Generative AI with Large Language Models

W1 - Introduction to LLMs and the generative AI project lifecycle

Generative AI - is general purpose technology. Like Deep Learning
Transformer allowed attention to work in a parallel way that kicked it off.

Generative Models project life cycle

1. Take foundation of the shelf
2. Or pre training own model

Big vs Small model

1. Big model, for general purpose task
2. For single task, smaller model may work Actually use smaller model

Foundation models exhibit emergent properties beyond language alone. Research unlocking their capability for a. Complex task reasoning b. Problem solving
Parameters like model memory. More parameters, more sophisticated tasks it can perform.
Subjective understanding of language also increases with increasing model size.

But smaller models can be fine tuned to do more specific tasks.

Foundation models / Base models

- BERT - 110M
- GPT
- LLaMa
- **Flan-T5**
- PaLM
- BLOOM -176B

Definitions:

- Prompt - input text to model
- Context window - space or memory available to the model. Typically few 1000 words
- Completion - output of the mode. Comprise of question with generated text
- Inference - act of generating text

LLM based on Next word prediction

Task for LLM

- Chatbot
- Essay Writer
- Summarize
- Language translation
- Code generation
- Entity extraction (e.g., identify places and improvement)

- Connecting with external data source - flight information

Can use LLM to connect to external data sources or external APIs. Provides model information it does not have in the pre-training.

Text Generation before Transformers : RNN - which were limited by the amount of computation and memory required for generative tasks. Difficult to scale.

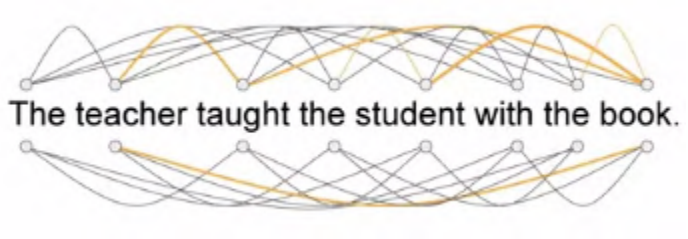
Challenges of Language:

- Homonyms - one word, multiple meanings. “bank” - financial institution or river side
    - Context of words help understand it’s meaning
- Syntactic ambiguity
    - Teacher taught the student with the book. Teacher’s book or student’s book?

Transformers

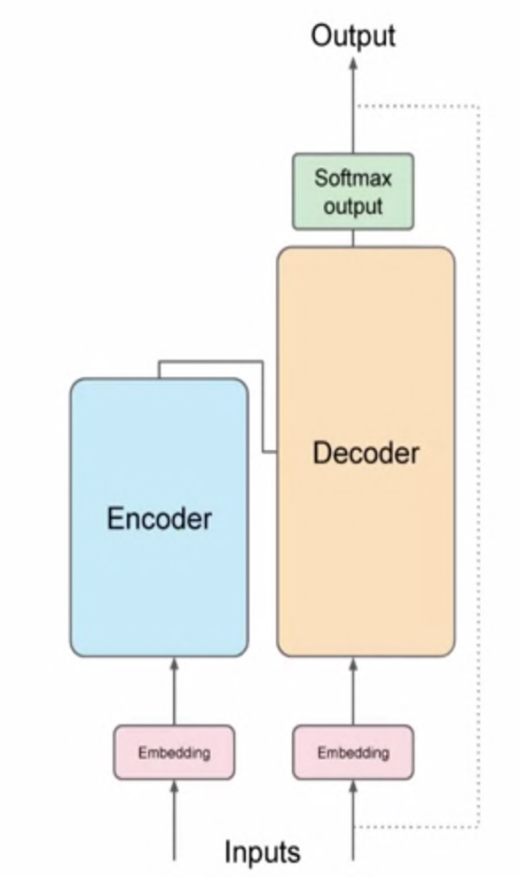
Transformers advantage:

- Scaled efficiently to use multi-core GPU
- Parallel process input data - making use of much larger dataset
- Attention to input meaning
    - Learns relevant and context of all the words in a sentence by applying attention weights, no matter where they are in the input



LLM using transformer architecture improved the performance of natural language task over earliers

ML models are big statistical calculators that work with numbers.



Architecture comprises 2 parts.

Encoder Decoder

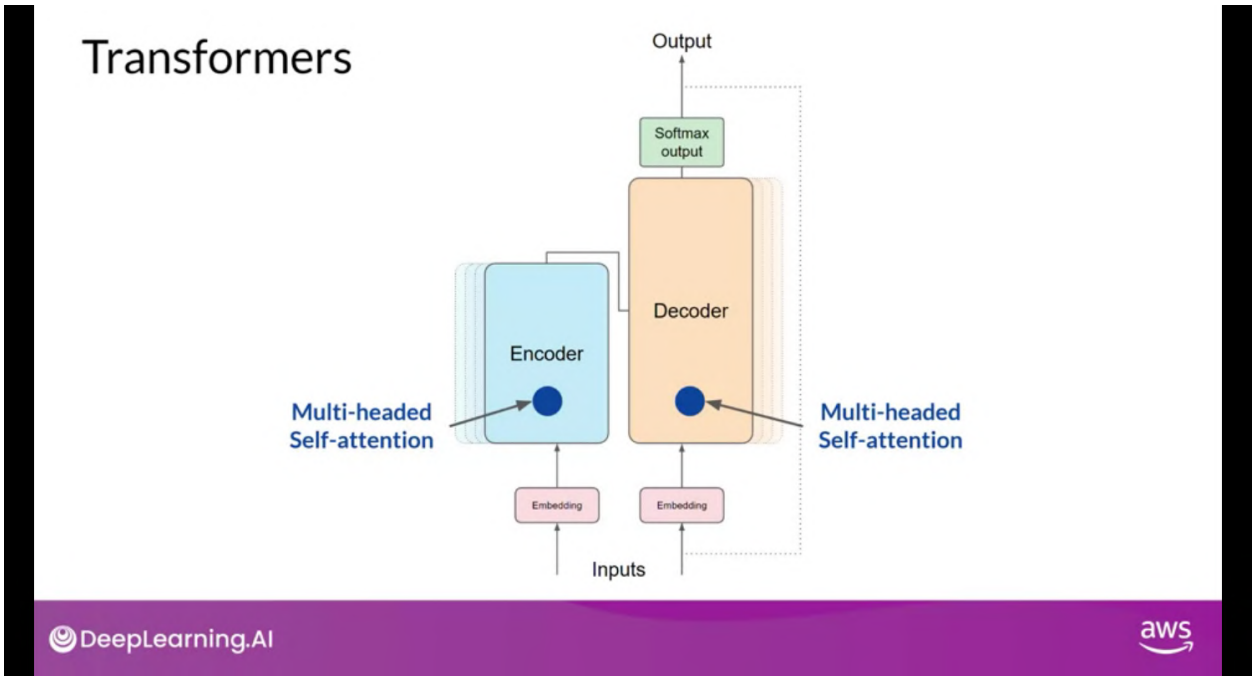
Tokenization - Tokenize the word to number before feeding into the model. Use the same tokenizer for training and inference. Each word mapped to a token ID or each sub word is mapped to a token ID.

Embedding - passed after tokenizing. Pass to the embedding layer. This is a trainable vector embedding space, high dimensional space where each token is represented as a vector and occupies a unique location within that space.

Embedding Vector Space - Each token ID in vocabulary is matched to a multi-dimensional vector. These vectors learn to encode the meaning and individual tokens in the input sequence.
Each token ID is mapped to a vector
Widely used in NLP. Previously used algorithms are word2vec.
In the original transformers paper, the vector size is 512.

Angle between the words in high dimensions vector helps the model understand the language

Position encodings - model processes each input token in parallel, by adding positional encoding you preserve the information about the word order and don't lose the relevance of the position of the word in that sentence.



Once summed the input tokens and the position encodings, pass the resulting vector into the self attention layer. Here the model analyzes the relationship in the token in the input sequence. Self-attention weights are learned during the training and are stored in the attention layers, reflecting the importance of each word in the input sequence to all other words in the sequence.

Multi-headed attention - multiple sets of self attention weights are learned in parallel, independent of each other. The number of attention heads included in the attention layer. Number in the range of 12-100. Each self attention head will learn different aspects of language. Weights are randomly assigned.

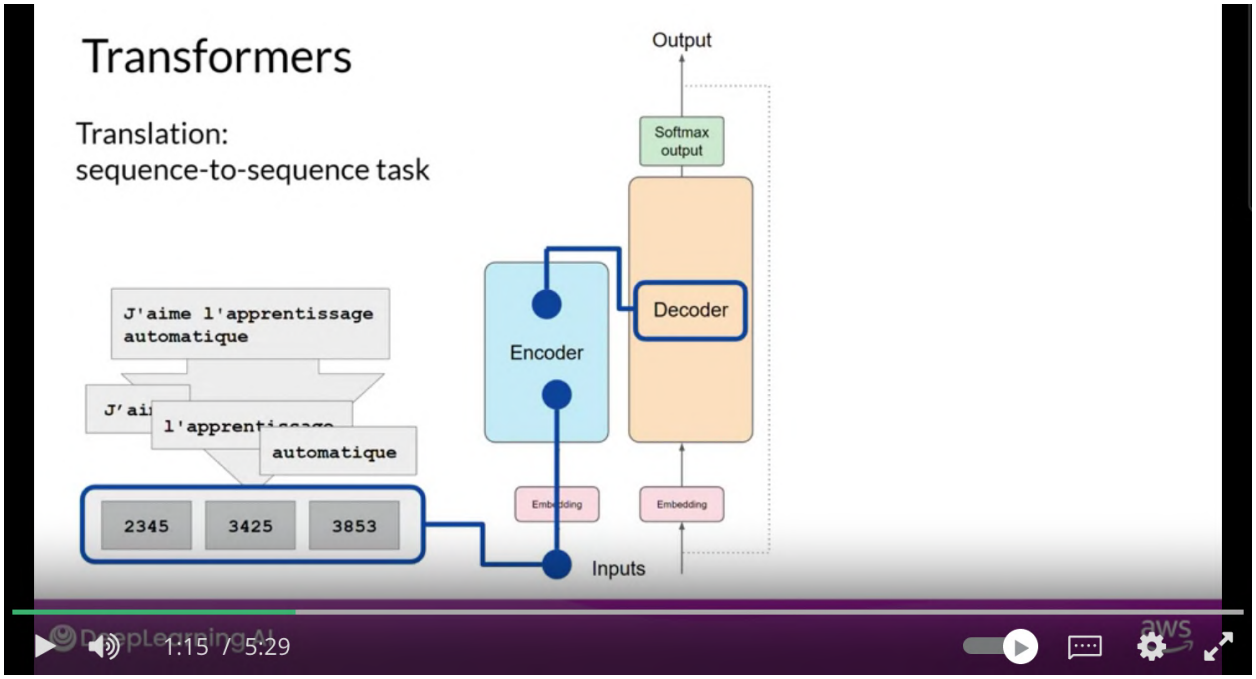
Some attention maps are easy to interpret, others may not be.

Feed Forward network - after attention weights applied to the input data, the output is processed through a Fully connected Feed Forward network. The output of FFN is a vector of logits proportional to each and every token in the tokenizer dictionary. Pass these logits to a softmax layer, where they are normalized into a probability score for each word. This output included probability for every single word in the vocab. The token which has the highest score is the most likely predicted token.

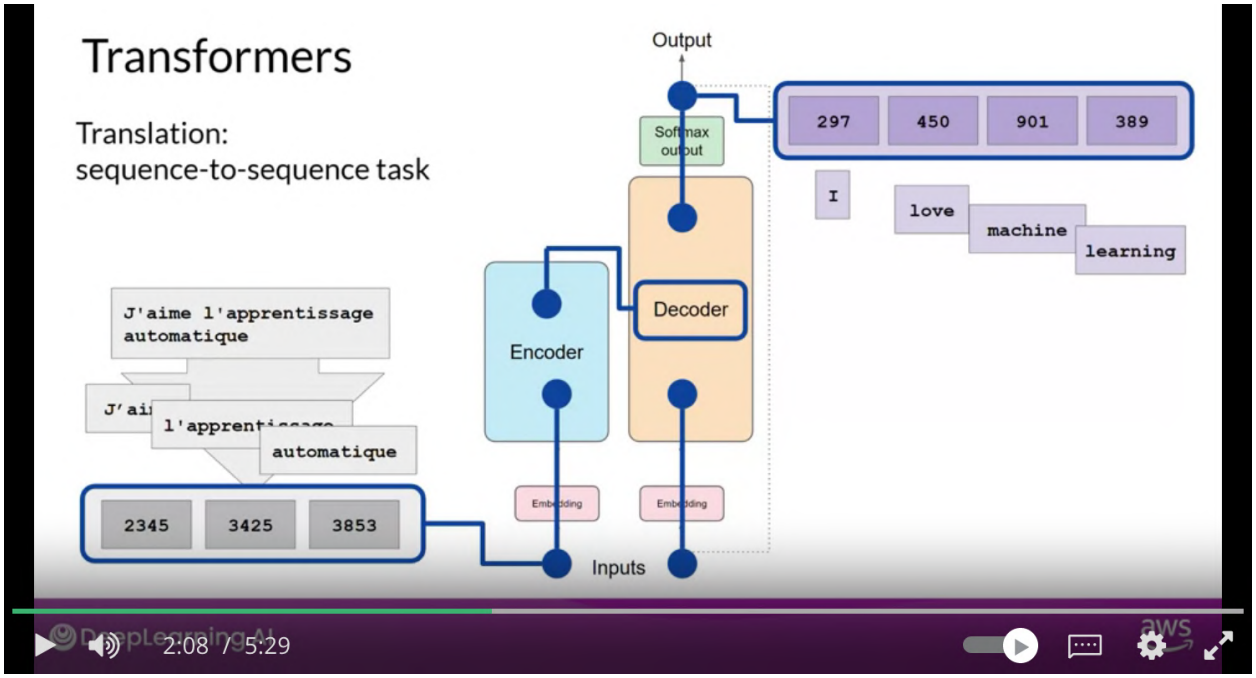
Study Notes: Building Large Language Models with the Transformer Architecture

- The transformer architecture has significantly improved the performance of natural language tasks compared to previous generation models like RNNs.
- The power of the transformer architecture lies in its ability to learn the relevance and context of all the words in a sentence, not just neighboring words.
- Attention weights in the transformer architecture enable the model to learn the relevance of each word to every other word in the sentence.
- Attention weights are learned during the model's training, and they can be visualized using an attention map to illustrate the relationships between words.
- The transformer architecture consists of two parts: the encoder and the decoder, which work together and share similarities.
- Tokenization is the process of converting words into numbers, with each number representing a position in a dictionary of words that the model can work with.
- The embedding layer maps token IDs to high-dimensional vectors that encode the meaning and context of individual tokens in the input sequence.
- Positional encoding is added to preserve information about the word order in the input sequence.
- Self-attention layers analyze the relationships between tokens in the input sequence, allowing the model to capture contextual dependencies.
- Multi-headed self-attention involves learning multiple sets of attention weights in parallel, where each attention head focuses on different aspects of language.
- The output of the self-attention layer is processed through a fully-connected feed-forward network.
- The output of the feed-forward network is a vector of logits, representing the probability scores for each token in the tokenizer dictionary.
- The logits are passed through a softmax layer to normalize them into probability scores for each word.
- The word with the highest probability score is the most likely predicted token.
- Various methods can be used to select the final token from the probability scores.

Generating Text with Transformers

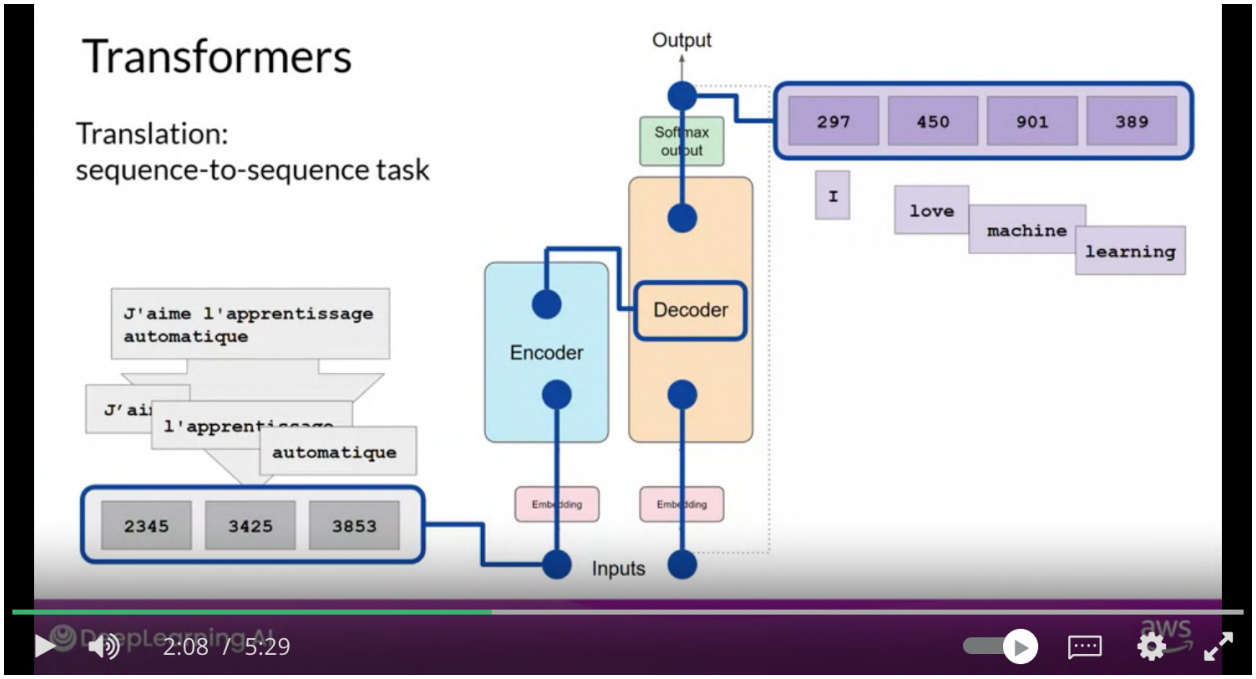


Data that leaves an encoder has learned a deep representation of the structure, meaning of the input sequence.
This representation is inserted into the middle of the decoder to influence decoder's self attention mechanism.
Next, a start of sequence token is added to the input of the decoder. This triggers the decoder to predict the next token, which it does based on the contextual understanding which is being provided by the encoder.
The output of the decoder's elf attention layers gets passed through decoder feed forward network and and through a final softmax layer.



The output is passed to input to trigger the generation of the next token. Continue this loop passing output back to the input until the model generates end of sequence tokens.

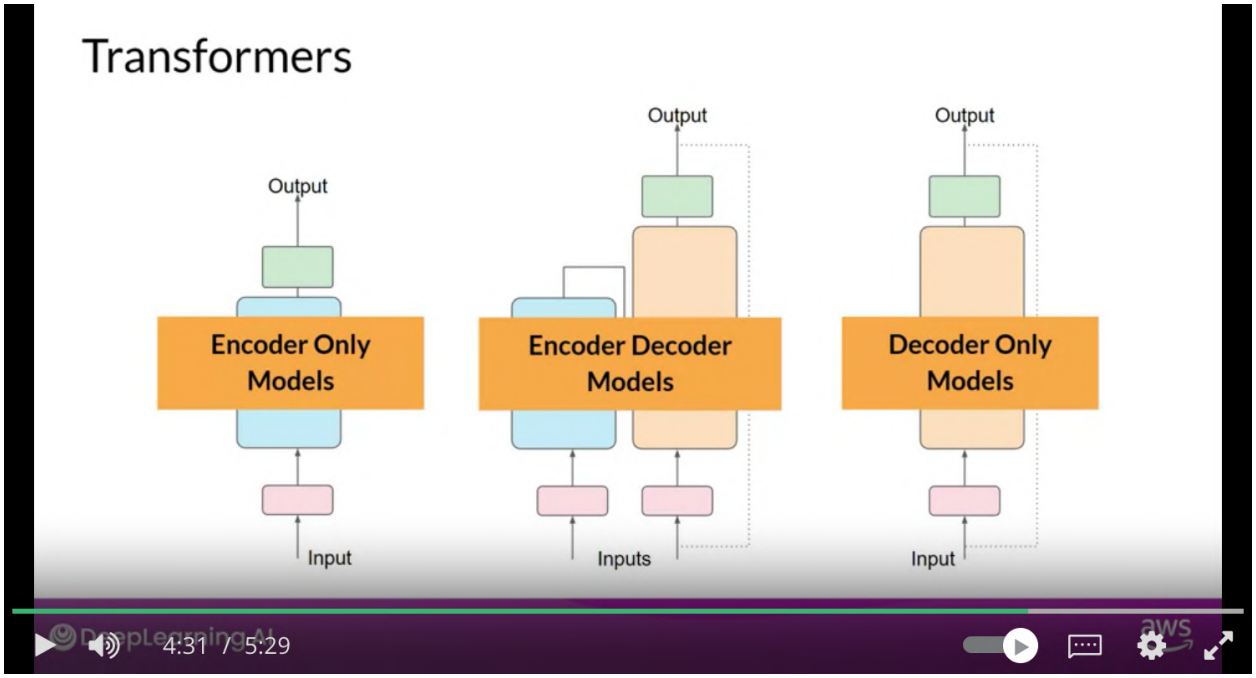
Using softmax layer output -There are multiple ways to use output to softmax layer which determines how creative generated text is.



Your continue this loop passing the output token back to the input to trigger the generation of the next token until the model predicts an end of sequence token at this point the final sequence of tokens can be detoised into words and you have your output in this case I love machine learning there are multiple ways in which you can use the

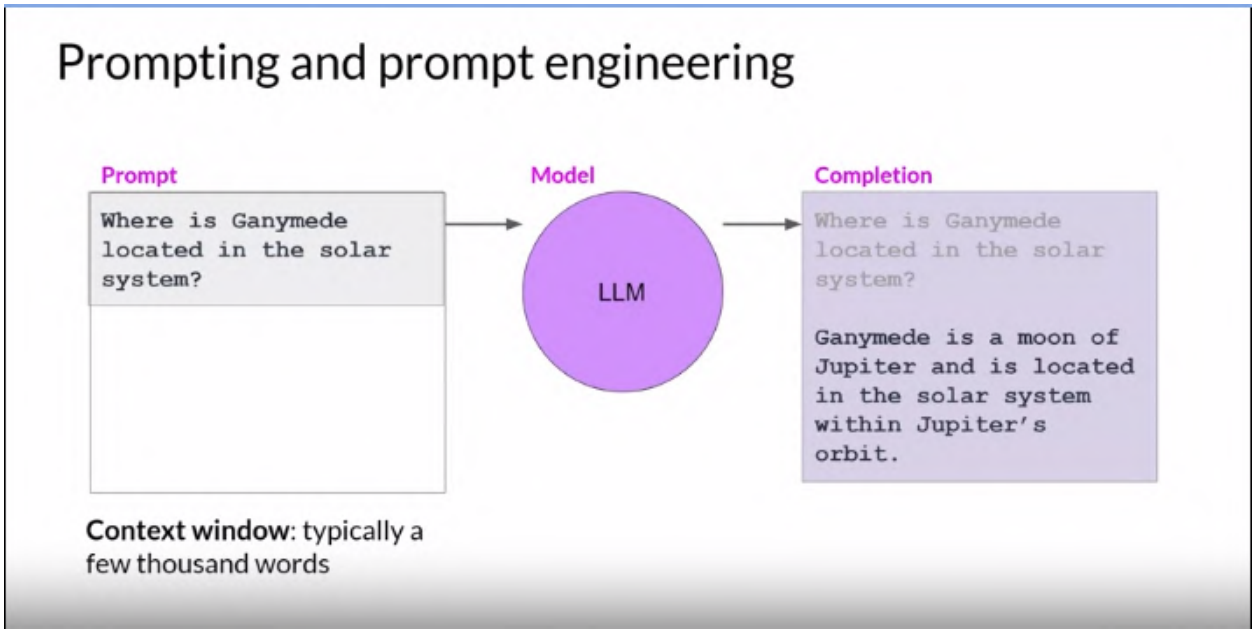
Study Notes: Transformer Architecture and Model Components

- The transformer architecture consists of an encoder and decoder components.
- The encoder encodes input sequences into a deep representation of the structure and meaning of the input.
- The decoder uses the encoder's contextual understanding to generate new tokens.
- The complete prediction process in a transformer model involves the following steps:
  1. Tokenization: Input words are tokenized using a tokenizer.
  2. Encoder Input: The tokens are added to the input on the encoder side, passed through the embedding layer, and then fed into the multi-headed attention layers.
  3. Encoder Output: The outputs of the multi-headed attention layers are passed through a feed-forward network to the output of the encoder, resulting in a deep representation of the input sequence's structure and meaning.
  4. Decoder Input: The deep representation from the encoder is inserted into the middle of the decoder to influence the decoder's self-attention mechanisms.
  5. Start of Sequence Token: A start of sequence token is added to the input of the decoder to trigger the generation of the next token.
  6. Decoding Loop: The decoder predicts the next token based on the contextual understanding provided by the encoder. The output of the decoder's self-attention layers goes through the decoder feed-forward network and a final softmax output layer.
  7. Token Generation Loop: The output token is passed back to the input to trigger the generation of the next token, continuing until the model predicts an end-of-sequence token.
  8. Detokenization: The final sequence of tokens can be detokenized into words, resulting in the output.



