduplicator:** A scalable system to identify and remove near-duplicate documents. This is the most complex component, as a naive document-to-document comparison is computationally impossible at this scale. You will implement an efficient, approximate algorithm to find these duplicates.

The final output will be a collection of clean text files, significantly smaller but much higher in quality than the original raw data, ready to be tokenized and fed into a model.

Why this is where the learning happens

Lectures can introduce you to concepts like "C4" or "RefinedWeb," but building your own data pipeline forces you to confront the messy reality they abstract away. You will learn firsthand that "web data" is not a clean corpus but a chaotic mix of high-quality prose, machine-generated spam, and everything in between. The true challenge—and the real learning—is in making principled, pragmatic trade-offs between processing speed, memory usage, and data quality at a scale where brute-force solutions fail.

What makes it hard

This assignment combines data engineering, algorithmic thinking, and heuristic design. The primary challenges are not in the complexity of any single line of code, but in making the entire system work correctly and efficiently at scale.

- **Scale and Memory:** A single Common Crawl dump is many terabytes. Even a small fraction is too large to fit into a single machine's memory. You cannot load the whole dataset to process it; you must design a streaming pipeline where documents are processed one by one or in small batches, without assuming you can see all the data at once.
- **Algorithmic Complexity of Deduplication:** The core of deduplication is finding pairs of similar documents. A naive approach would compare every document with every other document, an $O(N^2)$ operation that is computationally infeasible for millions of documents. The key challenge is to replace this with a near-linear time algorithm. This requires understanding and implementing techniques like:
  - **Shingling:** Representing a document as a set of n-grams (its "shingles").
  - **MinHash:** Creating a small, fixed-size "signature" from the infinite set of shingles, with the special property that the probability of two signatures matching is proportional to the Jaccard similarity of the original documents: $P(h_{min}(A) = h_{min}(B)) = \frac{|A \cap B|}{|A \cup B|}$.
  - **Locality Sensitive Hashing (LSH):** A technique to find documents with similar MinHash signatures without comparing all pairs. It works by hashing signatures into buckets such that similar signatures are likely to land in the same bucket.
- **Heuristics and "Correctness":** There is no single "correct" way to clean web text. HTML-to-text extraction is inherently lossy. Quality filters (e.g., "must have 5 sentences" or "must not contain bad words") are brittle heuristics. You will constantly face design decisions with no perfect answer, forcing you to define what "quality" means for your dataset and accept the trade-offs. For example, a filter that removes pages with `{` might improve quality by removing code, but it also prevents you from training a model that can code.

> **How to approach it**

Tackle this assignment by building and testing your pipeline incrementally. Do not attempt to run on a large dataset until you are confident in each component.

1. **Start Small:** Begin with a single, small WARC file. Your first goal is to write a script that can read this one file, extract plain text from the HTML payloads, and print it. Don't worry about quality yet; just get text out. Manually inspect the output. Does it contain navigation links, cookie banners, or JavaScript?
2. **Build the Filter Stage:**
   - First, integrate a language identification library (like a pre-trained `fastText` model) into your script. Process your single WARC file and verify that you can correctly label and filter out non-English documents.

    - Next, add a series of simple, rule-based quality filters inspired by datasets like C4. For example, filter out documents that are too short, have a low alphabet-to-character ratio, or contain placeholder text. Test each heuristic on your sample file and observe what it discards.
3. **Implement the Deduplication Engine (On a Small Scale):** This is the most challenging part. Break it down.
    - **Shingling & MinHashing:** Write a function that takes a single document's text and produces its MinHash signature. To test this, create two nearly identical documents and one very different one. Verify that the MinHash signatures of the similar documents are themselves similar, while the third is different.
    - **LSH for Candidate Pairs:** Implement the LSH banding technique. Your goal is a function that takes a batch of MinHash signatures and outputs *candidate pairs* for full comparison. The key insight is that you only need to compare documents that hash to the same bucket in at least one LSH band. This is what lets you avoid the $O(N^2)$ nightmare.
    - **End-to-End Deduplication:** Connect the pieces. For a small collection of documents (e.g., from one WARC file), generate all MinHash signatures, use LSH to find candidate pairs, and then for those pairs *only*, compute the true Jaccard similarity to make a final duplication decision.
4. **Integrate and Scale:** Combine your extractor, filter, and deduplicator into a single pipeline. A major engineering challenge here is managing the state needed for deduplication. You can't hold all MinHash signatures in memory. How can you process files sequentially while still comparing a new document to all *previous* documents? Think about how LSH bucketing can be implemented in a streaming or multi-stage fashion. Once your pipeline is robust on one file, test it on ten, then one hundred, refining your approach to handle the scale.