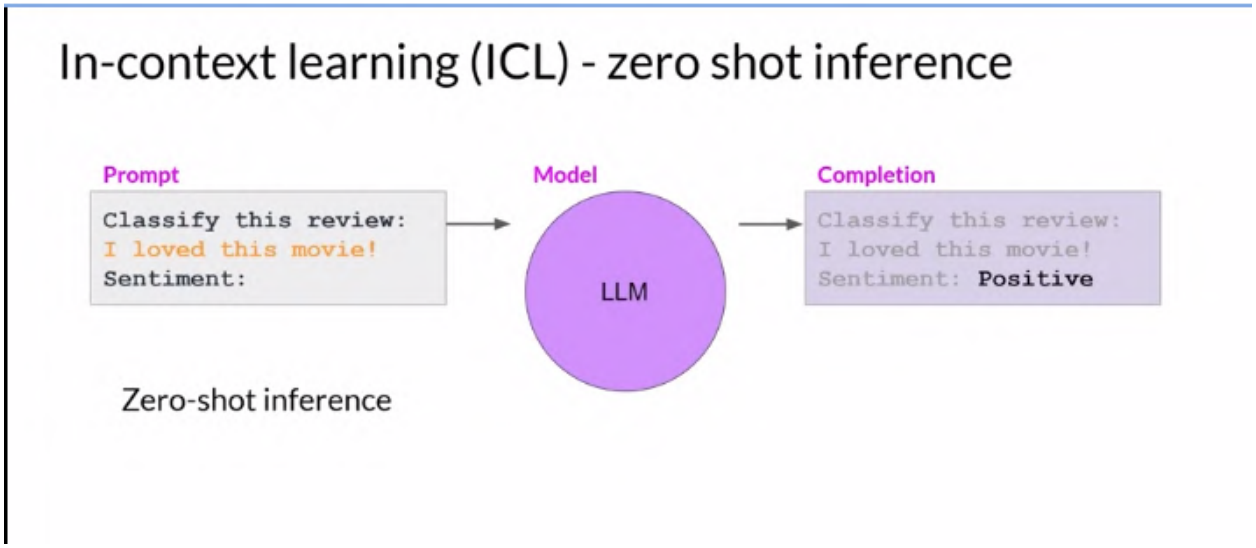
- There are variations of the transformer architecture:
  - Encoder-only models: They work as sequence-to-sequence models and encode input sequences, but the input and output sequences have the same length. E.g., BERT.
  - Encoder-decoder models: They perform well on sequence-to-sequence tasks, such as translation, where the input and output sequences can have different lengths. These models can also be used for general text generation tasks. E.g., BART, T-5
  - Decoder-only models: They are commonly used and can generalize to most tasks. Examples include the GPT family of models. Bloom, Jurassic, Llama. Generalize to most task
- Prompt engineering: When interacting with transformer models, prompts are created using written words, and there is no need to understand all the details of the underlying architecture.
- Prompt engineering will be explored in the next part of the course.

Prompting and prompt engineering

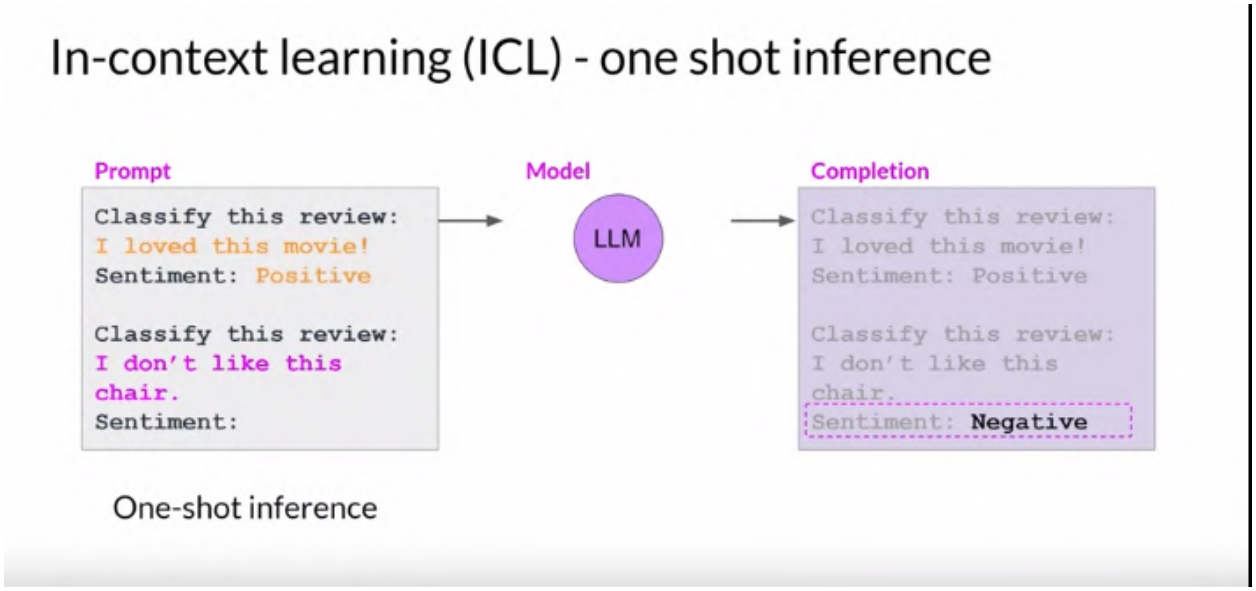


- Terminology:
  - Prompt: The text fed into the model.
  - Inference: The act of generating text.
  - Completion: The output text generated by the model.
  - Context Window: The full amount of text available for the prompt. - memory available

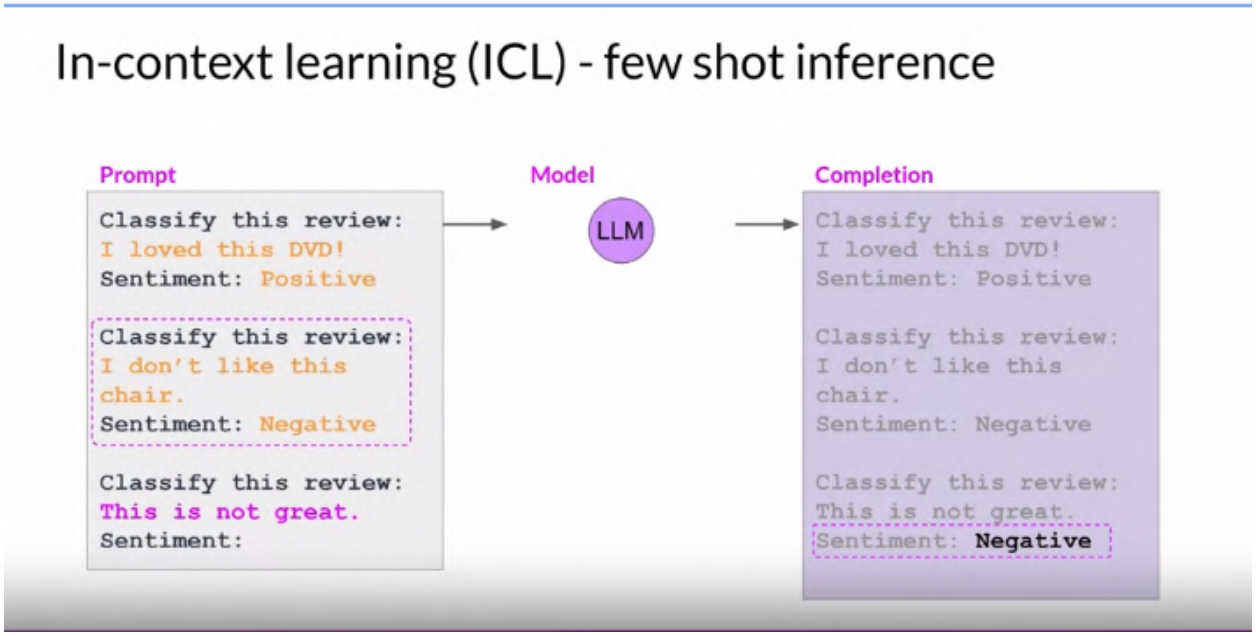
- Prompt Engineering:
  - It involves revising the prompt language or structure to get the desired model behavior.
  - Including examples of the desired task within the prompt can improve outcomes.
  - In-context learning: Providing examples inside the context window.



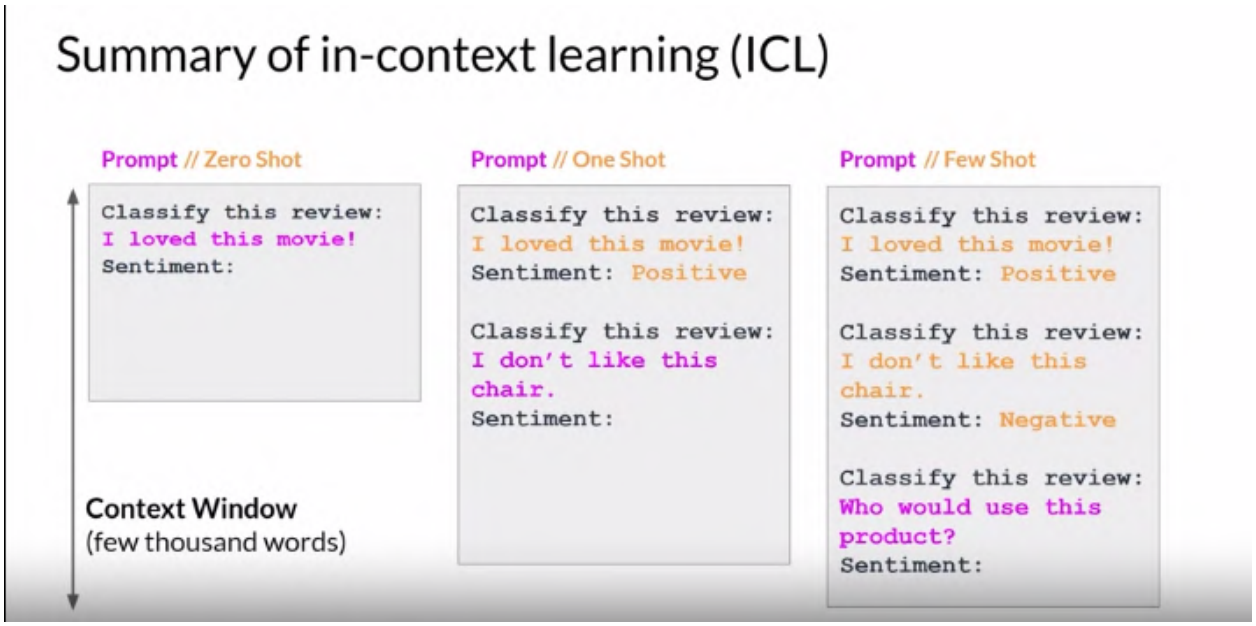
- Zero-shot Inference:
  - Involves including input data within the prompt.
  - Large language models (LLMs) are good at zero-shot inference.
  - LLMs can grasp the task and produce accurate answers.
  - Smaller models may struggle with zero-shot inference.



- One-shot Inference:
  - Involves providing a single example within the prompt.
  - Helps smaller models understand the task better.
  - Improves performance compared to zero-shot inference.



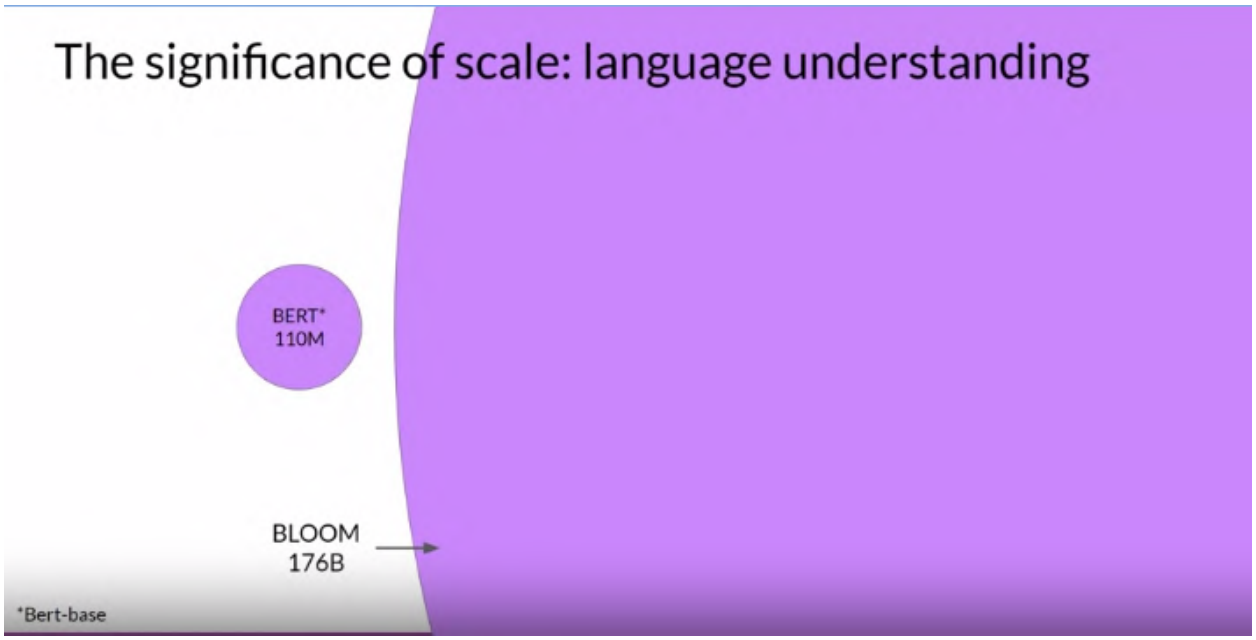
- Few-shot Inference:
  - Involves including multiple examples within the prompt.
  - Helps the model learn with different output classes.
  - Useful when a single example is not enough for the model to learn.



- Prompt Engineering for Learning by Examples:
  - Large models are good at zero-shot inference.
  - Smaller models benefit from one-shot or few-shot inference.
  - Prompt engineering guides the model using example-based learning.

- Context Window and Model Performance:

- The context window has a limit on in-context learning.
- If the model performs poorly with multiple examples, fine-tuning the model is recommended.
- Fine-tuning involves additional training with new data to improve model capability.

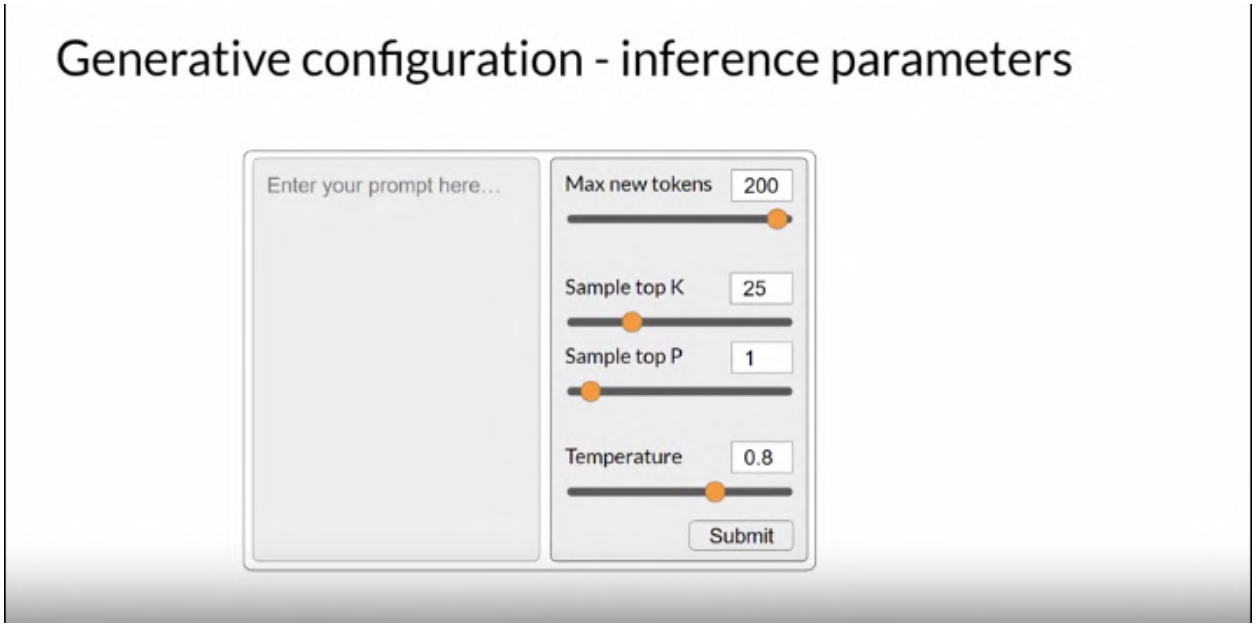


- Scale of Models and Task Performance:
  - Larger models perform better at multiple tasks.
  - More parameters enable greater understanding of language.
  - Smaller models are generally limited to tasks similar to their training.

- Choosing the Right Model:
  - Different models may be suitable for different use cases.
  - Experimenting with models helps find the most appropriate one. Try out different models for use case

- Configuration Settings:
  - Influence the structure and style of generated completions.
  - Can be experimented with to achieve desired outputs.

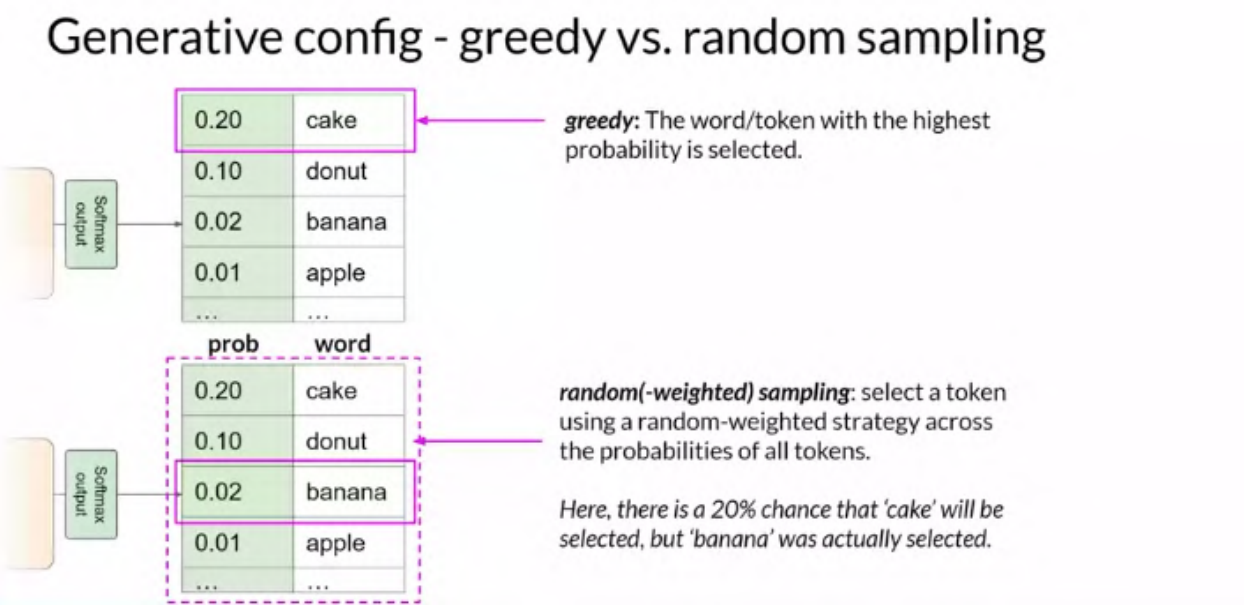
Generative configuration



- Configuration Parameters for Model Output:
  - Different methods and configuration parameters can influence next-word generation.
  - Configuration parameters are separate from training parameters.
  - Parameters control things like maximum tokens, creativity, and randomness.

- Max New Tokens:
  - Limits the number of tokens generated by the model.

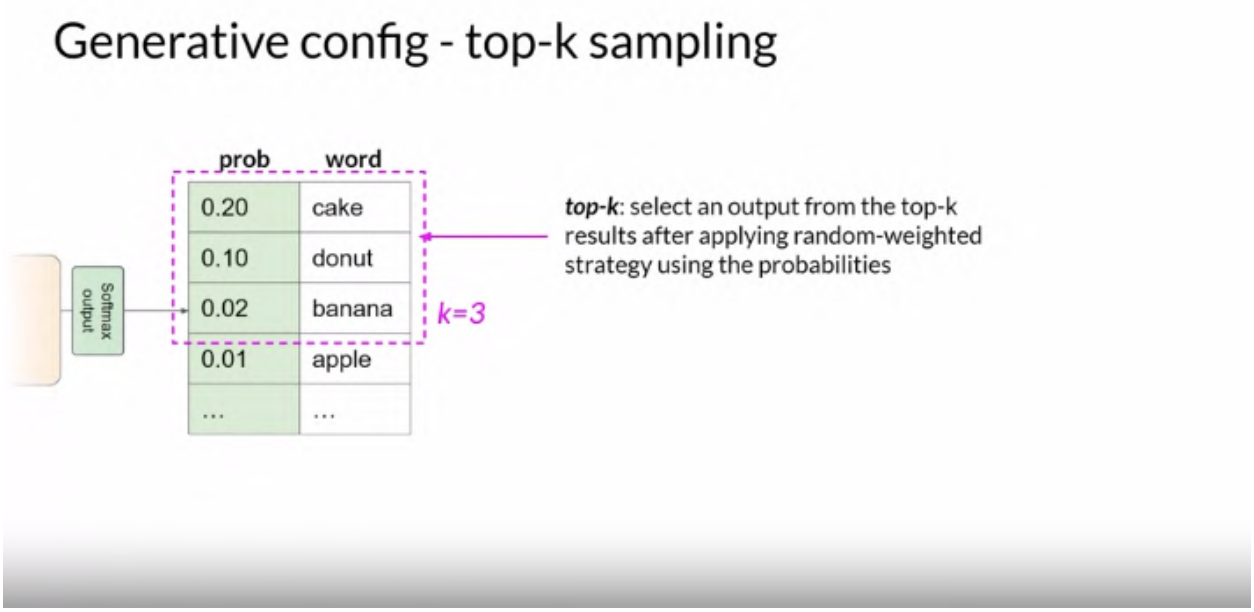
- Controls the selection process and can result in shorter completions.
- Not a fixed number but a maximum limit. Stop token can occur before reaching max token



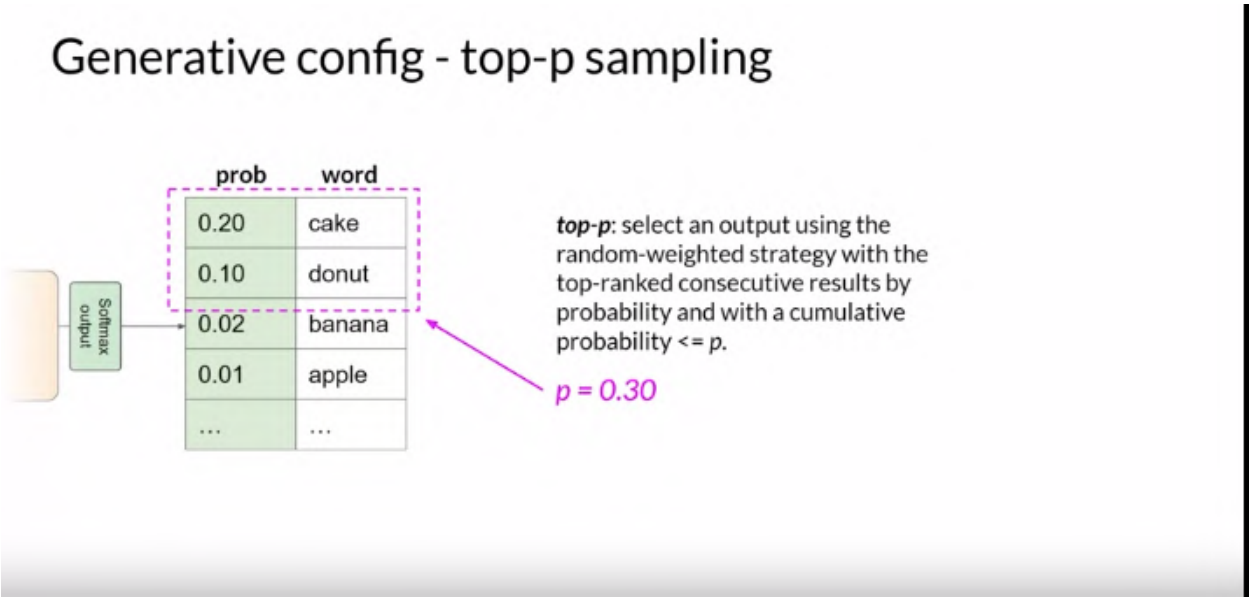
- Softmax Layer and Probability Distribution:
  - The output of the softmax layer is a probability distribution.
  - It represents the likelihood of each word in the model's dictionary.
  - Probability scores determine word selection.

- Greedy Decoding:
  - Simplest form of next-word prediction.
  - Always chooses the word with the highest probability.
  - Prone to repeated words or sequences.

- Random Sampling:
  - Introduces variability and avoids word repetition.
  - Words are selected randomly based on probability distribution.
  - Reduces the predictability of the generated text.

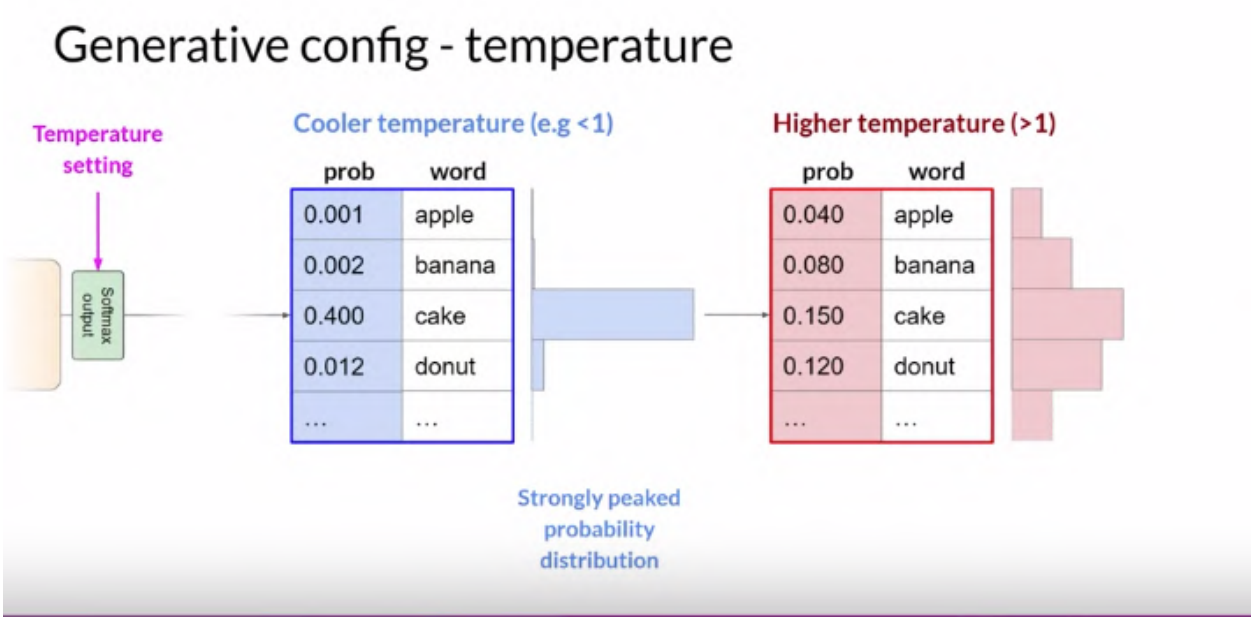


- Top k Sampling:
  - Limits random sampling options to the top k tokens with the highest probability.
  - Increases randomness while avoiding highly improbable words.
  - Helps generate more sensible and reasonable text.
  - Here specify the top K for the model to choose word from



- Top p Sampling:
  - Limits random sampling to predictions whose combined probabilities do not exceed p.
  - Selects from tokens with probabilities within the defined limit.
  - Provides control over randomness while maintaining sense.
  - Here specify the top P (probability) to choose from

- Temperature:
  - Influences the shape of the probability distribution.
  - Higher temperature increases randomness, lower temperature decreases randomness.
  - Scaling factor applied in the softmax layer.



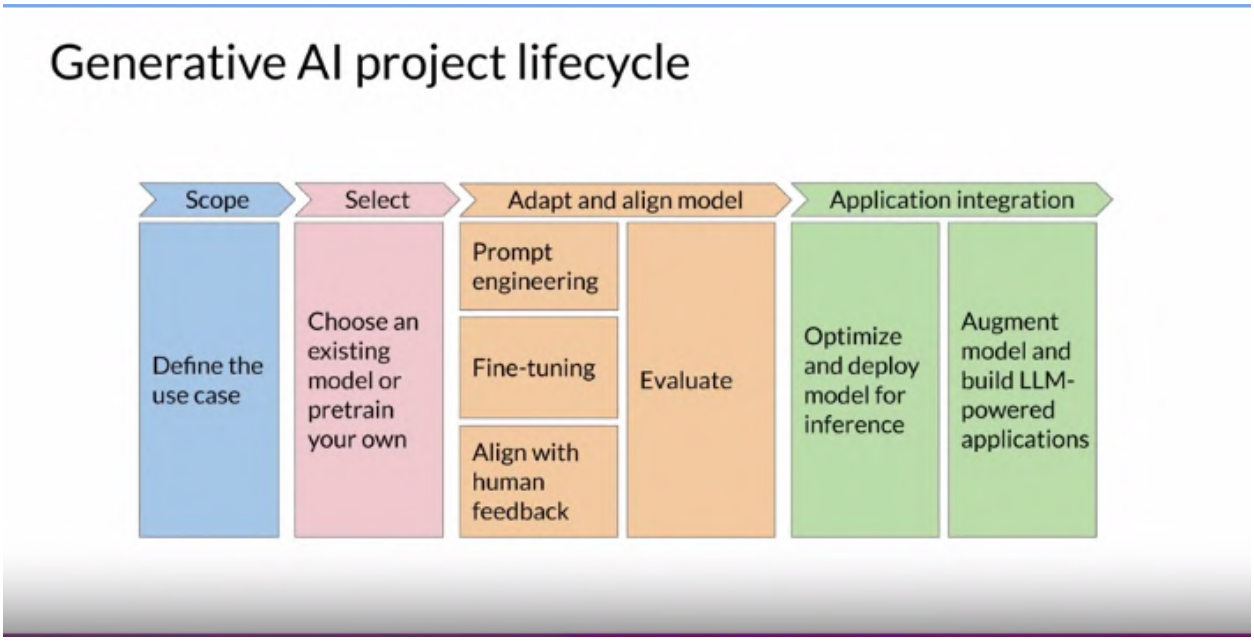
- Low Temperature:
  - Concentrates probability on a smaller number of words.
  - Results in less random text following likely word sequences.

- High Temperature:
  - Spreads probability more evenly across tokens.
  - Generates text with higher randomness and variability.

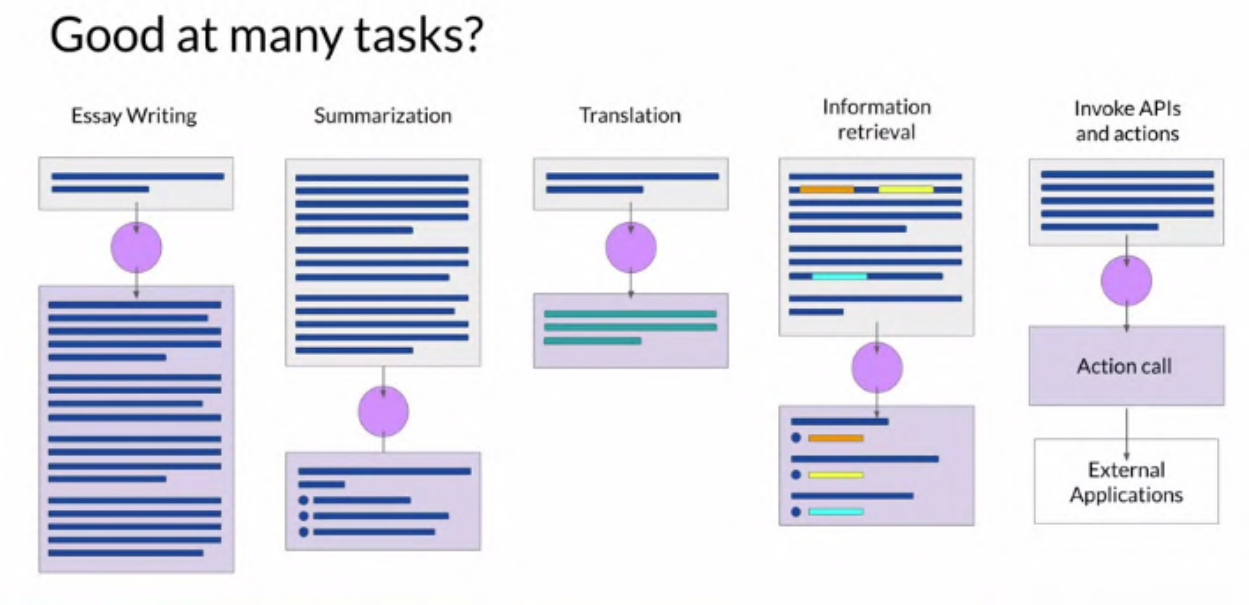
- Default Temperature:
  - Leaving the temperature as 1 uses the unaltered probability distribution.
  - Default behavior of the softmax function.

- Prompt Engineering and Inference Configuration:
  - Prompt engineering optimizes model behavior.
  - Experimenting with inference configuration parameters can improve performance.

Generative AI project lifecycle



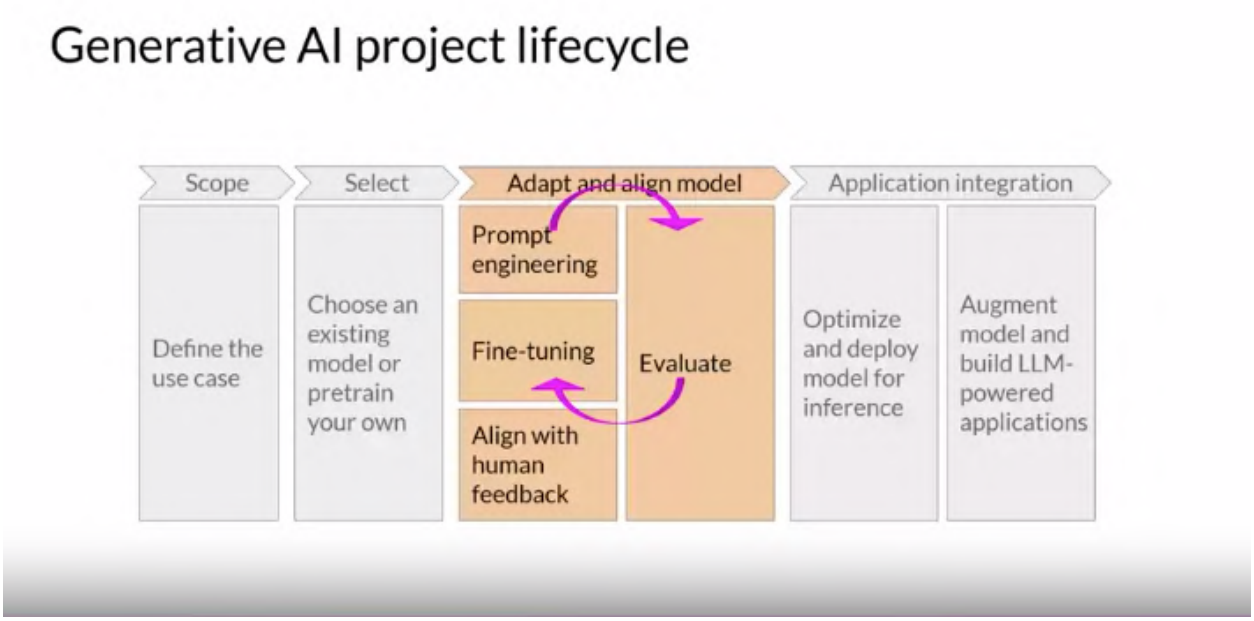
- Generative AI Project Life Cycle:
  - The life cycle guides the development and deployment of an LLM-powered application.
  - It covers the tasks from conception to launch.



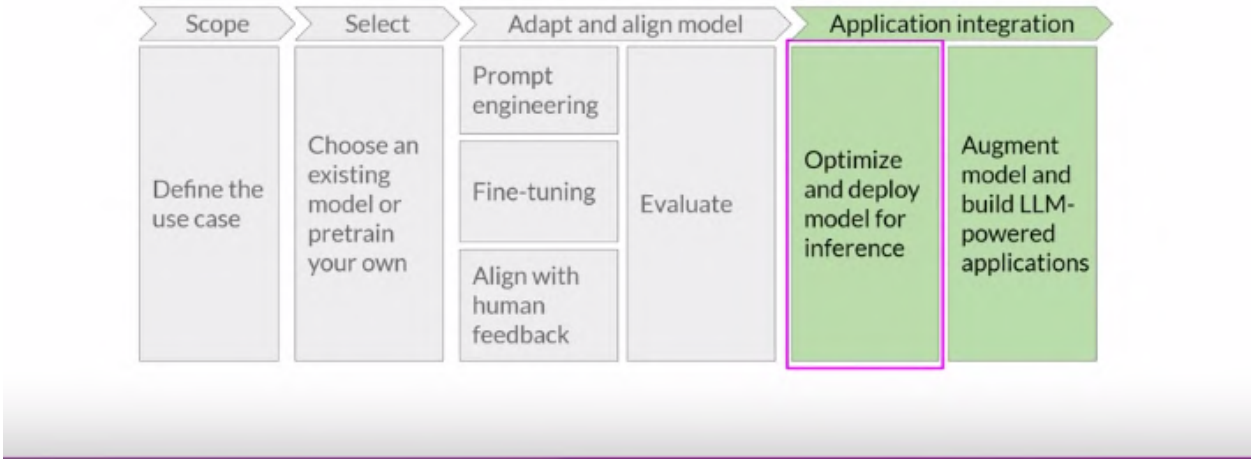
- Defining Scope:
  - Define the scope of the project accurately and narrowly.
  - Consider the LLM's function and capabilities required for the application.
  - Specificity in requirements can save time and compute costs.

- Training Model:
  - Decide whether to train your own model or use an existing base model.
  - Starting with an existing model is common.
  - Training considerations and feasibility are discussed later in the course.

- Assessing and Improving Model Performance:
  - Evaluate the model's performance for the application.
  - In-context learning with prompt engineering can enhance performance.
  - Fine-tuning can be applied if the model requires further improvement.
  - Reinforcement learning with human feedback ensures desired model behavior.



- Evaluation:
  - Metrics and benchmarks are used to assess model performance and alignment.
  - Iterative process of evaluating and refining prompt engineering and fine-tuning.



- Deployment and Integration:
  - Deploy the well-performing and aligned model into the application infrastructure.
  - Optimize the model for deployment to utilize compute resources efficiently.
  - Consider additional infrastructure requirements for the application's success.

- Overcoming Model Limitations:
  - LLMs have limitations in inventing information and complex reasoning.

Introduction to AWS labs

- Hands-on Experience:
  - Practical application helps solidify the concepts learned in the course.
  - Lab exercises allow trying out key concepts from the videos.
  - The lab environment, Vocareum, provides access to Amazon SageMaker.

- Vocareum Lab Environment:
  - Launches an AWS account with access to Amazon SageMaker.
  - AWS status changes from red to yellow to green.
  - 2 hours provided to complete each lab.
  - No need to click "end lab"; simply close the browser and move on.

- Accessing SageMaker Studio:
  - Click on the AWS button to launch the AWS console.
  - If already logged in, click "logout" before clicking the AWS button.
  - Type "SageMaker" in the search field and click on it.
  - Click on "Open Studio" to access the Jupyter-based IDE.

- Jupyter Notebook Setup:
  - Pre-provisioned setup, no additional actions required.
  - Dark mode enabled by default, can switch to light mode.
  - Click "Open Launcher" and then "System terminal."

- Copying Lab Materials:
  - Instructions provide code to copy from S3 bucket.
  - Paste the code in the system terminal.
  - Click on the folder to see the downloaded notebook.

- Running the Labs:
  - Shift+Enter moves from cell to cell in Jupyter notebooks.
  - Alternatively, use "Restart Kernel and Run All Cells" for quick execution.
  - Encouraged to run step by step for the labs.

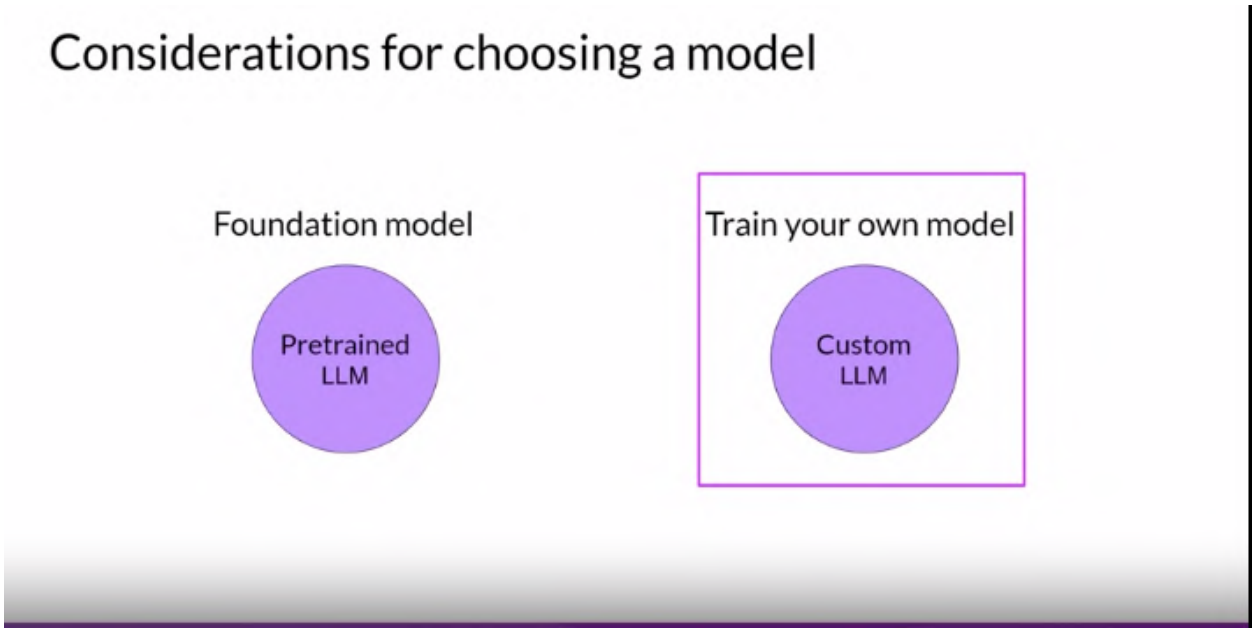
Lab 1:

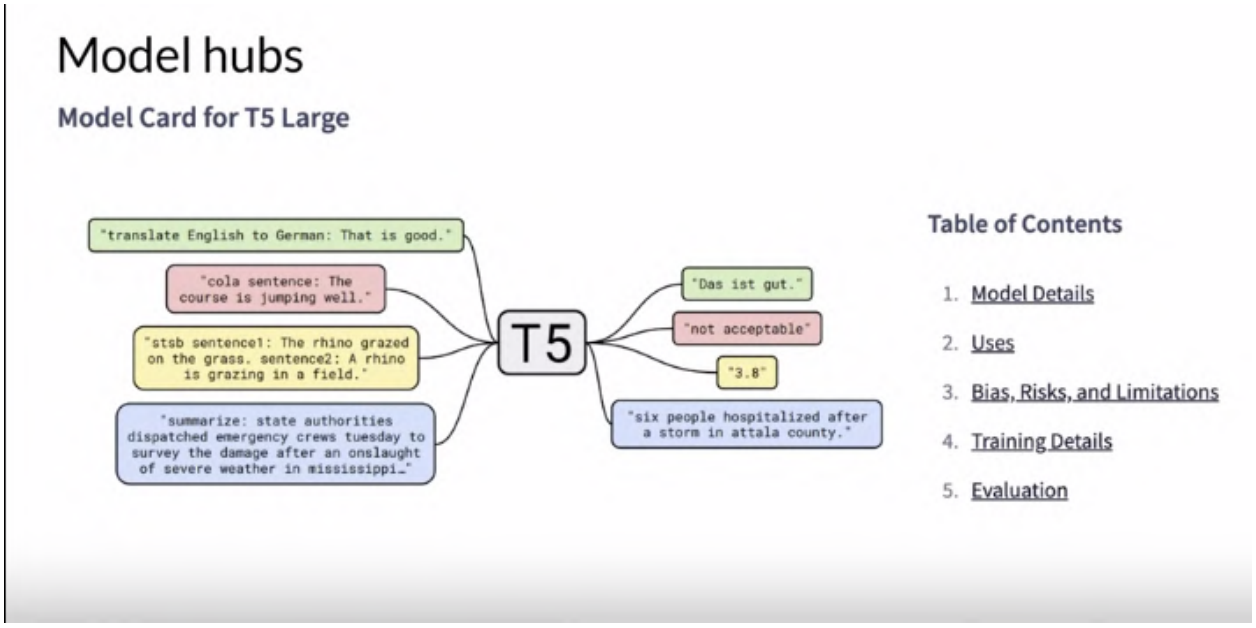
- The goal is to grab a data-set of conversations and summarize them.
- The data-set will be used to summarize customer support dialogues at the end of the month.
- The system has eight CPUs and 32 gigs of RAM, using Python 3 and various libraries.
- The libraries being installed include PyTorch, Torch data (for data loading with PyTorch), and Transformers (a library by Hugging Face for working with language models).
- It may take a few minutes to load the libraries, and some warnings may appear, but they can be ignored.
- The next step is to import the necessary functions, models, and tokenizers for the lab.
- The lab uses a data-set called "Dialogue Sum" provided by Transformers.
- The data-set contains conversations and human-labeled summaries.

- The baseline human summaries will be compared to the summaries generated by the model.
- The FLAN-T5 model is used for conversation summarization.
- The tokenizer converts raw text into numerical representations (embeddings) that can be processed by the model.
- The conversations are passed through the FLAN-T5 model to generate summaries.
- The initial results show that the model's summaries are not accurate.
- Different prompt engineering techniques are explored to improve the model's performance.
- Zero-shot inference with an instruction ("summarize the following conversation") is tried but doesn't yield significant improvement.
- Another prompt "dialogue corn" is used to experiment with different phrasing, but the results are still not optimal.
- One-shot inference is attempted by giving the model complete examples with and without the summary, but the improvement is minimal.
- Few-shot inference is tested by providing multiple examples with and without the summary, but the results vary.
- Prompt engineering is an initial step in understanding and exploring the capabilities of a language model, and it helps in choosing the right approach for fine-tuning.
- In future lessons, fine-tuning the model will be explored.
- Configurable parameters such as sampling and temperature can be adjusted to influence the model's responses.
- Higher temperatures (close to 2) lead to more creative responses, while lower temperatures (0.1) make the responses more conservative and repetitive.