> **Common pitfalls & debugging tips**

- **My deduplication is too slow.** You are almost certainly falling back to a pairwise comparison. The entire point of LSH is to avoid this. Ask yourself: "Am I iterating through every document to compare it with the current one?" If so, you need to rethink your LSH implementation. Your goal is to only retrieve documents from a *specific hash bucket*.
- **My memory usage explodes.** You are trying to keep too much data in memory. Are you storing the full text of every document? The set of shingles for every document? You should only need to keep the compact MinHash signatures (and their LSH hashes) in a central data

structure. For exact-duplicate detection (a useful first pass), consider using a memory-efficient probabilistic data structure like a Bloom Filter.

- **Deduplication finds too many/too few duplicates.** This is a tuning problem. The "sensitivity" of your LSH is controlled by the number of hash functions per signature (k), the number of bands (B), and the number of rows per band (R). What is the relationship between these parameters and the Jaccard similarity threshold at which two documents are likely to be flagged as a candidate pair? Try to reason about the probability $1 - (1 - s^R)^B$ without just plugging in numbers. If you're finding too few duplicates, you need to make it *easier* for documents to collide in a bucket (e.g., fewer rows per band).
- **The extracted text quality is poor.** HTML parsing is messy. Don't try to write your own parser with regex. Use a robust library like `trafilatura`. Even then, inspect your "bad" outputs. Is the library failing to find the main content block? Is it grabbing comment sections? You may need to add post-processing heuristics on top of the library's output.
- **Sanity-check your components in isolation.** Before building the full pipeline, test each part on a tiny, handcrafted dataset of 5-10 documents. For example, create `doc1.txt`, `doc2.txt` (a near-duplicate of 1), `doc3.txt` (in another language), `doc4.txt` ("asdf qwer"), and `doc5.txt` (a good, unique document). Run them through each stage and assert that the output is exactly what you expect. This will save you hours of debugging on a multi-gigabyte corpus.

Assignment 5: Alignment & Reasoning RL

Official starter code & handout ↗ · due Jun 3, 2026 (2026 offering) · **No solutions here — by design.**

Before you start — lectures: Lec 15: Alignment — SFT/RLHF · Lec 16: Alignment — RL 1 · Lec 17: Alignment — RL 2 | deep dives: Alignment math

You will implement a pipeline to improve a language model's mathematical reasoning ability. This involves two main stages, mirroring the modern alignment process:

1. **Supervised Fine-Tuning (SFT) Model:** You'll start with a pre-trained base model and fine-tune it on a dataset of math problems and their correct solutions (expert demonstrations). This creates a specialized model, `SFT_model`, that is better at math than the base model. This `SFT_model` will also serve as the crucial reference policy (π_{ref}) in the next stage.

2. **Direct Preference Optimization (DPO) Model:** You will take your `SFT_model` and further train it using a preference dataset. This dataset contains pairs of responses to a math problem: one "winning" response (y_w) and one "losing" response (y_l). You will implement the DPO algorithm to train the model to prefer generating winners over losers. The final output is a `DPO_model` that is aligned with the preference data, hopefully making it an even better math reasoner.

The core deliverable is the implementation of the SFT and DPO training loops and the resulting models, evaluated on their ability to solve math problems.

> **Why this is where the learning happens**

Lectures explain that DPO is a clever way to do reinforcement learning with a simple classification loss, but only by implementing it do you truly grasp the mechanics. You will confront the engineering reality of managing multiple models (the policy and the frozen reference), calculating sequence-level log-probabilities, and debugging a process where the "right" answer isn't just a label, but a preference. This struggle forces you to connect the abstract math of the DPO derivation to concrete PyTorch operations, which is where deep understanding is forged.

> **What makes it hard**

This assignment moves beyond standard supervised learning into the more complex world of policy optimization. The challenges are both conceptual and technical.