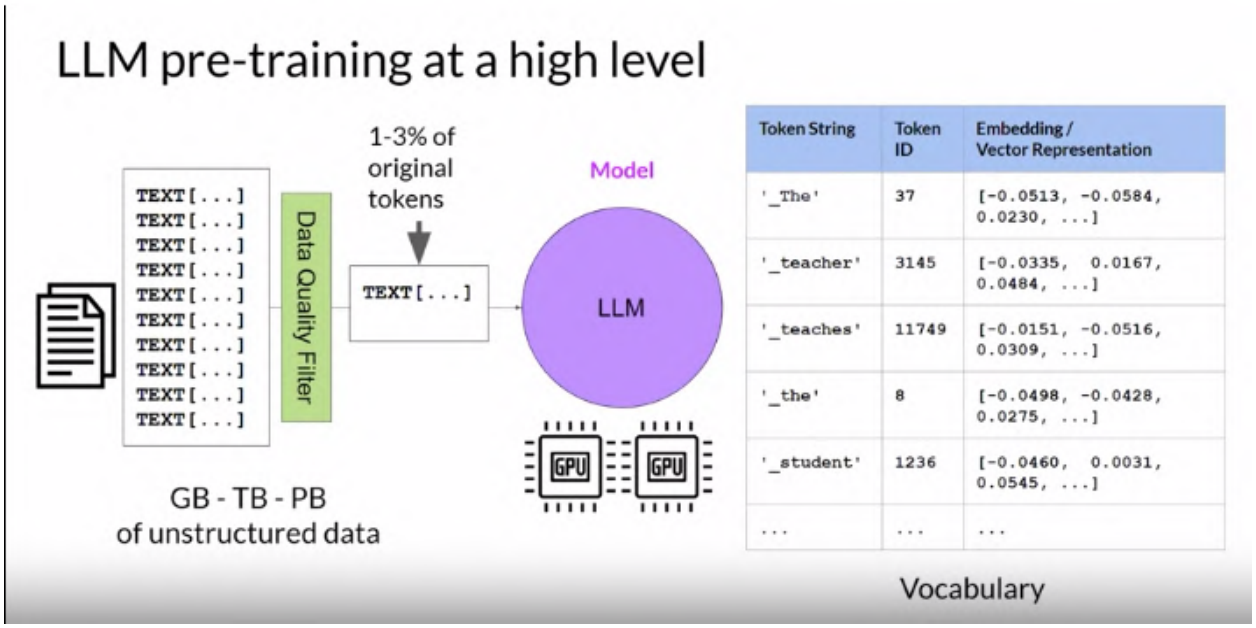
W1 - LLM PreTraining and scaling laws

Pre-training large language models

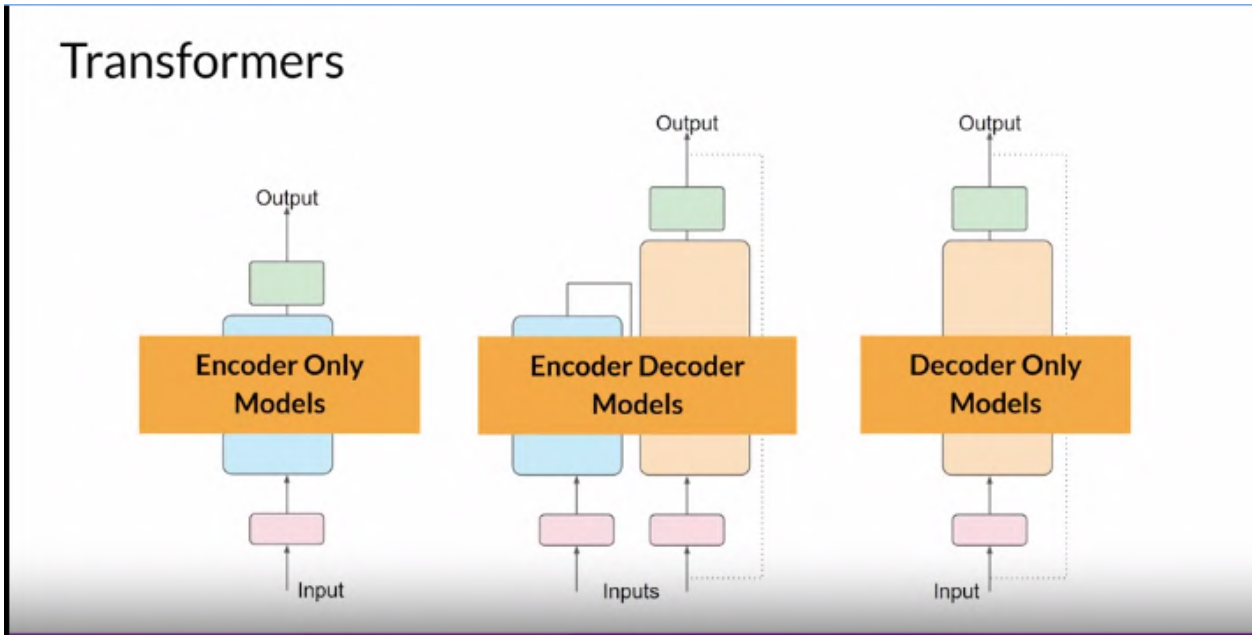




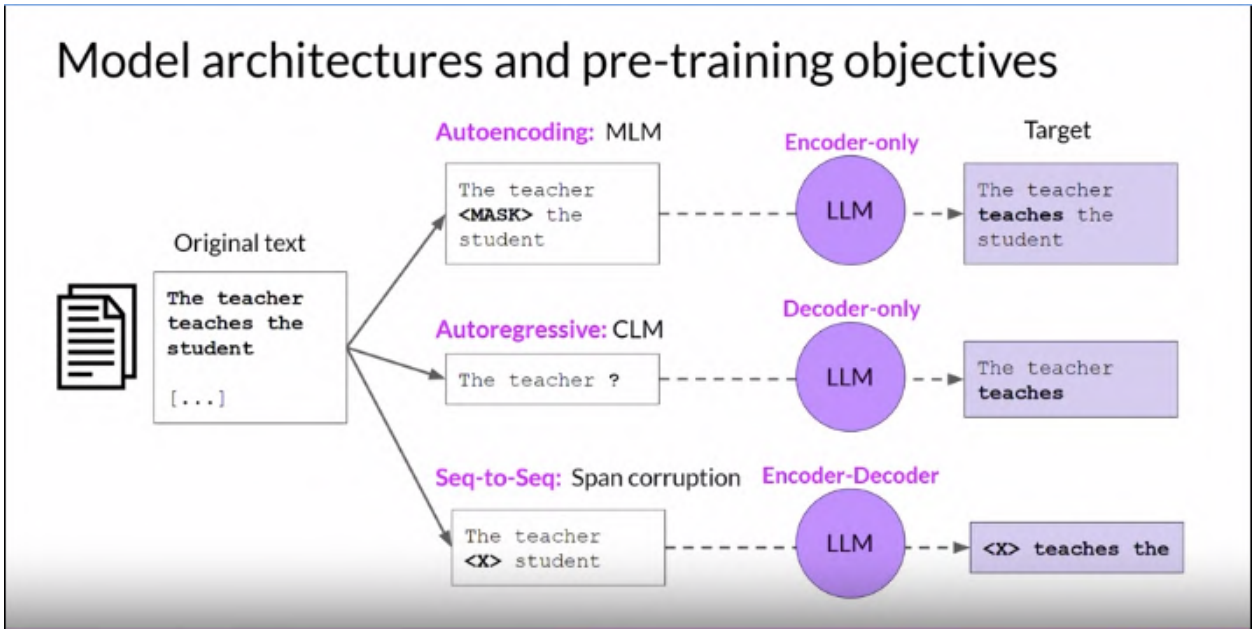
- Selecting a Model:
  - Choose between working with an existing model or training your own.
  - Existing models are commonly used, and open-source options are available.
  - Model hubs like Hugging Face and PyTorch provide curated models and model cards.

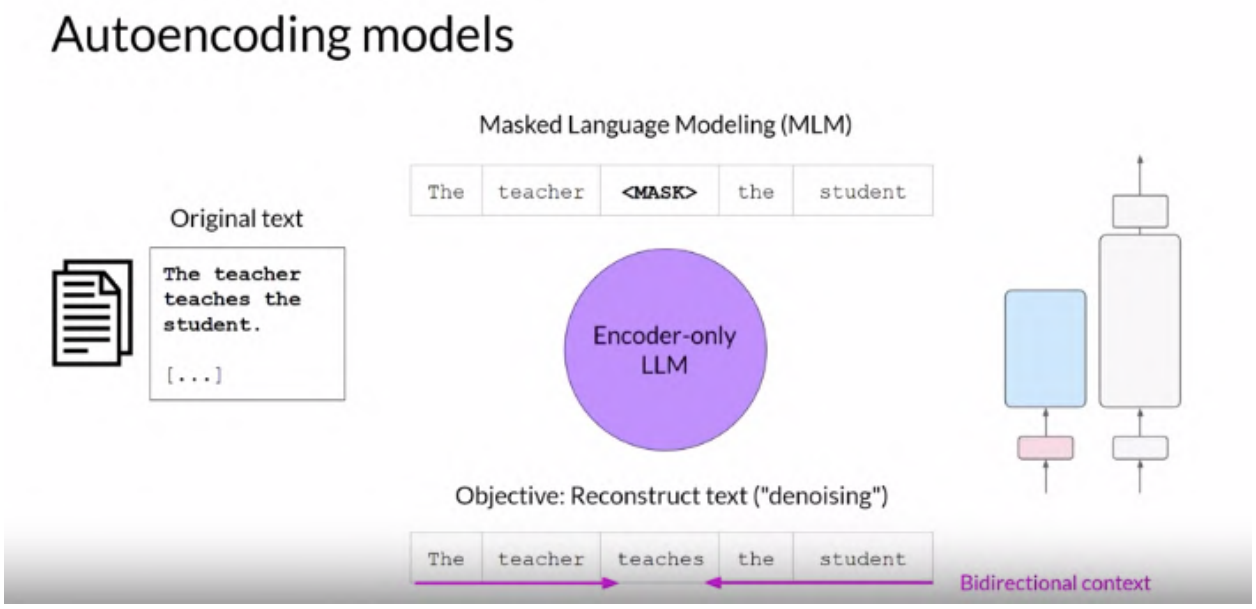


- Training Large Language Models (LLMs):
  - Pre-training phase develops the model's understanding of language.
  - LLMs learn from vast amounts of unstructured textual data.
  - Data is sourced from the internet, corpora, and training-specific text collections.
  - Self supervised step model internalized the pattern and structures present in the language
  - Pre-training updates model weights to minimize the loss of the training objective (training objective depends on the architecture of the model).
  - Large compute resources and GPUs are used during pre-training.
  - Only 1-3% of data collected actually used in pre-training

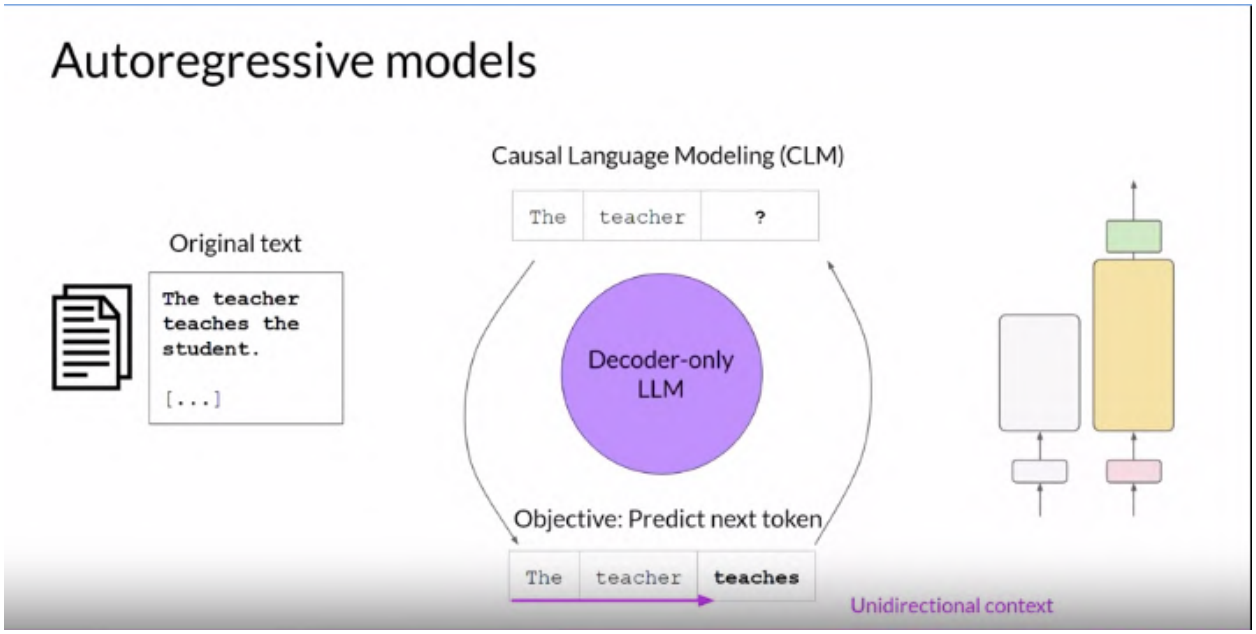


- Variance of Transformer Model Architecture:

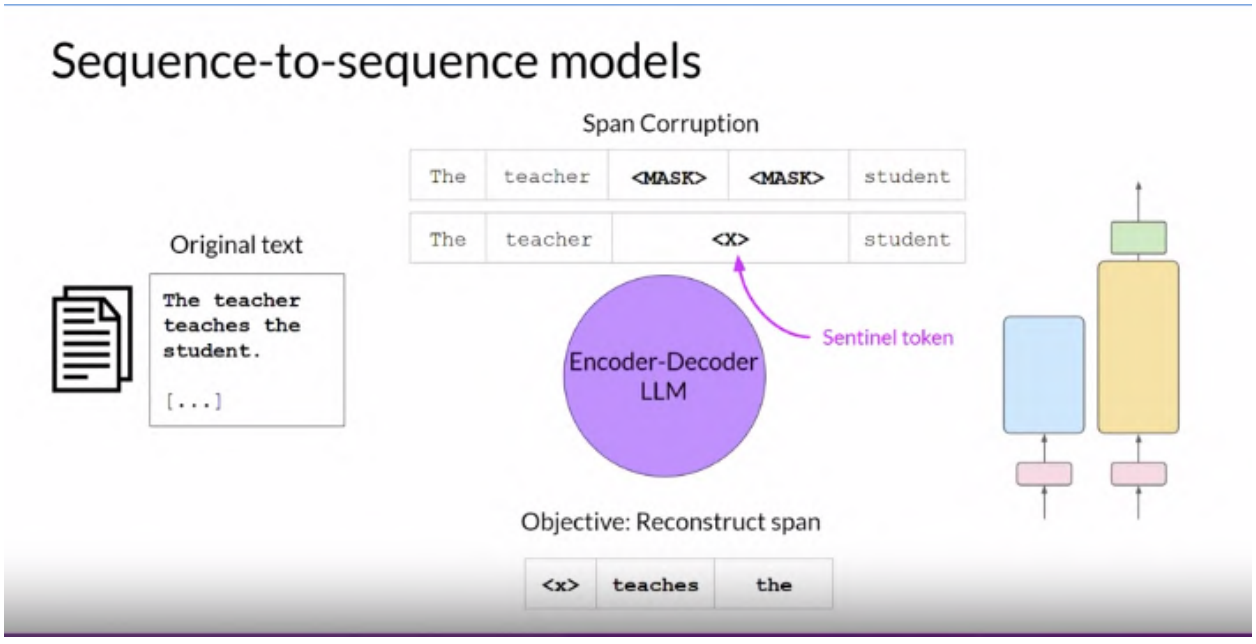




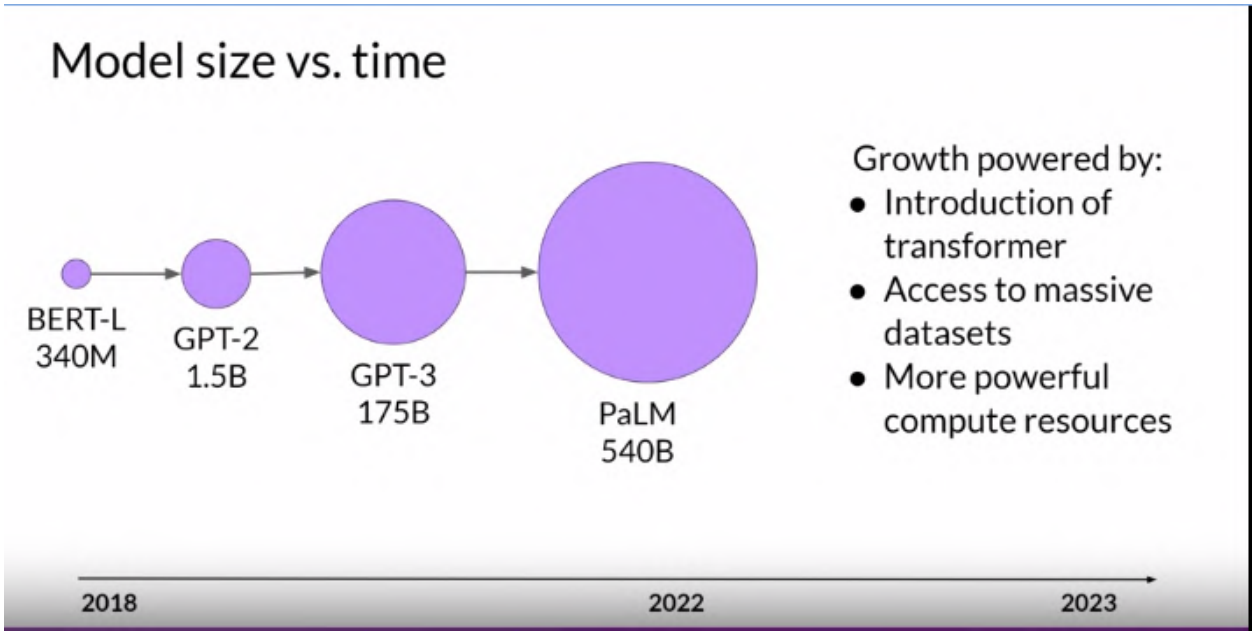
- **Encoder-only models** (Autoencoders) are trained with masked language modeling.
  - Encode full token context, suitable for sentence and token-level tasks that use bidirectional context.
  - Can be used for sentence classification task like sentiment analysis or token level task like NER or word classification
    - Examples: BERT, RoBERTa.



- **Decoder-only models** (Autoregressive) use causal language modeling.
  - Predict next token based on previous tokens, unidirectional context.Predicting next token sometimes call full language modeling
  - Decoder based auto regressive models mask the input sequence and can only see the input tokens leading upto token in question. The model has no knowledge of the end of the sentence . The model then iterate over the input sequence one by one to predict the following token
  -By learning to predict the next token from vast number of training examples, the model builds up the statistical representation of language.
  - Primarily used for text generation; larger models show zero-shot inference abilities.
  - Examples: GPT, BLOOM.



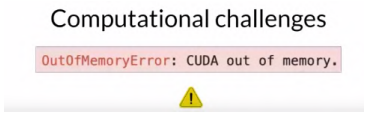
- **Sequence-to-sequence** models utilize both encoder and decoder components.
  - Pre-training objective varies; example: T5 uses span corruption.Masks random sequences of input tokens. Those masked sequences are replaced by sentinel tokens. Sentinel tokens are special tokens added to the vocabulary but do not correspond to any actual word from the input text. The decoder is then tasked with reconstructing the masked token sequence auto regressively. The output is the sentinel token followed by the predicted token.
  - Useful for translation, summarization, and question-answering. Generally useful in cases where you have a body of text as both input and output.
  - Examples: T5, BART.



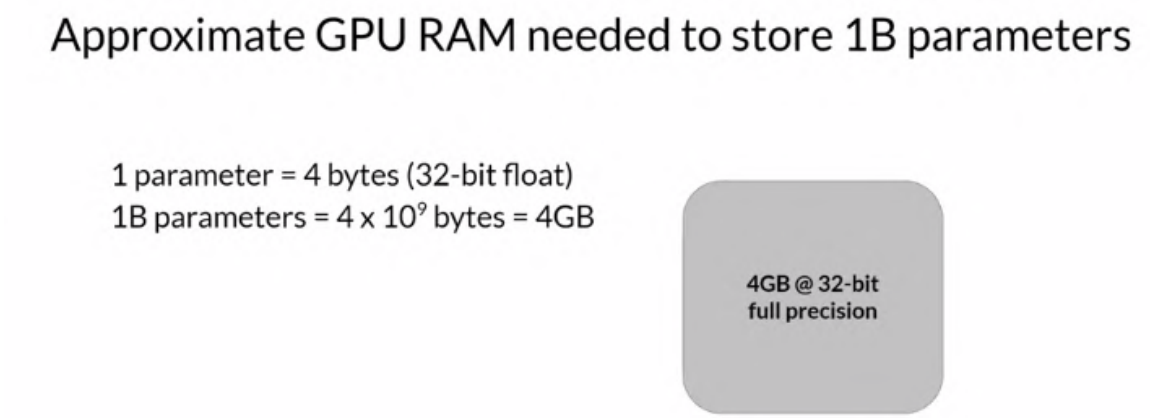
- Model Selection and Capability:
  - Choose the model architecture based on the specific task requirements.
  - Larger models tend to perform better without additional in-context learning or further training.
  - Model capability increases with size, driven by advancements in architecture, data, and compute resources.
  - Continuous training of larger models is challenging and expensive.

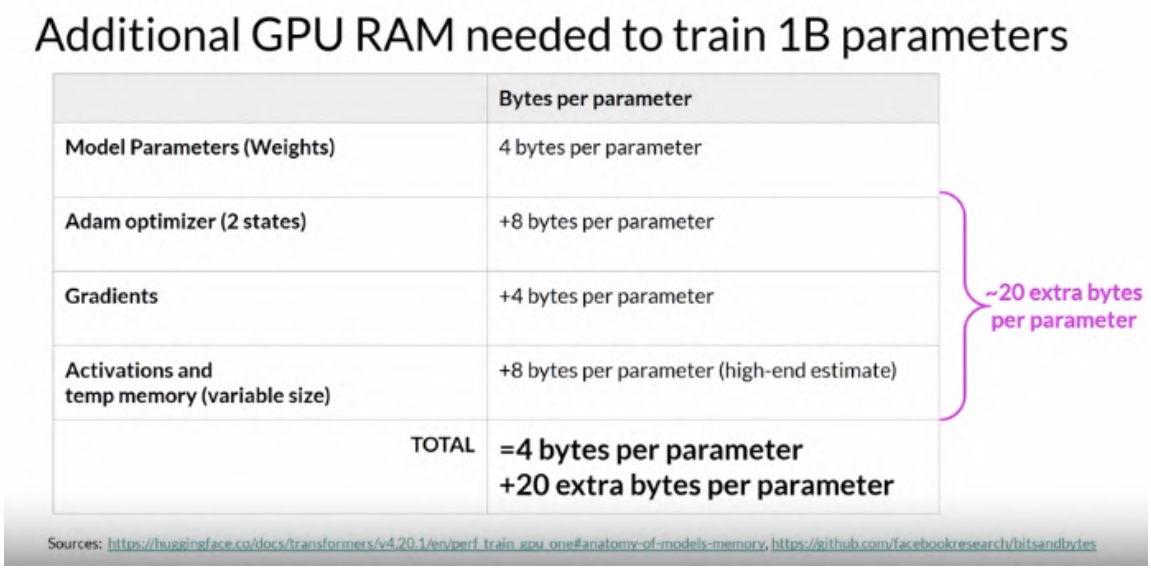
- Growth of Large Language Models:
  - Researchers have observed increased model capability with larger size.
  - Development driven by scalable transformer architecture, ample training data, and powerful compute resources.
  - Hypothesis of a new "Moore's law" for LLMs.
  - Challenges and feasibility of training larger models are explored in the next video.

Computational challenges of training LLMs

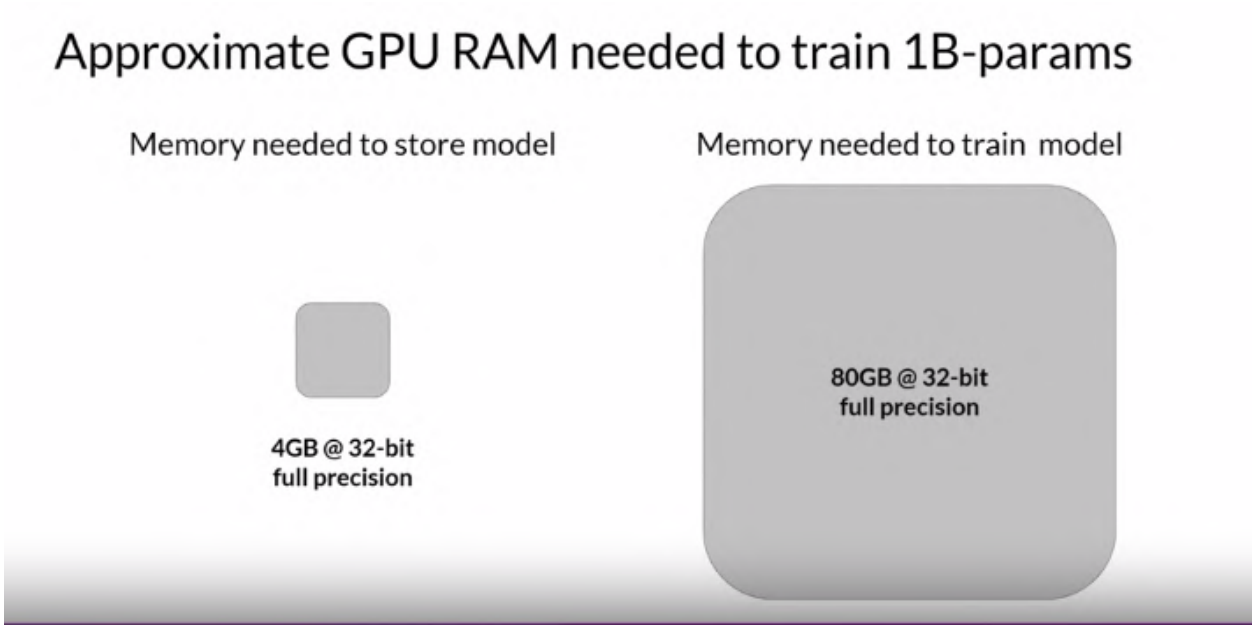


- Memory Challenges in Training Large Language Models (LLMs):
  - Out-of-memory issues are common when training large language models.
  - LLMs require significant memory to store and train their parameters.
  - CUDA (Compute Unified Device Architecture) libraries and tools are used for Nvidia GPUs. Pytorch and Tensorflow use CUDA to boost performance on matrix multiplication and other operation
  - Most LLMs are huge and require a substantial amount of memory.
  - Memory estimation requires accounting for model weights and additional components used during training.

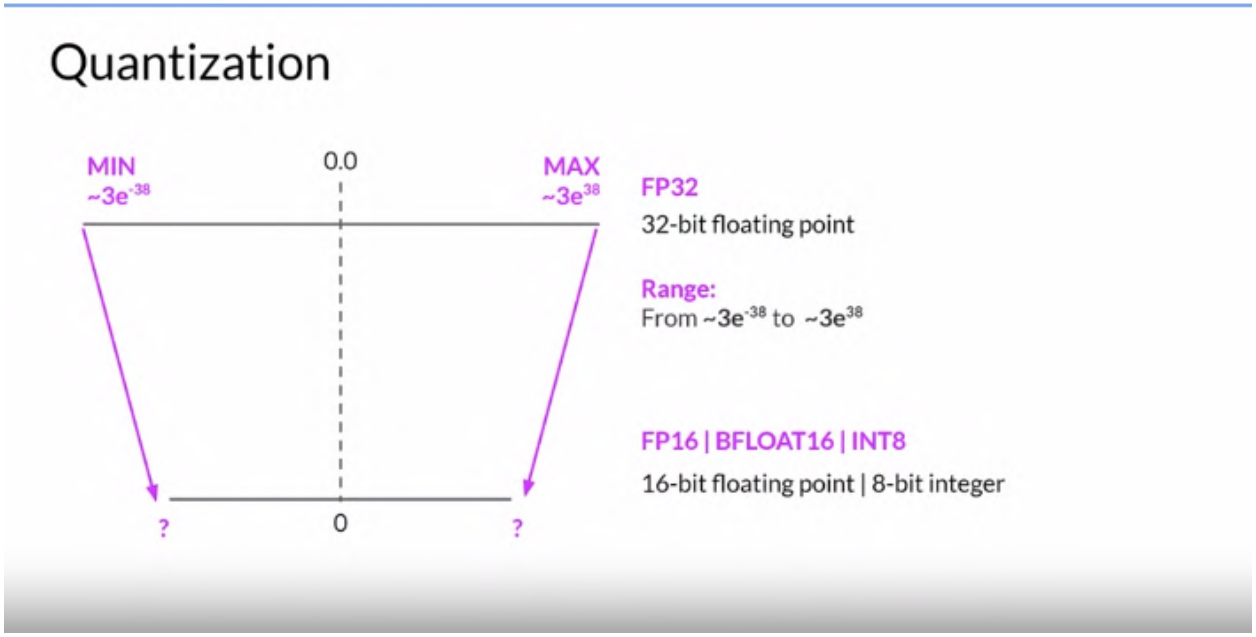




- 20 extra bytes of memory per model parameter

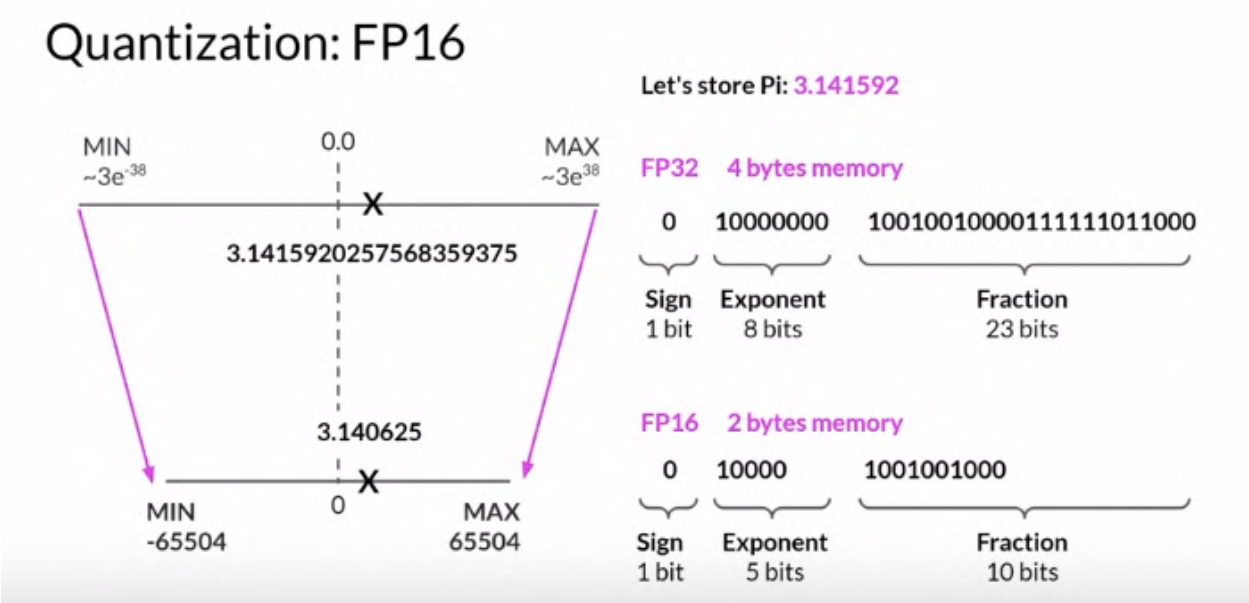


- To account for this during training, require 20 times the model GPU ram that the model weights alone take. NVIDIA A100 - 80 GB memory - is commonly processor used for ML task



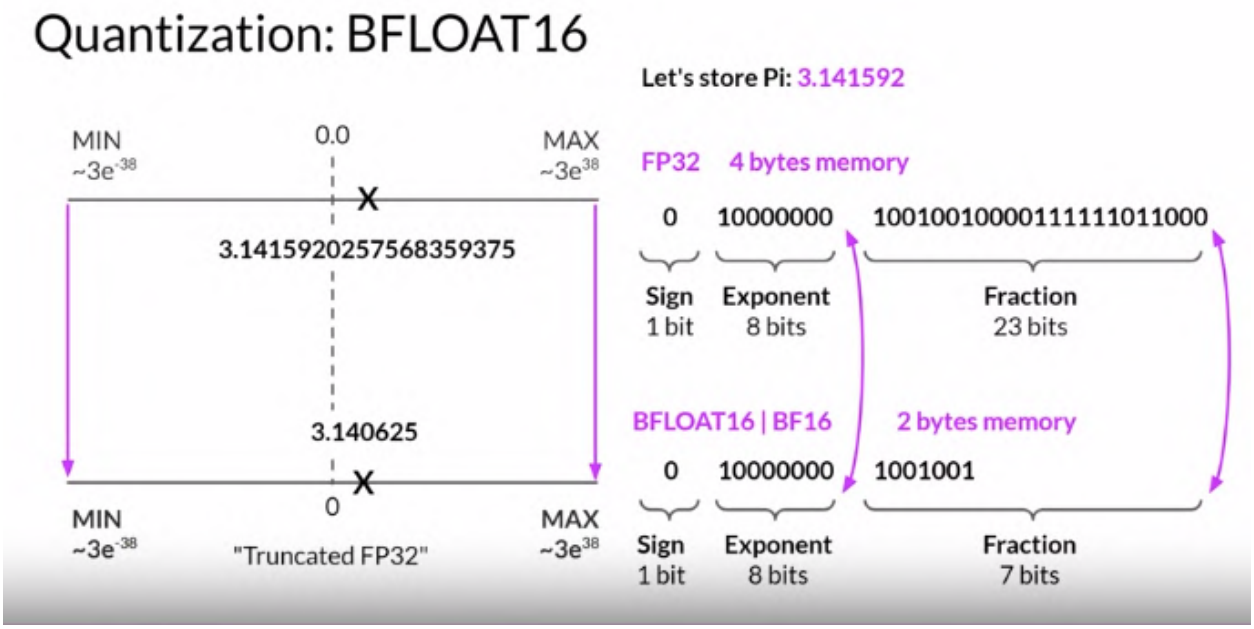
- Quantization to Reduce Memory:
  - By default model weights, activations and other model parameters are stored in FP32 .The range of numbers you can represent with FP32 goes from approximately 3*10^-38 to 3*10^38
  - Quantization reduces the memory required to store model weights.
  - Precision can be reduced from 32-bit floating point numbers to 16-bit or 8-bit representations.
  - Common data types: FP32 (32-bit), FP16/Bfloat16 (16-bit), INT8 (8-bit).

- Quantization statistically projects original 32-bit numbers into lower precision spaces using scaling factor calculated based on the range of the original 32bit numbers



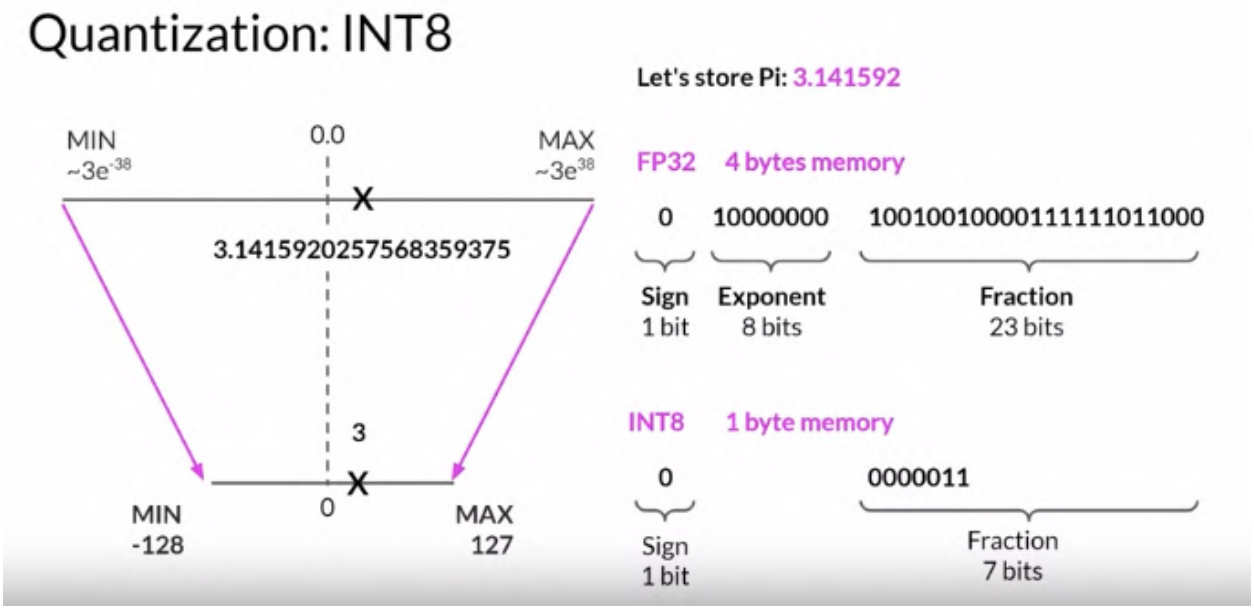
- For FP32 - 1 bit for sign (0 positive 1 negative), 8 bit for the exponent of the number and 23 bit representing the fraction of the number. Fractions also represented by Mantissa or Significand. It represents the precision bits of the number. If converting back a 32 bit float back to decimal value, there is slight loss in the precision
- FP16 - 1bit for sign - 5 bits for exponent and 10 bits for representing the fraction. Therefore the range of numbers that you can represent with FP16 is vastly smaller - from -65504 to 65504. Loose some precision . Only 6 places after the decimal above. This loss in precision is acceptable in most cases because try to optimize for memory footprint.

- Memory reduction is achieved by storing values in fewer bytes.FP32 4 bytes memory, FP16 2 bytes memory



- BFLOAT16 (BF16) has become popular due to its dynamic range and reduced memory footprint. Brain floating point format - popular choice in deep learning. Many LLMs including FlanT5 have been trained with BF16. It is a hybrid between half precision FP16 and Full precision FP32.BF16 Significantly helps with training stability.Supported by newer GPUs like Nvidia A100.Often described as truncated 32 bit float. Captures full dynamic range of 32 bit float yet uses only 16 bits. BF16 uses 8 bits exponents and truncates the fraction to 7 bits. This saves memory but also improves model performance by speeding up calculations.Downside - BF16 is not suited for integer calculations.

- INT8



Memory required only 1 byte but dramatic loss of precision

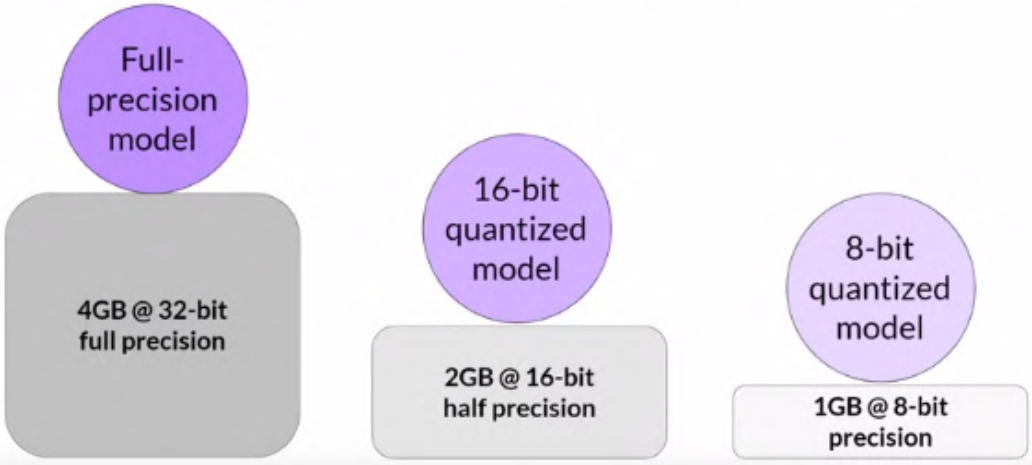
Quantization: Summary

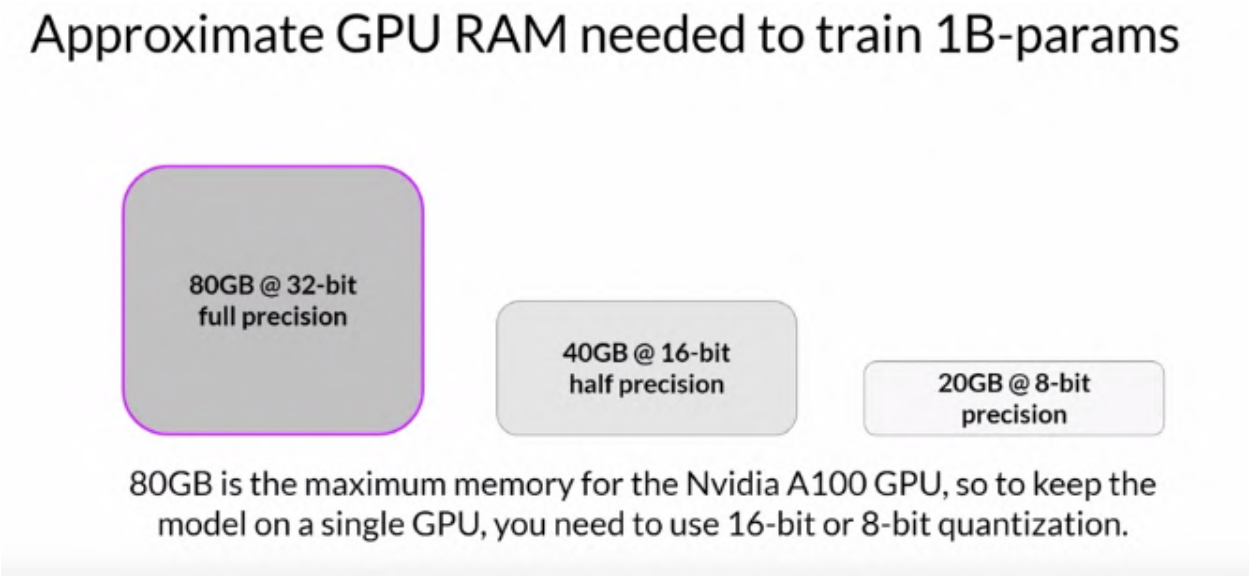
	Bits	Exponent	Fraction	Memory needed to store one value
FP32	32	8	23	4 bytes
FP16	16	5	10	2 bytes
BFLOAT16	16	8	7	2 bytes
INT8	8	-/-	7	1 byte

FLAN T5

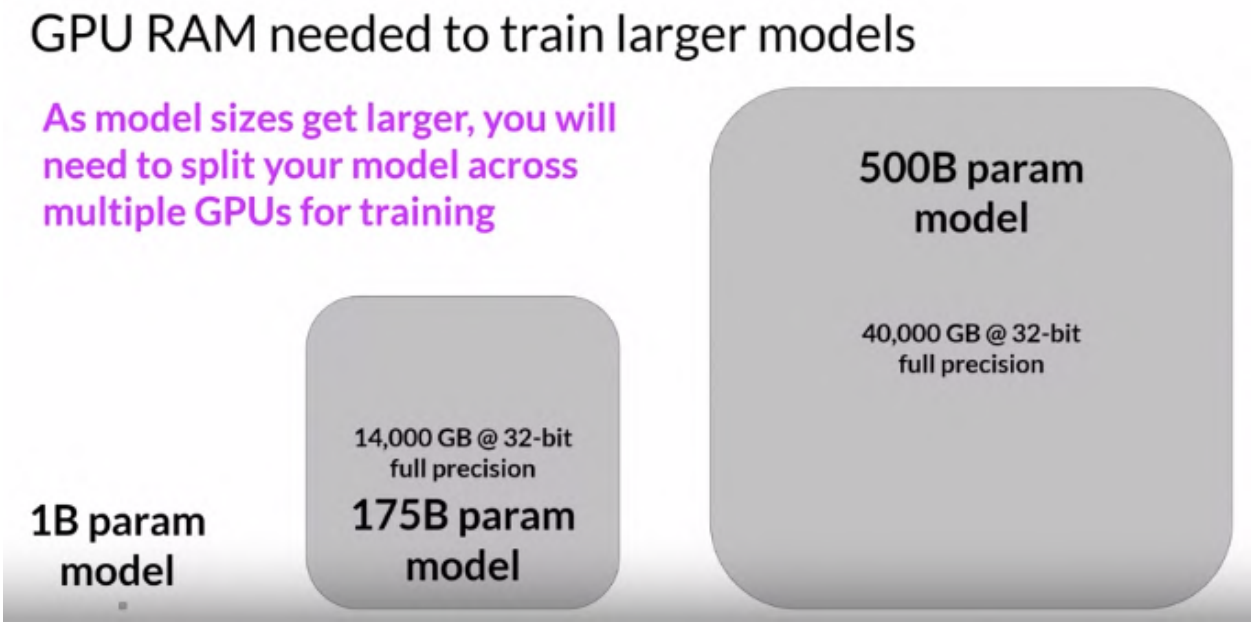
- Reduce required memory to store and train models
- Projects original 32-bit floating point numbers into lower precision spaces
- Quantization-aware training (QAT) learns the quantization scaling factors during training
- BFLOAT16 is a popular choice

Approximate GPU RAM needed to store 1B parameters





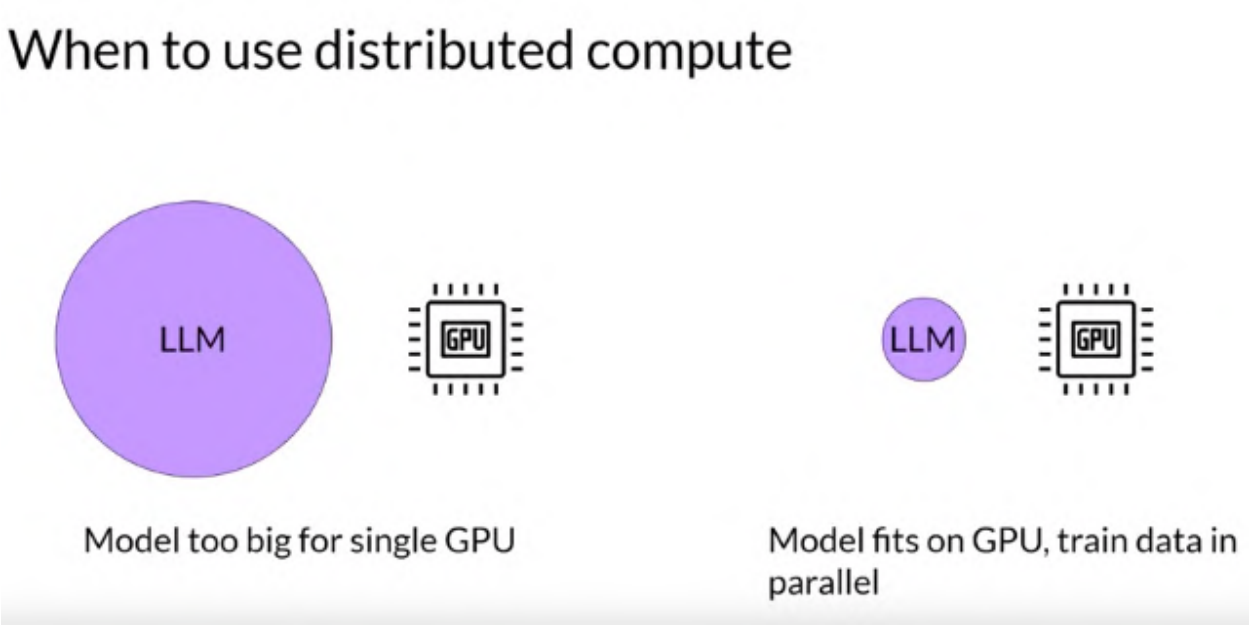
- Impact of Quantization:
  - Quantization can significantly reduce memory consumption during training.
  - With quantization, memory consumption can be reduced to 2GB (16-bit) or 1GB (8-bit) for a one billion parameter model. Note here - still a model with 1 B paramters
  - Models with billions of parameters may require distributed computing techniques and multiple GPUs for training. Have to consider 8bit or 16bit quantization if training on a single GPU



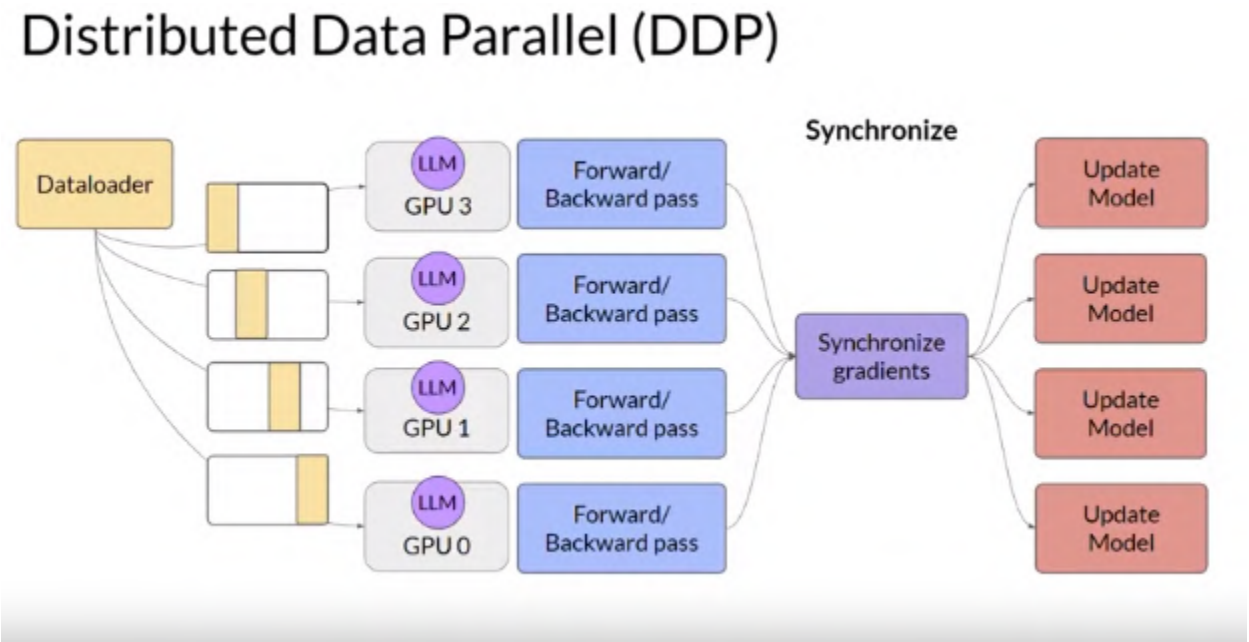
- Training Challenges and Distributed Computing:
  - Training models with billions of parameters may require distributed computing techniques.
  - Distributed training across multiple GPUs becomes necessary for large models.
  - Fine-tuning is another training process that requires memory for all parameters.