- **Managing Multiple Models:** The DPO algorithm requires simultaneous access to the policy you are training (π_θ) and a frozen reference policy (π_{ref}, your SFT model). Keeping these two models in memory, ensuring one is frozen while the other trains, and correctly routing data through them is a significant engineering hurdle.
- **Correctly Calculating Log-Probabilities:** The DPO loss is a function of the log-probabilities of entire sequences. You need to write a function that, given a model and a sequence of tokens, correctly computes $\log P(y|x) = \sum_{i} \log P(y_i | y_{
- **The DPO Loss Itself:** The loss function involves a difference of log-probability ratios. The full expression is:

$$\mathcal{L}_{DPO} = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{ref}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{ref}(y_l|x)}\right)\right]$$

Translating this equation into efficient, stable code is the main challenge. You must perform four forward passes (policy on y_w, policy on y_l, reference on y_w, reference on y_l) and combine the resulting log-probabilities correctly.

- **Debugging is Non-Intuitive:** When a standard classification model fails, you can inspect the labels. Here, the "label" is a preference. If your DPO loss isn't decreasing, is it a bug in your log-prob calculation, an issue with the reference model, or a poor choice of the hyperparameter β? The feedback loop is much less direct than in SFT.

How to approach it

Follow this sequence. Do not move to the next step until the current one is working and verified.

1. **Implement and Verify SFT:** First, focus entirely on Supervised Fine-Tuning. Take the base model and fine-tune it on the provided math problems and their solutions. Your goal is to create a model that is demonstrably better at math than the base model. Before moving on, generate some samples from your SFT model and evaluate its accuracy on a held-out test set. This SFT model is your deliverable for the first part and becomes the essential π_{ref} for the second.
2. **Build the DPO Data Pipeline:** Create a `Dataset` and `DataLoader` for the preference data. Each item in your batch should contain a tokenized prompt, a tokenized winning response, and a tokenized losing response. Pay close attention to padding and creating the correct attention masks for each of the three components.
3. **Implement the Log-Probability Helper:** Write and isolate a helper function that takes a model, a batch of prompts, and a batch of corresponding completions, and returns the sum of the log-probabilities for each completion in the batch. Test this function thoroughly. Can it handle variable-length sequences? Does it correctly ignore the prompt tokens in its calculation?
4. **Assemble the DPO Training Step:** With your data pipeline and log-prob helper ready, you can now write the main DPO training logic.
    - Load your SFT model twice: once as the policy model to be trained, and once as the reference model (remember to freeze this one!).
    - In your training loop, for each batch, get the log-probabilities for both the winning and losing responses from *both* the policy and reference models.
    - Use these four sets of log-probabilities to compute the DPO loss according to the formula.
    - Perform backpropagation and update the policy model's weights.

5. **Train and Evaluate:** Run the DPO training. Monitor not only the DPO loss but also the model's math accuracy on the validation set. A decreasing DPO loss means you are successfully learning the preferences, but only an increase in math accuracy tells you that this alignment is achieving the ultimate goal.

Common pitfalls & debugging tips

- **Reference model is not actually frozen.** Your reference model's parameters should not be passed to the optimizer, and you should call `.eval()` on it. A simple check: verify that the weights of your reference model are not changing after a training step.
- **Incorrect log-probability calculation.** This is the most common bug. The log-probability should be calculated *only* for the tokens in the completion (`y`), not the prompt (`x`). Ensure your attention mask correctly separates the two and that you are summing the log-probs of the correct tokens.
- **Forgetting the `beta` hyperparameter.** The DPO loss is scaled by a temperature parameter `beta` . If you forget it, or set it incorrectly, your training may be unstable or ineffective. What does a very high or very low `beta` imply about the loss?
- **Sanity-check your initial loss.** Before training, the policy model (π_θ) is identical to the reference model (π_{ref}). Therefore, the log-probability ratios should be 1, and their logs should be 0. The argument to the sigmoid function will be 0. What should the value of $-\log(\sigma(0))$ be? If your initial loss is not close to this value, something is wrong in your loss calculation.
- **Overfit on a single batch.** A classic debugging technique. Take a single batch of preference pairs and train on it for many iterations. Your DPO loss should go to zero (or very close). If it doesn't, you have a bug in your gradient flow or loss computation.
- **Monitor the components of the loss.** Don't just watch the final loss value. Log the average log-prob ratios for winners and losers: $(\log \pi_\theta(y_w|x) - \log \pi_{ref}(y_w|x))$. As training progresses, you should see this term become positive for winners and negative for losers. This confirms the model is learning to increase the relative probability of preferred responses.