- Pre-training models from scratch becomes impractical due to the memory requirements and expenses involved.
- Fine-tuning a pre-trained model is a common approach to adapt models for specific tasks.

Efficient multi-GPU compute strategies

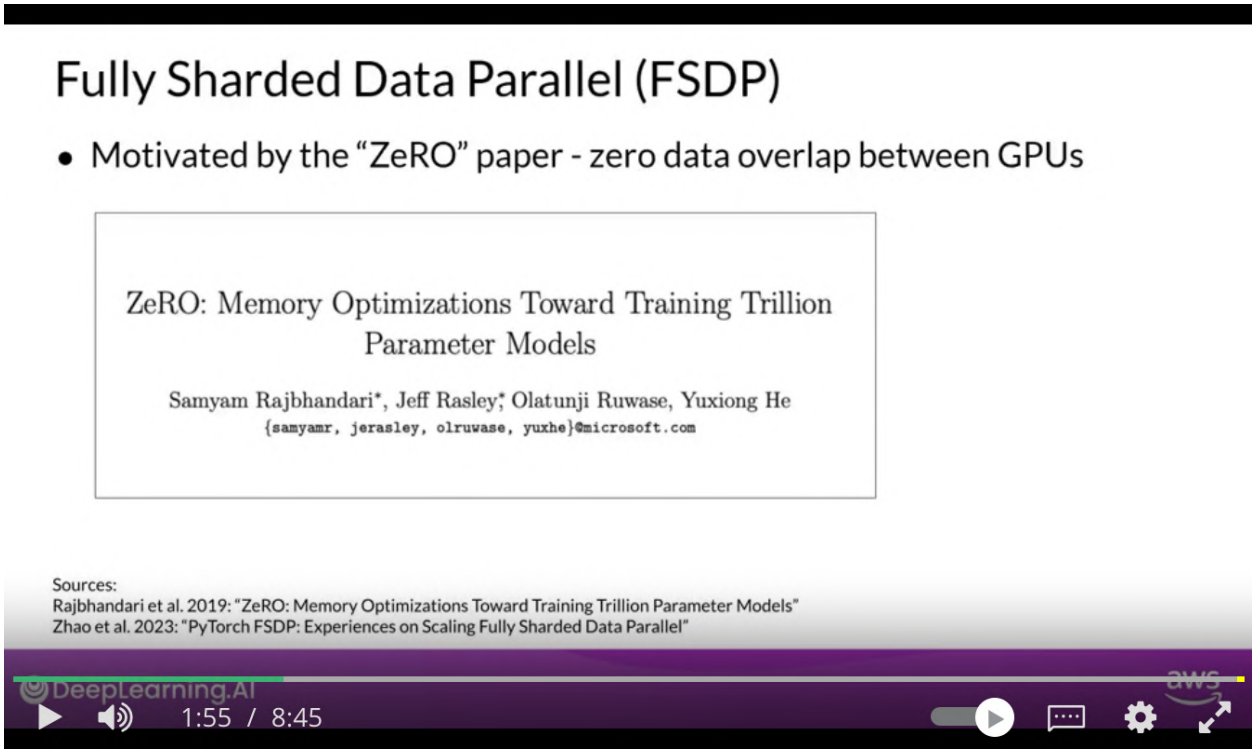


- Scaling Model Training with Multiple GPUs:
  - Scaling model training beyond a single GPU can provide benefits in terms of speed and efficiency.
  - Even for small models, distributing compute across multiple GPUs can be useful.
  - Two techniques for scaling model training: Distributed Data Parallel (DDP) and Fully Sharded Data Parallel (FSDP).



- Distributed Data Parallel (DDP):
Case - mode fits in a single GPU.

- DDP replicates the dataset and model on each GPU and processes batches of data in parallel. Popular implementation of model replication technique is PyToch's DDP. DDP copies models in each GPU and sends batches of data to each of the GPU in parallel. Each dataset is processed in parallel and the synchronization step combines the result of all the GPU. Which in turn updates the model on all GPUs, which(model) is identical across chips. This implementation allows parallel computation across all GPUs that results in faster training.
- Each GPU works on a subset of the data and updates the model in synchronization with other GPUs.
- DDP requires that the model weights and additional parameters fit on a single GPU. If not look into model sharding



- Fully Sharded Data Parallel (FSDP):
- Popular implementation of **model sharding** is PyTorch's Full Shared data parallel.
- Based on Microsoft's paper ZeRO - zero redundancy optimizer. Goal of zero is to optimize memory by distributing model states (sharding) across GPU with zero data overlap. This allows model to scale model training across GPU when model doesn't fit in the memory of a single chip.
- FSDP distributes or shards model parameters, gradients, and optimizer states across GPUs.
- FSDP eliminates redundant memory consumption and allows scaling when the model doesn't fit in a single GPU's memory.

Recap: Additional GPU RAM needed for training

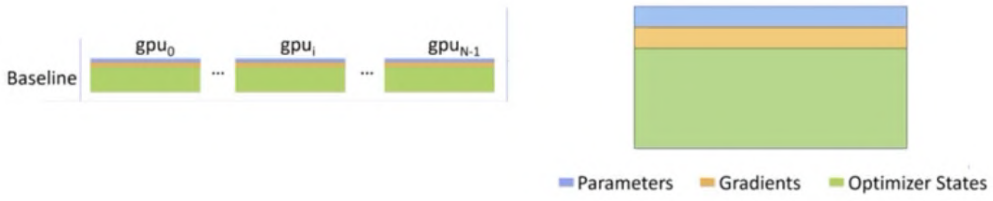
	Bytes per parameter
Model Parameters (Weights)	4 bytes per parameter
Adam optimizer (2 states)	+8 bytes per parameter
Gradients	+4 bytes per parameter
Activations and temp memory (variable size)	+8 bytes per parameter (high-end estimate)
TOTAL	=4 bytes per parameter +20 extra bytes per parameter

Sources: https://huggingface.co/docs/transformers/v4.20.1/en/perf_train_gpu_one#anatomy-of-models-memory, https://github.com/facebookresearch/bitsandbytes

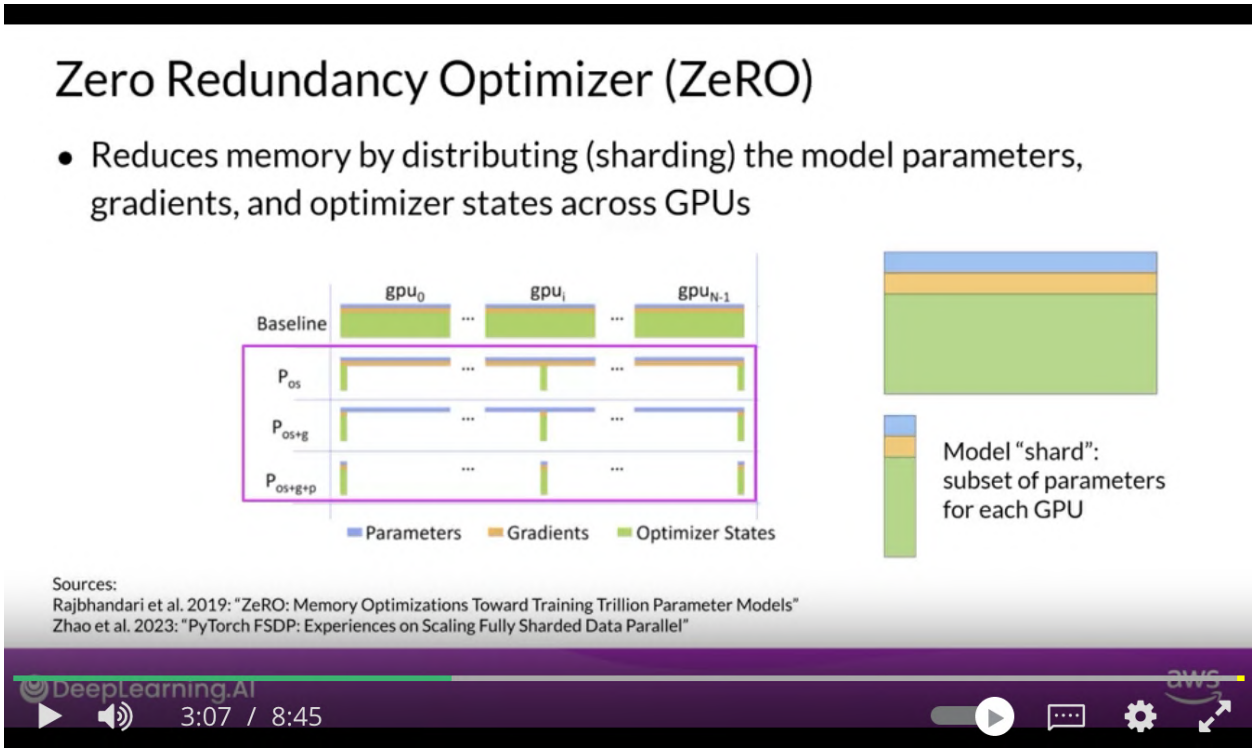
Largest memory requirement is for the optimizer state. Which take up twice the space as weights. Followed by weights themselves and the gradients.

Memory usage in DDP

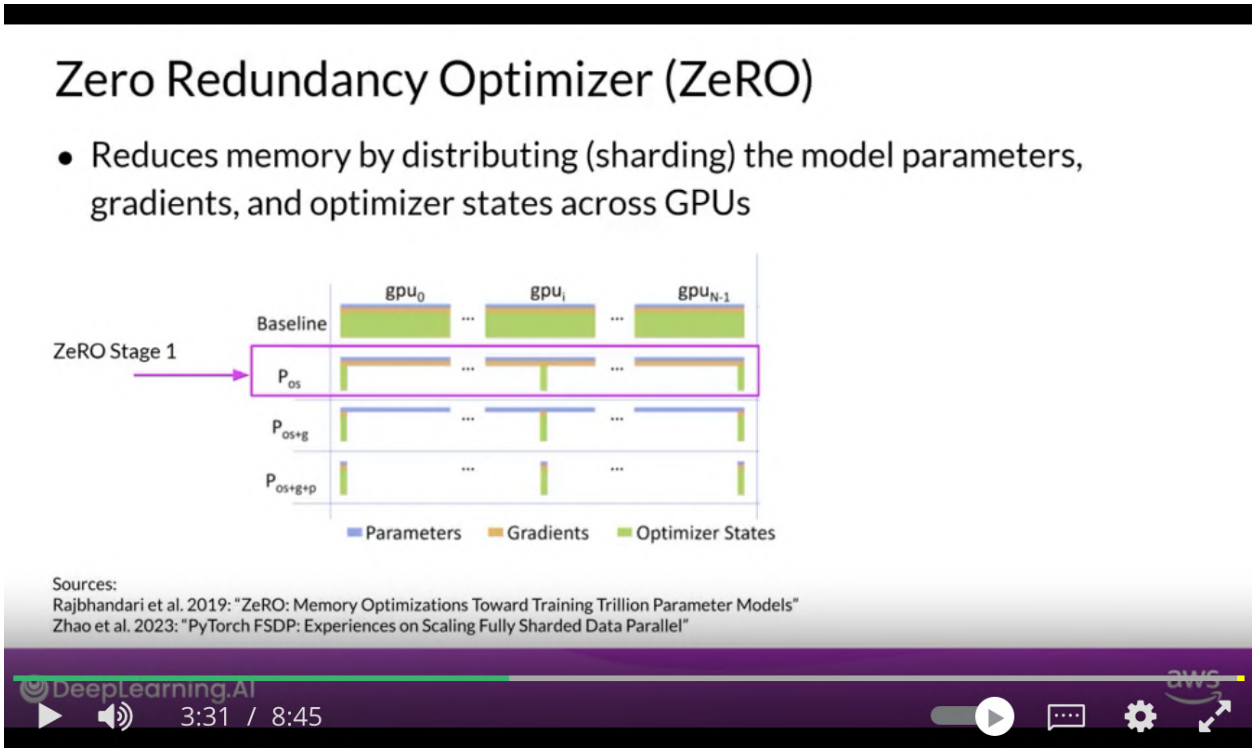
- One full copy of model and training parameters on each GPU



Sources:
Rajbhandari et al. 2019: "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models"
Zhao et al. 2023: "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel"

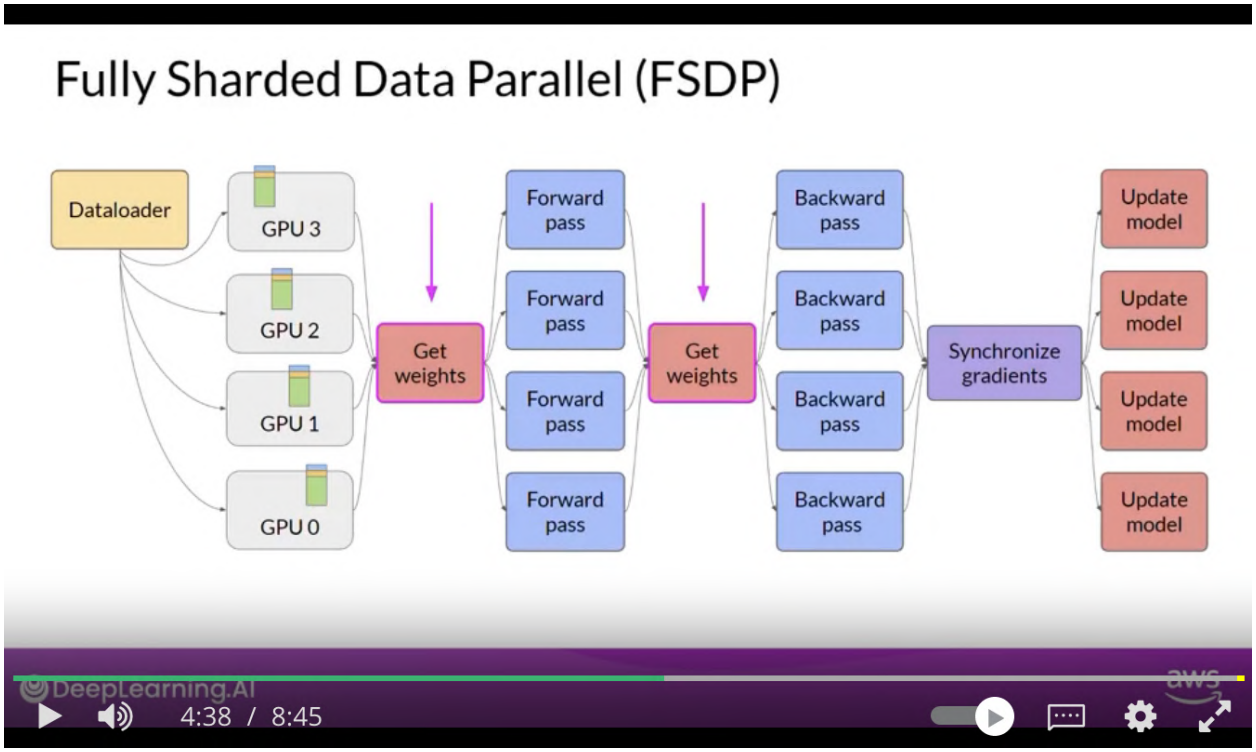


In Need to keep a full copy of the model on each GPU leading to redundant memory consumption. Storing the same numbers in every GPU. Zero eliminates this redundancy by distributing model parameters, gradients and optimizer state across GPUs instead of replicating them. At the same time the communication overhead for syncing model states stays close to as of DDP.



Zero optimizes in 3 stages.
Zero stage 1 - shards only optimizer states across GPUs. This can reduce memory footprint upto a factor of 4.
Zero stage 2 - also shards the gradients across the chips. When applied together with stage 1, this can reduce memory footprint upto 8 times.
Zero stage3 - shards all components including model parameters across GPUs. When applied with stage 1 and stage 2, memory reduction is linear with the number of GPUs. For e.g., sharding across 64 GPU would reduce the memory footprint by a factor of 64.

- FSDP optimizes memory usage with three stages: sharding optimizer states, sharding gradients, and sharding all components.
- Memory reduction is linear with the number of GPUs.



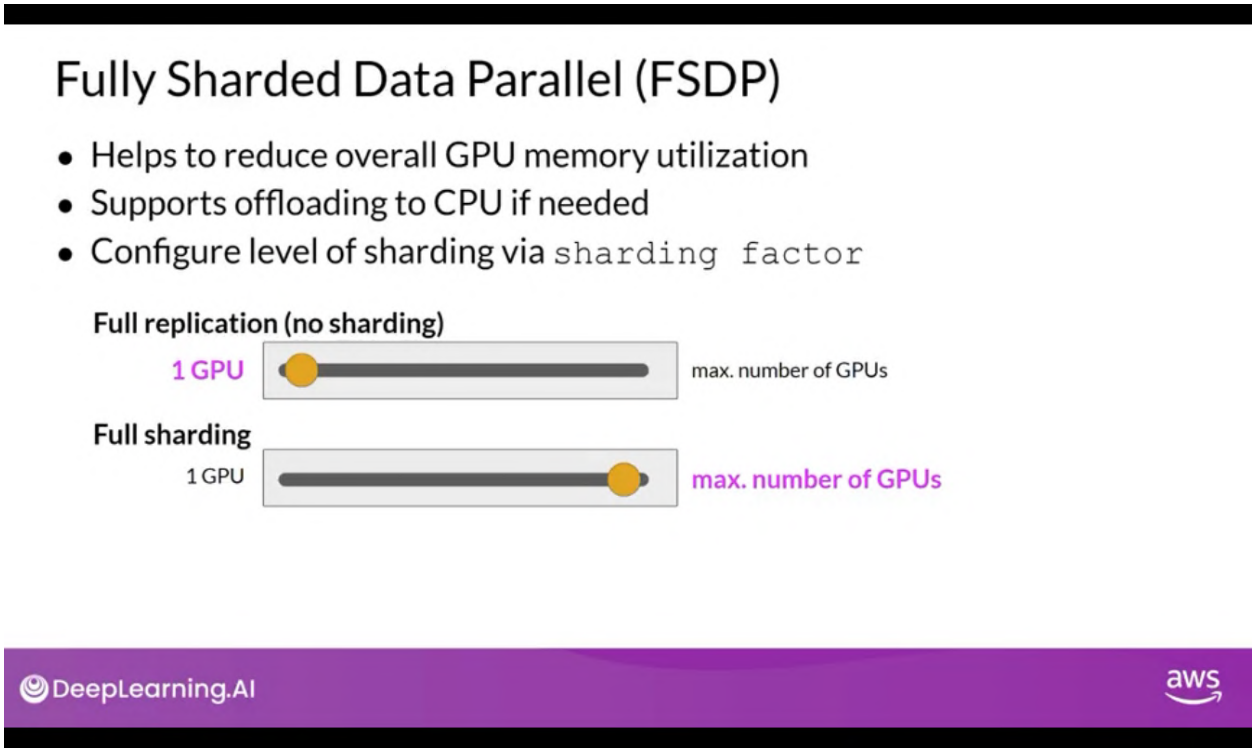
- FSDP Implementation:
  - FSDP distributes data across multiple GPUs and shards model parameters, gradients, and optimizer states.
  - Also shards model parameters, weight across the GPU nodes using one of the strategies specified in the zero paper.

Sharded data is collected from other GPUs when needed for forward and backward passes. With this strategy, can now work with models that are too big to fit on a single chip.

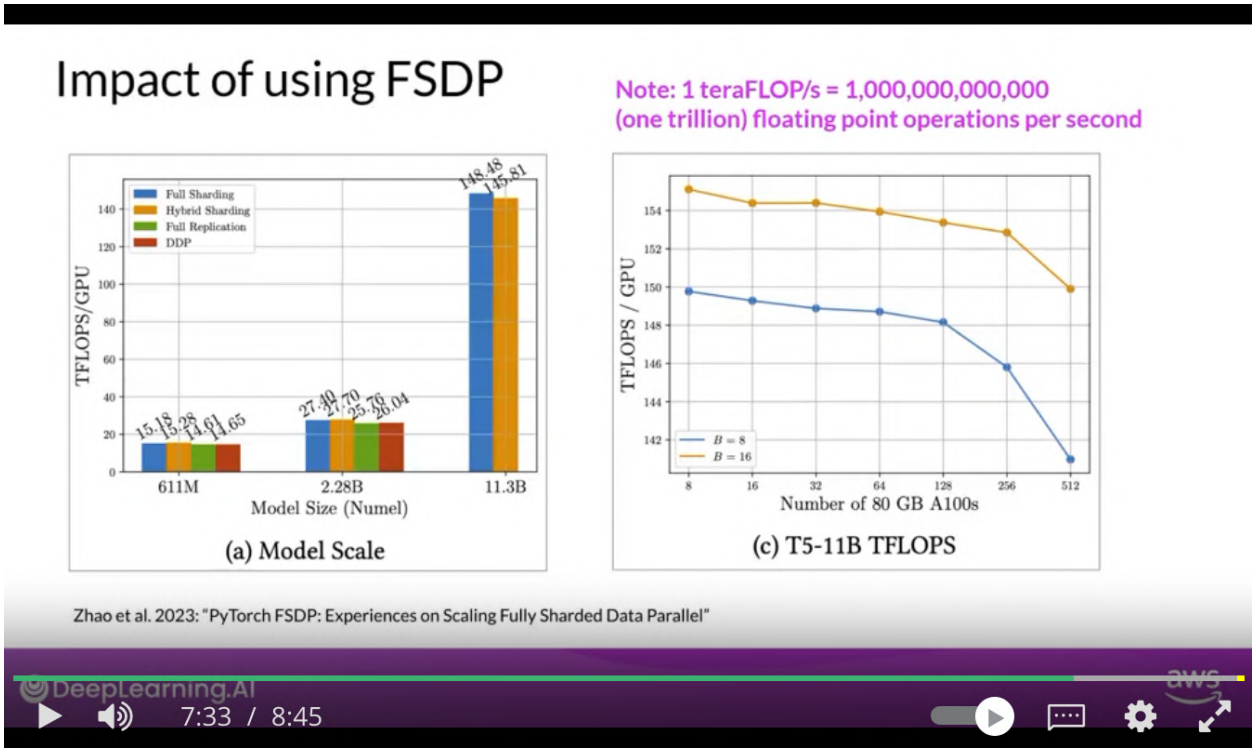
In contrast to DDP where each GPU has all the model states required to process each batch of data locally, FSDP requires you to collect this data from all of the GPUs before the forward and backward pass. Each GPU request data from the other GPUs on demand materializing the shareded data into unsharded data for the duration of the operation.

After operation, release the unsharded non-local data back to the other GPUs as original sharded data. Can also keep it for future operations, for e.g., during backward pass. Note this requires more GPU ram again.

Typical performance vs memory tradeoff decision. In the final step after the backward pass, FSDP schronizes the gradients across GPUs in the same way as they were with DDP.



- FSDP offers a configuration for sharing to manage the trade-off between performance and memory utilization.
- Sharding factor can be configured to control the level of sharding. Sharing factor =1 , removes sharding and replicates the full model similar to DDP. Full sharding = sharding factor to maximum number of available GPUs, turn on full sharding. This has most memory saving but increases the communication volume between GPUs. Any sharding factor in between enales hybrid.



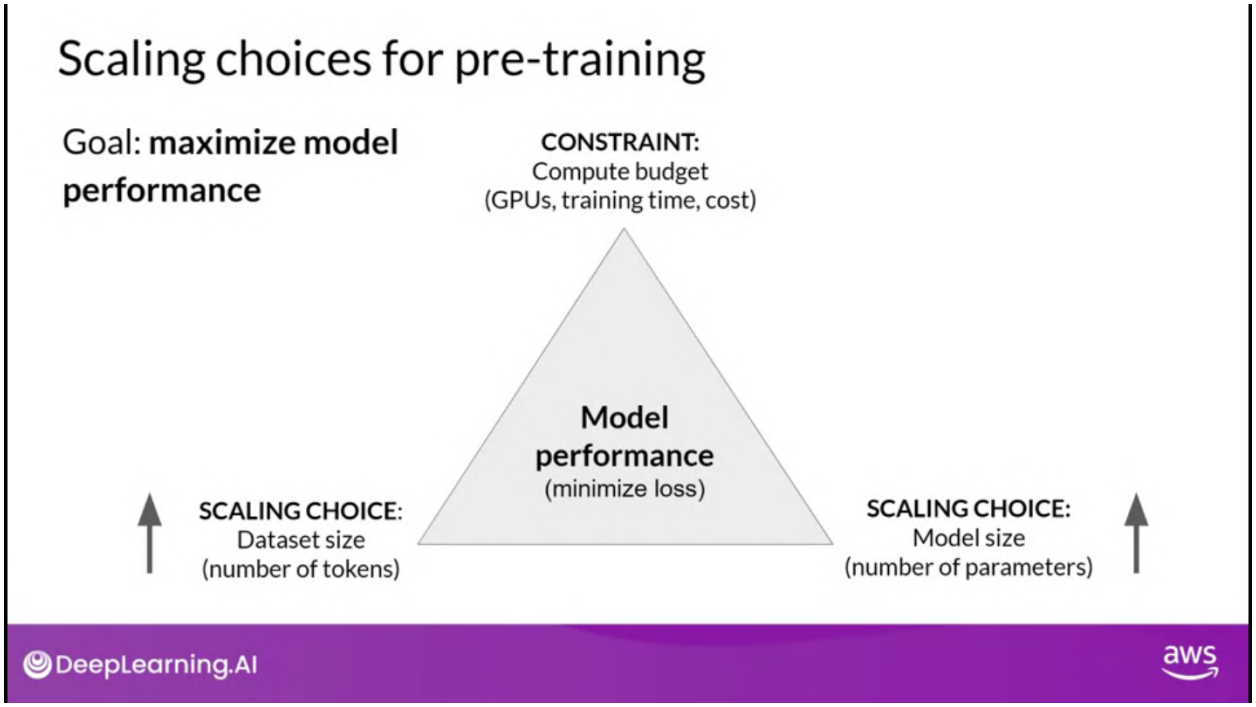
- Performance Comparison of FSDP and DDP:
  - 1 TFLOPS = 1 million floating point operations per second.
  - FSDP can handle small and large models, providing scalability across multiple GPUs. DPP runs out of memory errors.

  - FSDP achieves higher teraflops when reducing model precision to 16-bit. - Increased communication volume between GPUs can impact performance as the model size and number of GPUs increase.

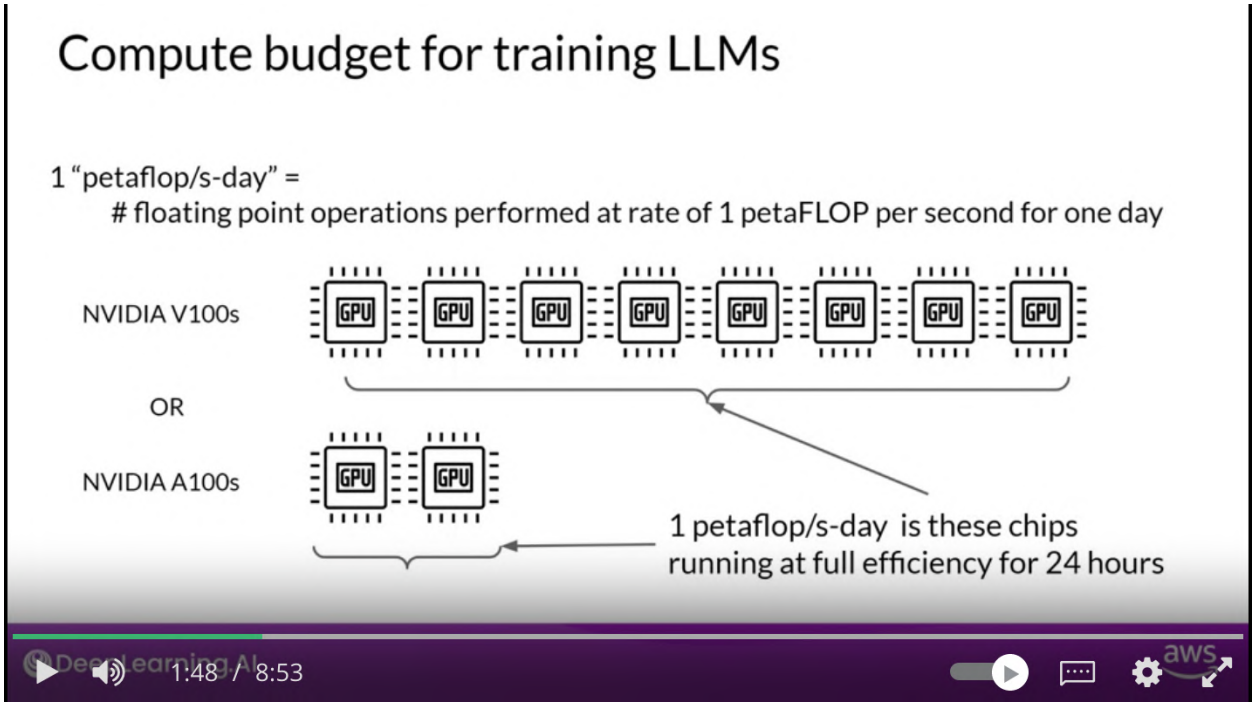
As the model grows in size and is distributed across more and more GPUs, the increase in communication volume between chips starts to impact the performance, slowing down the computation. In summary, can use FSDP can be used for both small and large models and scale your model training across multiple GPUs

- Importance of Understanding Data and Model Sharing:
  - Understanding how data, model parameters, and training computations are shared across processes is crucial for training large language models.

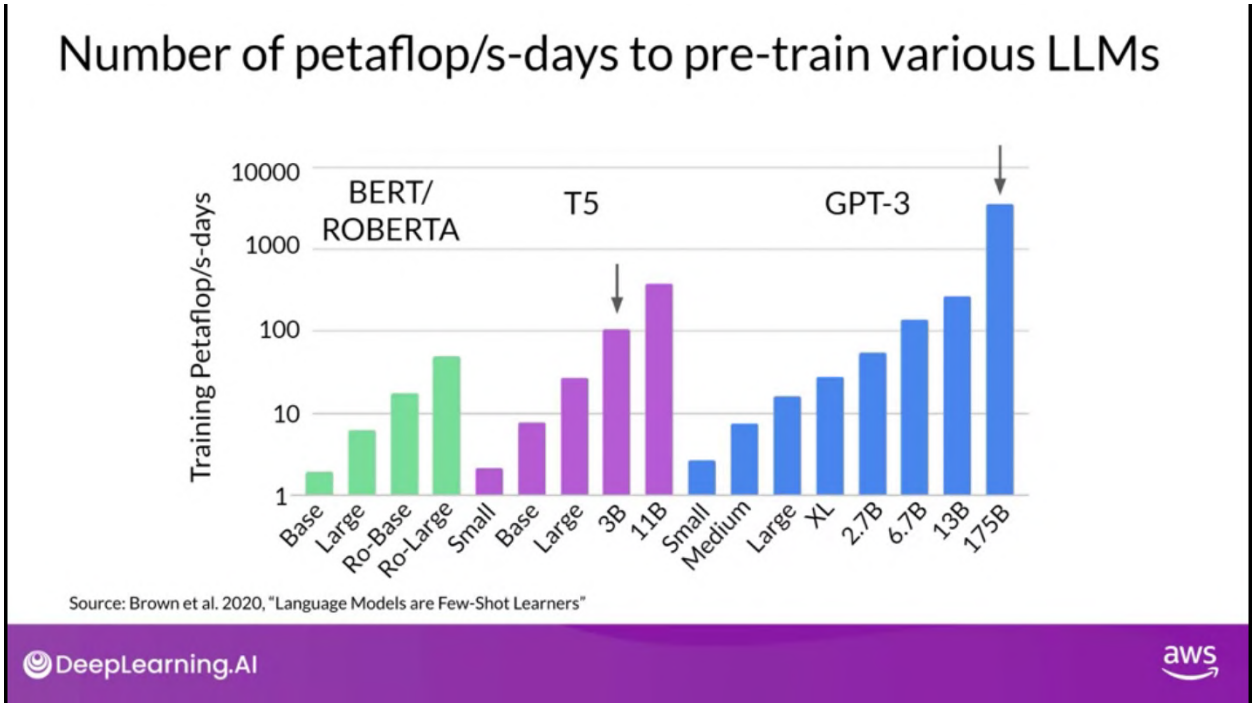
Scaling laws and compute-optimal models



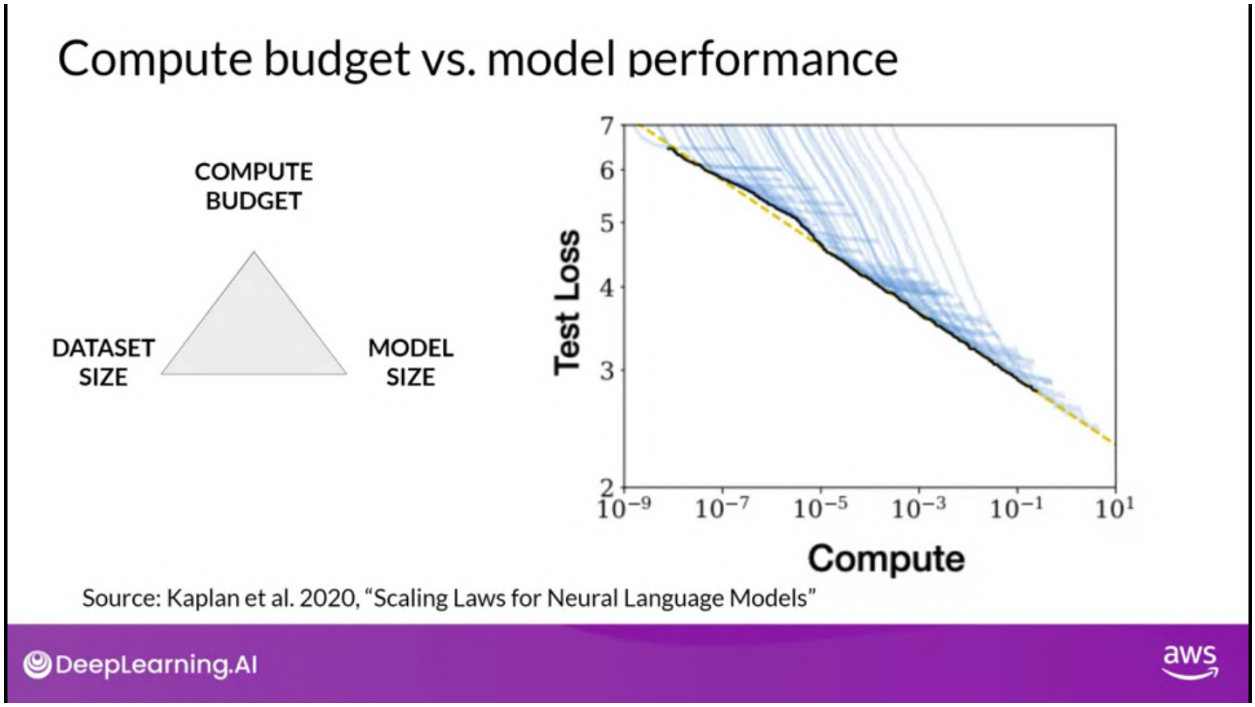
- Relationship Between Model Size, Training Configuration, and Performance:
  - The goal during pre-training is to maximize model performance by minimizing loss when predicting tokens.
  - Two options to improve performance: increase dataset size and increase the number of model parameters.
  - Compute budget, including factors like GPU availability and training time, is a crucial consideration.



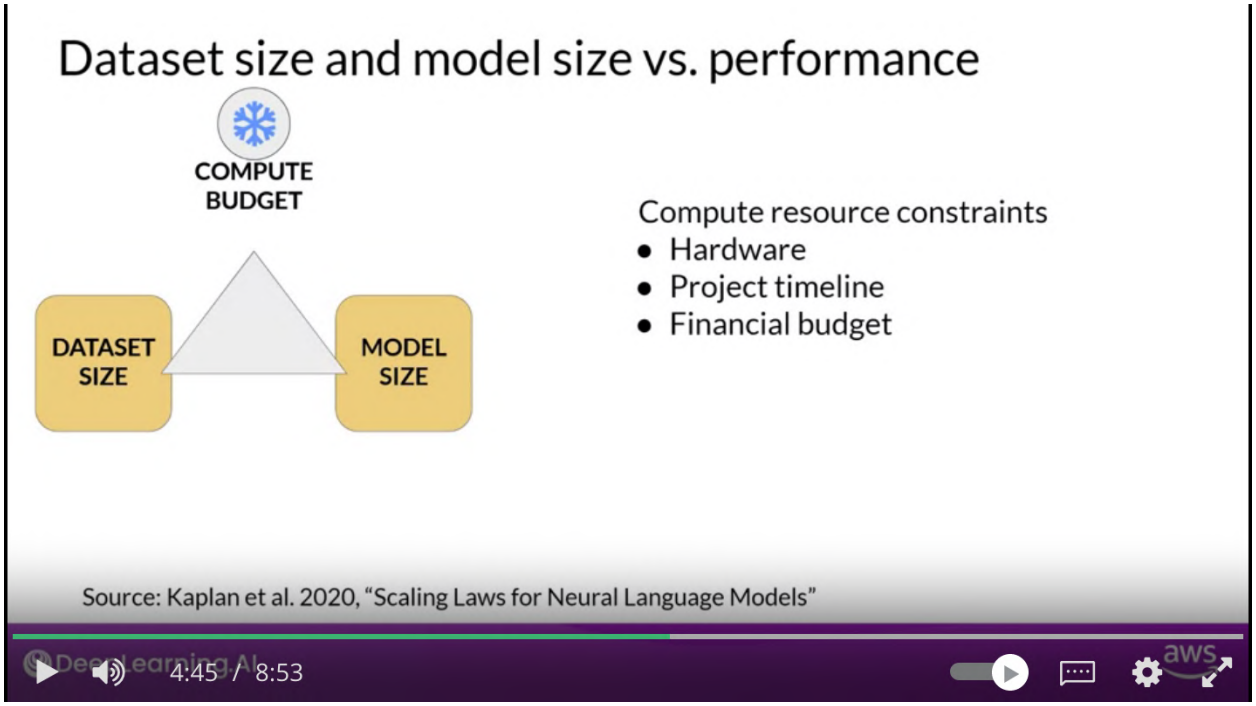
- Understanding Compute Budget:
  - Compute budget can be quantified by **petaFLOP per second days, measuring the number of floating point operations performed.**
  - 1 peta flots per day is approximately equivalent to 8 Nvidia V100s, operating at full efficiency for 1 full day. More powerful processor requires fewer chipsChips to
  - Larger models require more compute resources, and the scale is exponential.



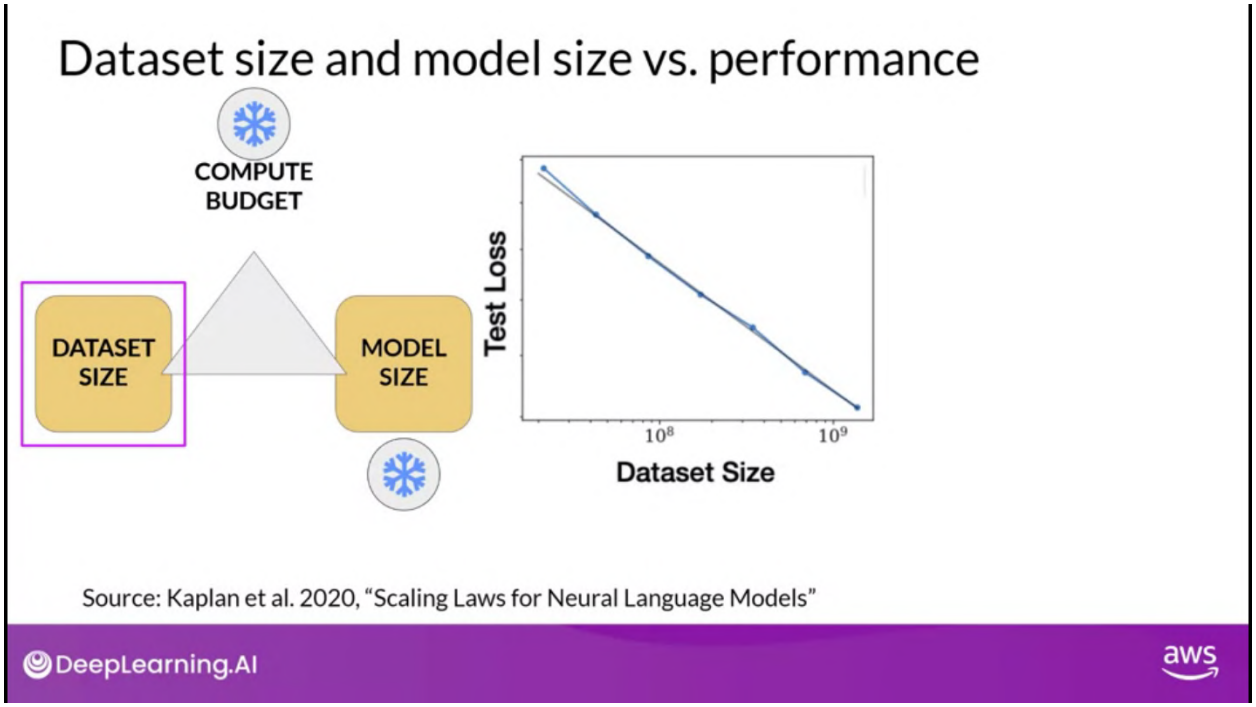
BERT base few hundred million, to 175B for GPT. Y axis in log. Huge compute required.

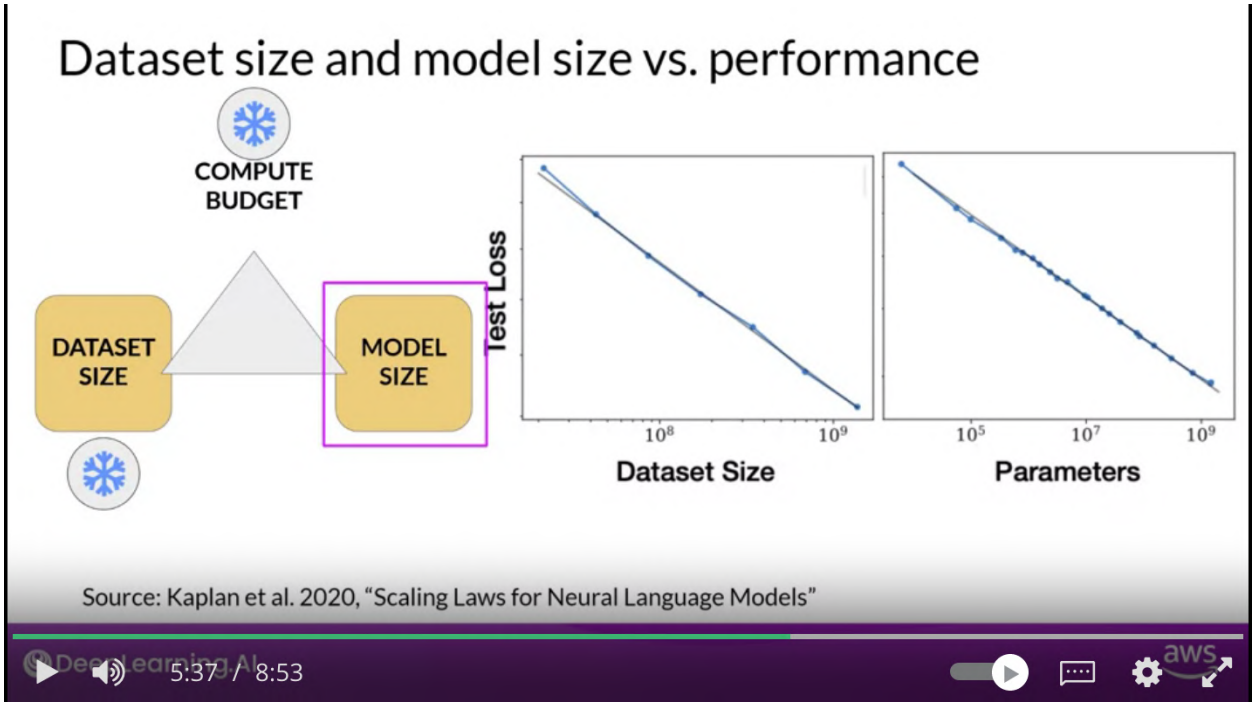


- Power-Law Relationship:
  - Researchers have explored the relationship between compute budget, model size, training dataset size, and model performance.
  - Power-law relationships exist between these variables, indicating that increasing compute budget improves performance. Y axis -test loss. Smaller better. X axis - compute budget in PFpsd. Larger numbers can be achieved by either using more compute power or training for longer or both. Each blue line shows the model loss over the single training run. Loss declines slowly for each run, reveals a clear relationship between the compute budget and the model's performance. This can be approximated by a power law relationship shown by the pink line. A power law is a mathematical relationship between 2 variables where 1 is proportional to the other raised to some power. When plotted on graph where both axis are logarithmic, power law relationships appear as straight line. The relationship here holds as long as model size and training data size don't inhibit the training process. Taken at facevalue, you can increase the compute budget to achieve better performance.



In practise, the compute resources available for training will generally be a hard constraint. Set by factor 1. Hardware you have access to 2. Time available for training 3. Financial budget of the project.





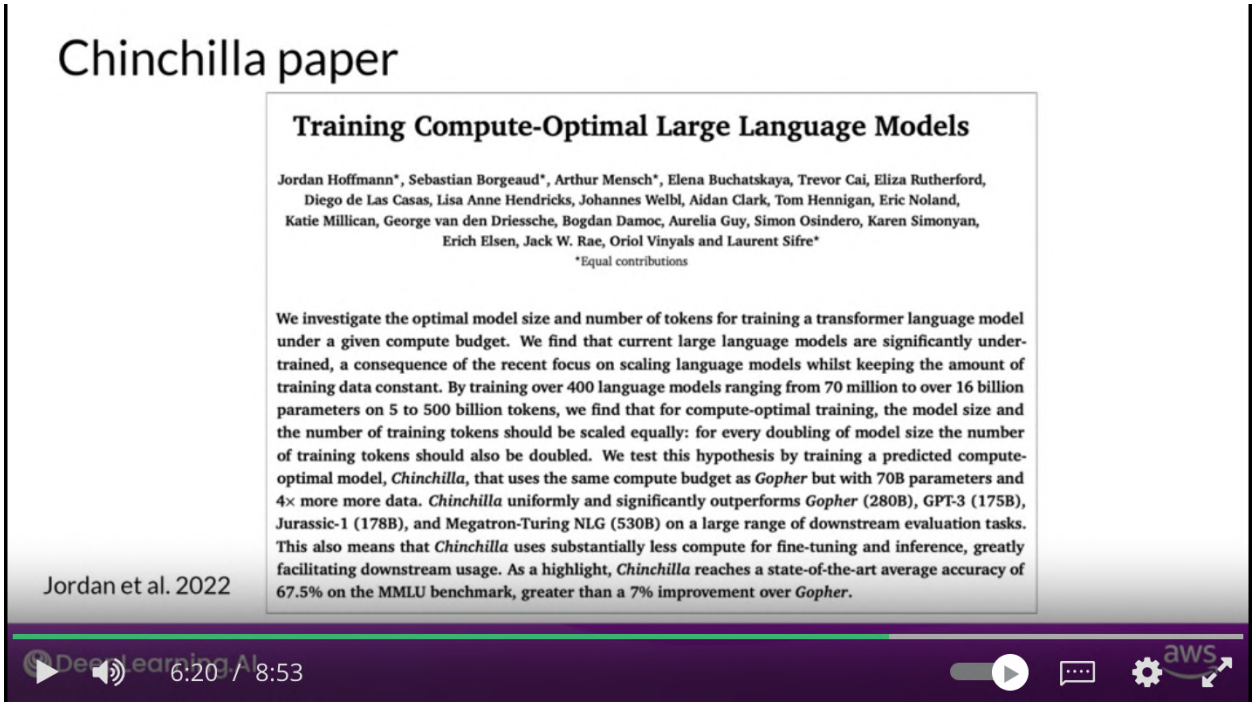
If compute budget is fixed, the 2 levers you have to improve the model performance us to

1. Size of the training dataset
2. Number of parameters in the model

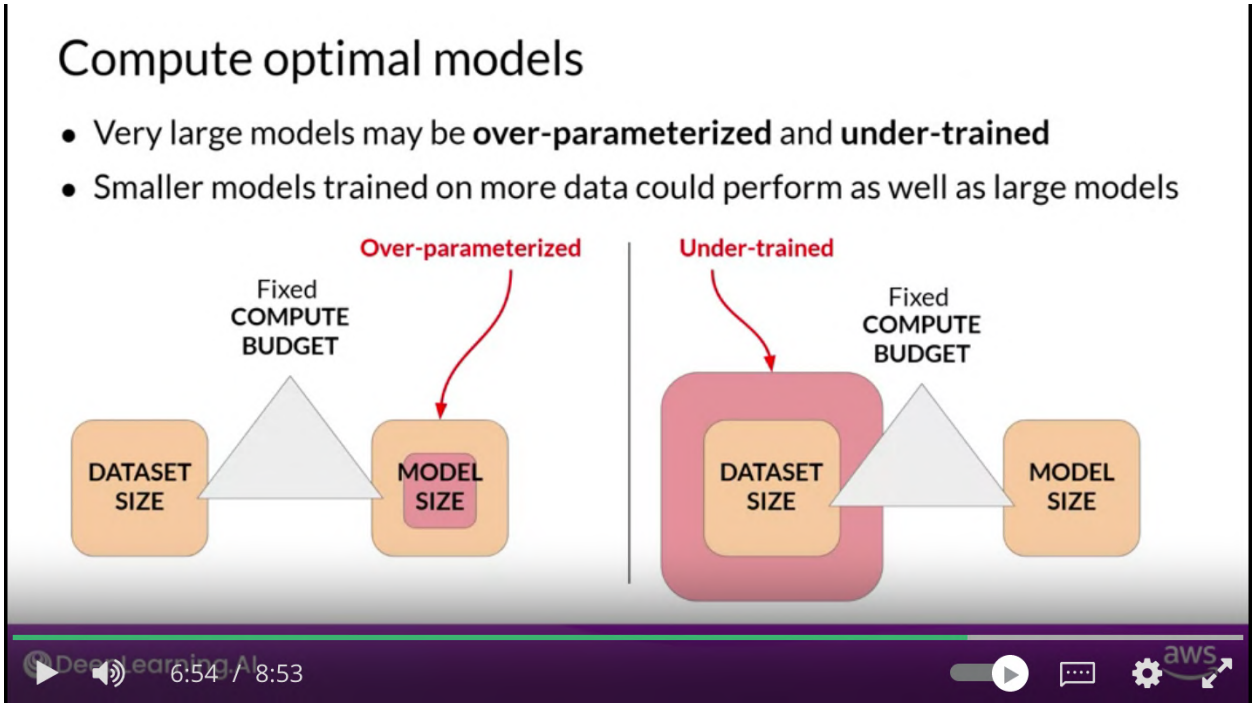
Model size and dataset size also show power law relationship with the test loss. In the case where other 2 variables are held fixed.

- However, practical constraints on compute resources make dataset size and model parameters crucial for performance improvement.

- Impact of Training Dataset Size and Model Parameters:
  - Increasing training dataset size leads to improved model performance.
  - Increasing model parameters also leads to better performance.
  - Empirical data shows the power-law relationship between dataset size, model parameters, and test loss.

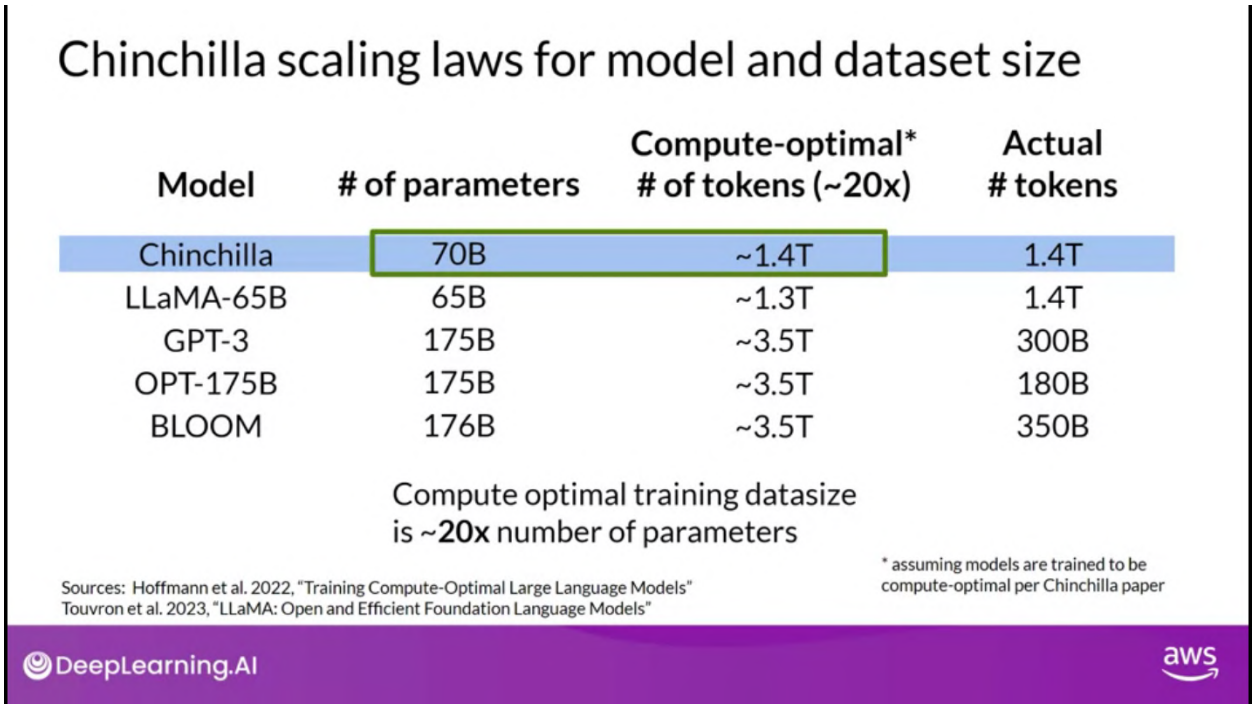


What is the ideal balance between 3 quantities? Optimum number of parameters and the volume of training data for a given compute budget.
Compute optimum model - Chinchilla

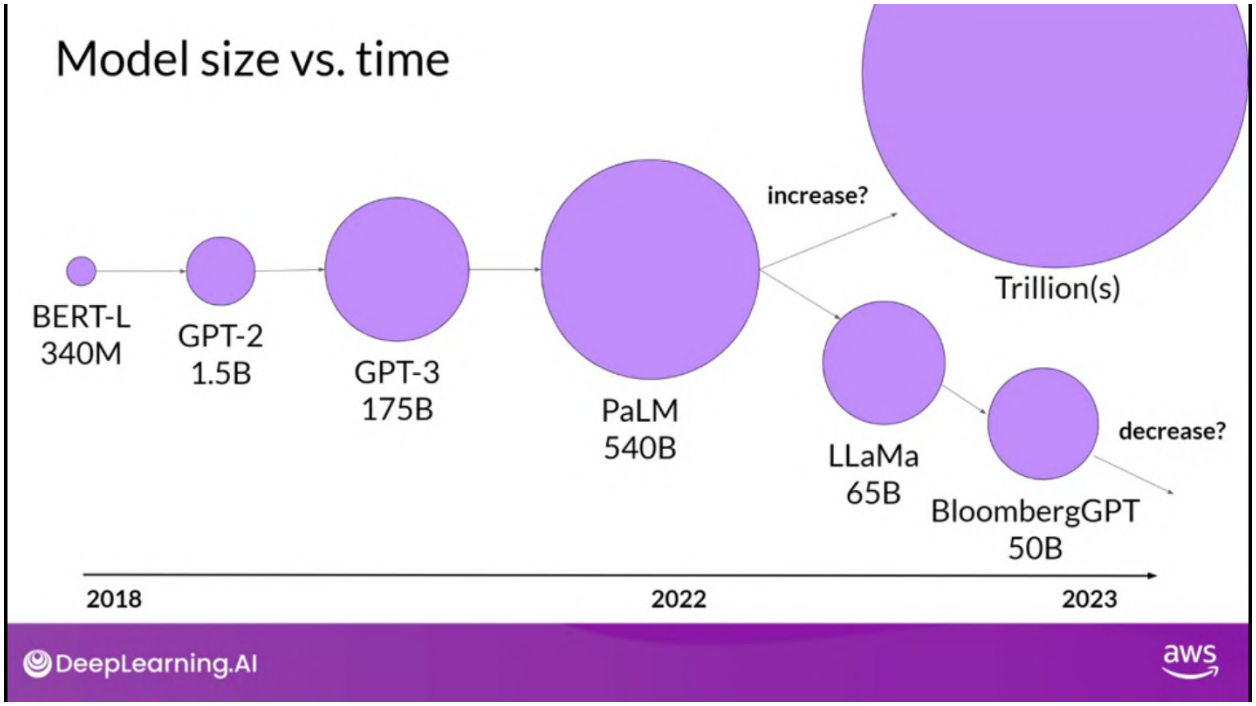


Compute Optimal Models:
- The "Chinchilla" paper studied the optimal number of parameters and training dataset size for a given compute budget.

- Hints that many of the 100B parameters large language models like GPT-3 may actually be overparameterized (they have more parameters than they need, to achieve a good understanding of the language) and undertrained so that they would benefit from seeing more training data.



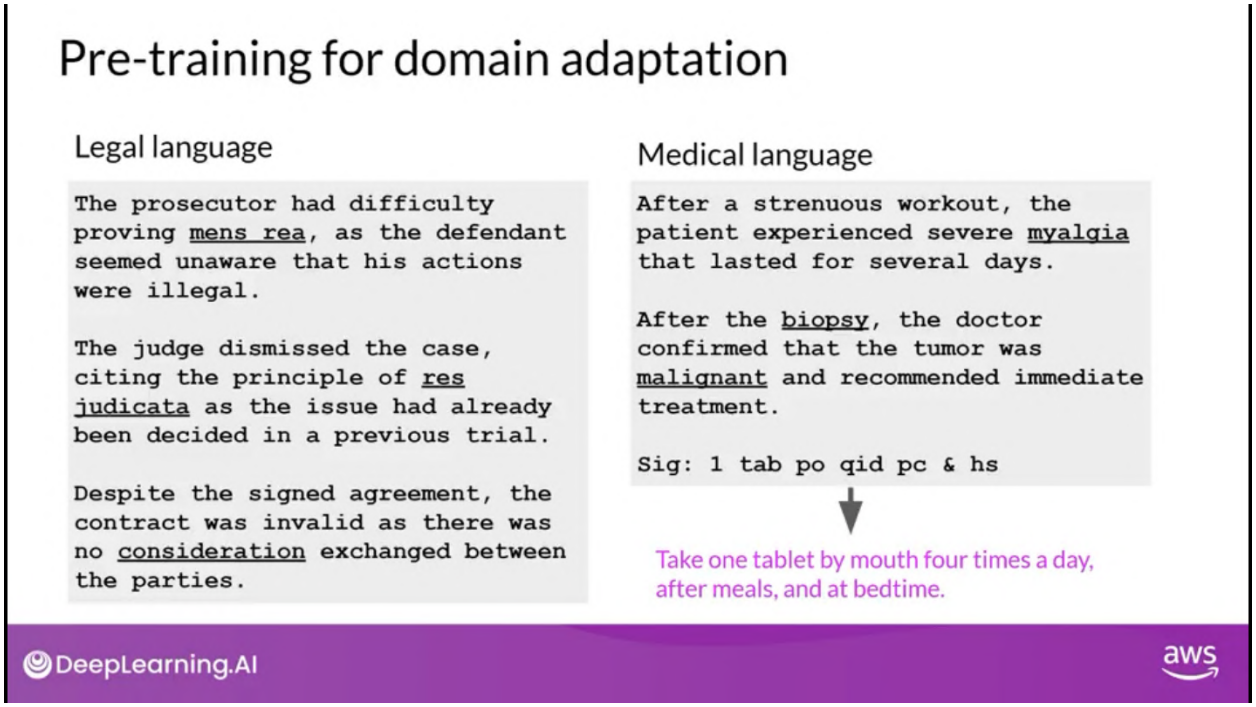
- **Smaller models trained on larger datasets** can achieve similar or better performance than larger non-optimal models.
- The **optimal training dataset size is about 20 times larger than the number of parameters in the model.**



With Chinchilla paper, teams now implementing smaller models.

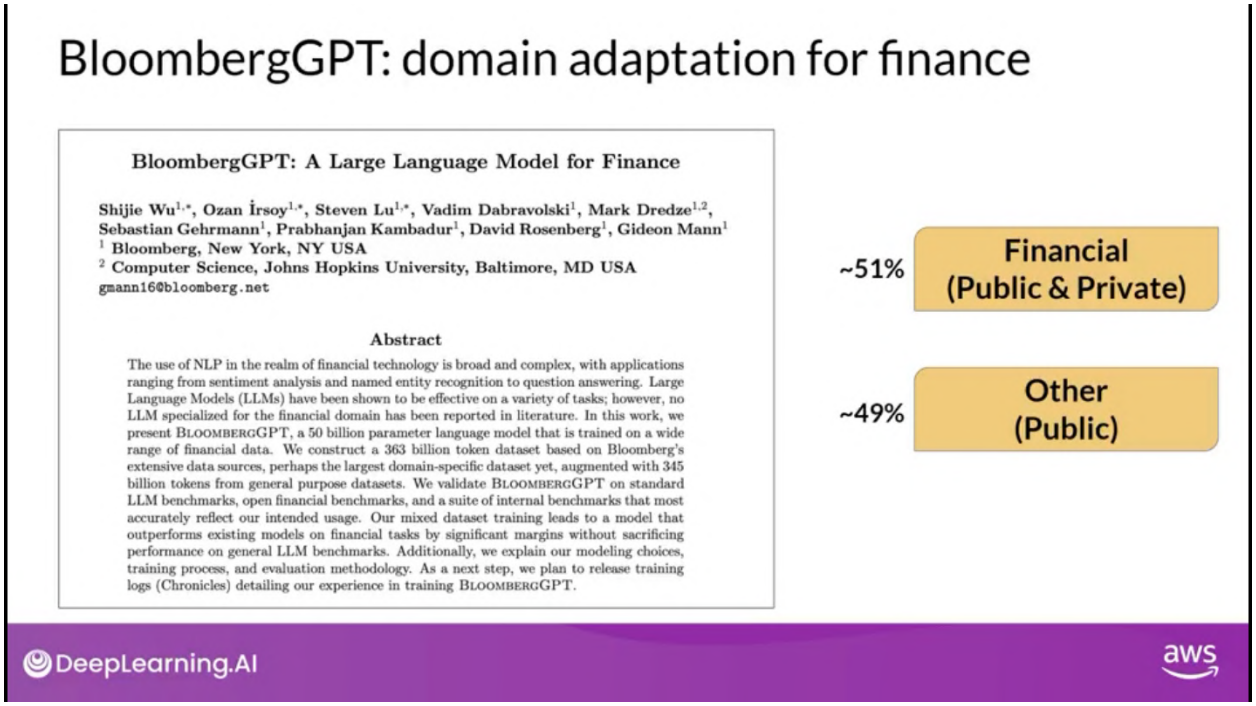
- Importance of Compute Optimal Model Design:
  - Compute optimal models outperform non-optimal models on downstream evaluation tasks.
  - Teams are developing smaller models that achieve comparable or better results than larger non-optimal models.
  - Compute optimal models challenges the notion that bigger is always better.

Pre-training for domain adaptation



- Pretraining for Specific Domains:
  - Pretraining your own model from scratch **may be necessary when working with specialized domains that have unique vocabulary and language structures**.
  - **Domain adaptation** is needed to achieve good model performance in these cases.
  - Examples include legal writing with specific legal terms, medical language with uncommon medical terms, and domains that use language idiosyncratically. Different meaning - for e.g,. Consideration means different meaning

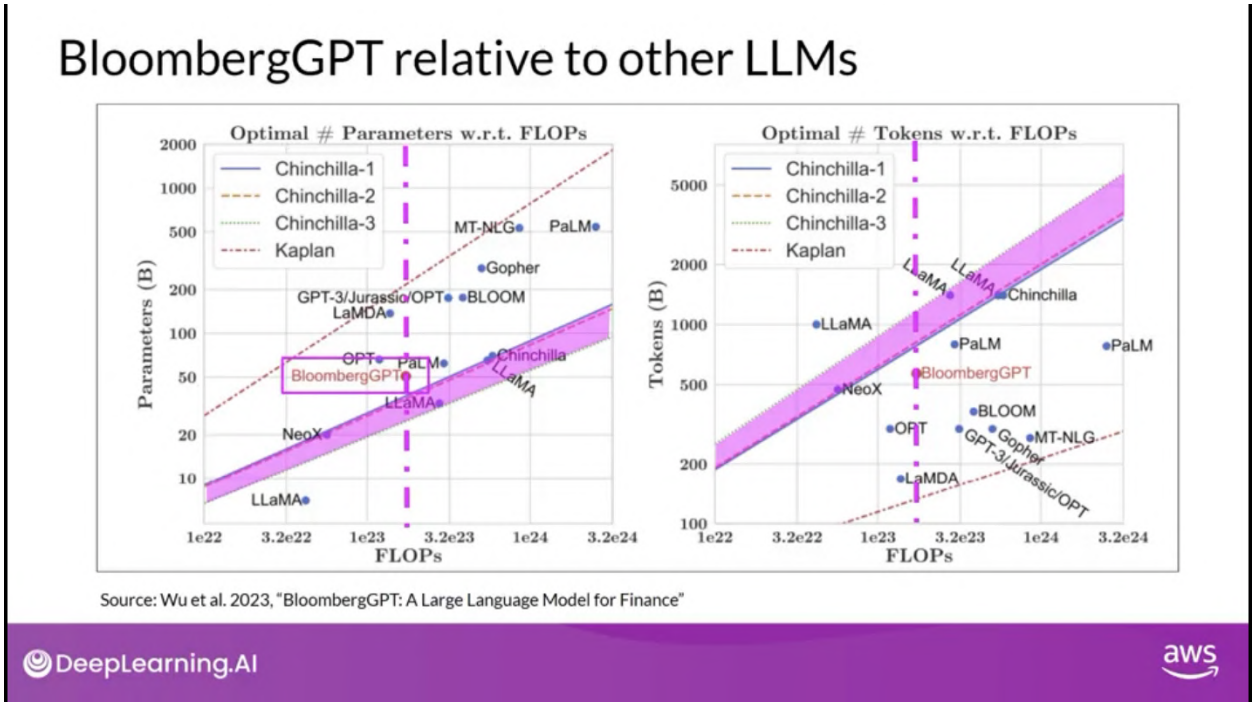
Pre training required for Law, Finance or science



BloombergGPT: Pretraining for Finance:

- BloombergGPT is an example of a large language model pretrained for a specific domain, finance.
- The model combines finance data and general-purpose text data for training.
- The researchers made trade-offs based on available compute budget and the availability of domain-specific data.

- Importance of Pre Training for Specialized Domains:
  - Pre Training models from scratch in specialized domains like law, medicine, finance, or science yields better models.
  - Existing language models may struggle with domain-specific vocabulary and language usage.



- Compute Budget and Model Size:
  - Compute budget, measured in petaFLOP per second days, plays a crucial role in determining model size and performance.
  - The Chinchilla paper provides guidance on the optimal number of parameters and training dataset size for a given compute budget.

- Trade-Offs in Model Design:
  - Real-world constraints may require trade-offs in pretraining models, such as limited availability of domain-specific data.
  - BloombergGPT demonstrates a close adherence to optimal model size while using a smaller-than-optimal training dataset.

- Week One Recap:
  - Covered common use cases for large language models and the transformer architecture.
  - Explored parameters influencing model output at inference time.
  - Discussed the generative AI project lifecycle for application development.
  - Learned about pretraining models on vast amounts of text data and the computational challenges involved.
  - Introduced scaling laws and the concept of compute optimal models.

Domain-specific training: BloombergGPT

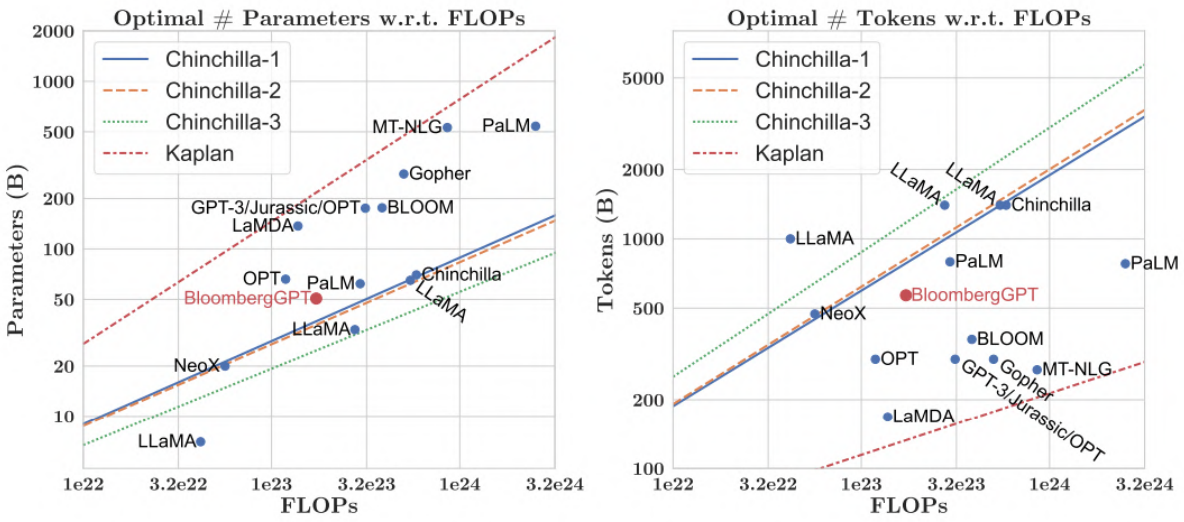


Figure 1: Kaplan et al. (2020) and Chinchilla scaling laws with prior large language model and BLOOMBERGGPT parameter and data sizes. We adopt the style from Hoffmann et al. (2022).

2 Dataset

- BloombergGPT:
  - Developed by Bloomberg, it is a large Decoder-only language model.
  - Pretrained using an extensive financial dataset to increase its understanding of finance and generate finance-related natural language text.

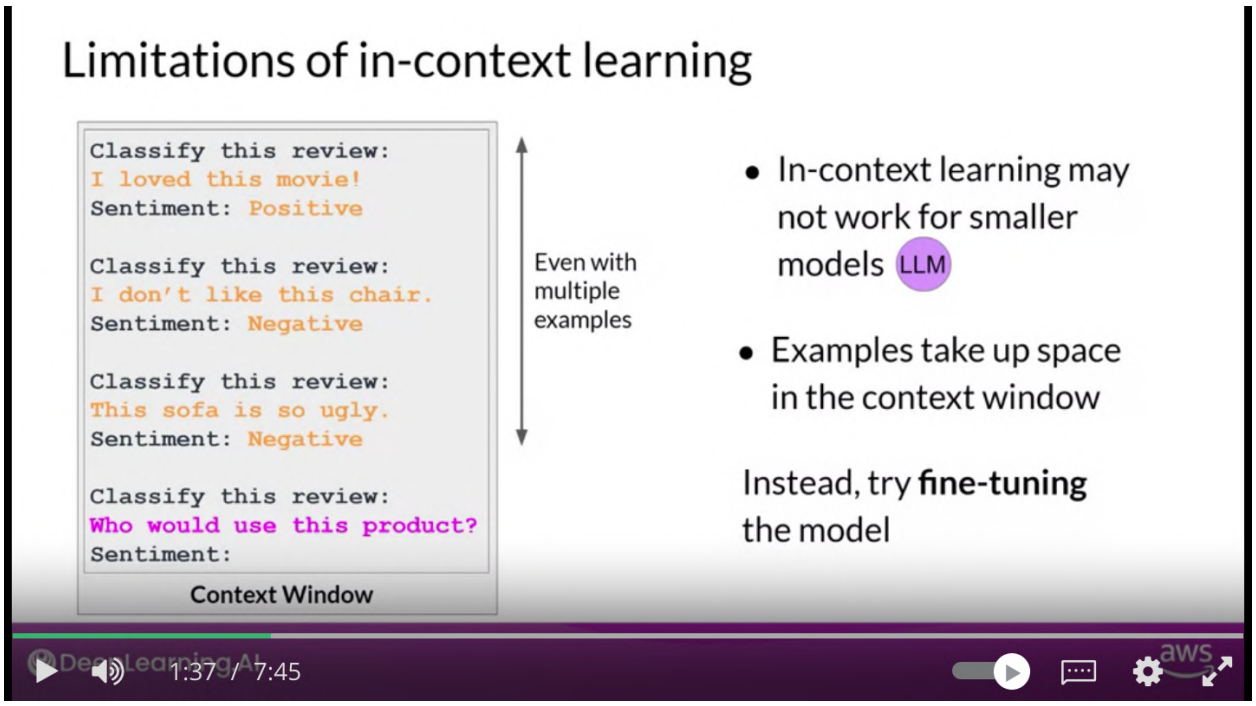
- Pretraining Configuration:

  - The Chinchilla Scaling Laws were used to guide the number of parameters and volume of training data for BloombergGPT.
  - The lines Chinchilla-1, Chinchilla-2, and Chinchilla-3 represent the recommendations of Chinchilla in the image.
  - BloombergGPT closely follows the Chinchilla recommendations.

- Challenges in Training Configuration:
  - The recommended configuration for the available training compute budget was 50 billion parameters and 1.4 trillion tokens.
  - However, acquiring 1.4 trillion tokens of training data in the finance domain was challenging.
  - The team constructed a dataset with only 700 billion tokens, which is below the compute-optimal value.
  - Due to early stopping, the training process terminated after processing 569 billion tokens.

- Trade-Offs in Model and Training Configuration:
  - The BloombergGPT project highlights the trade-offs made when pretraining a model for increased domain-specificity.
  - Real-world challenges, such as limited availability of domain-specific training data, can impact the compute-optimal model and training configurations.

W2 - Fine Tuning LLM with instructions

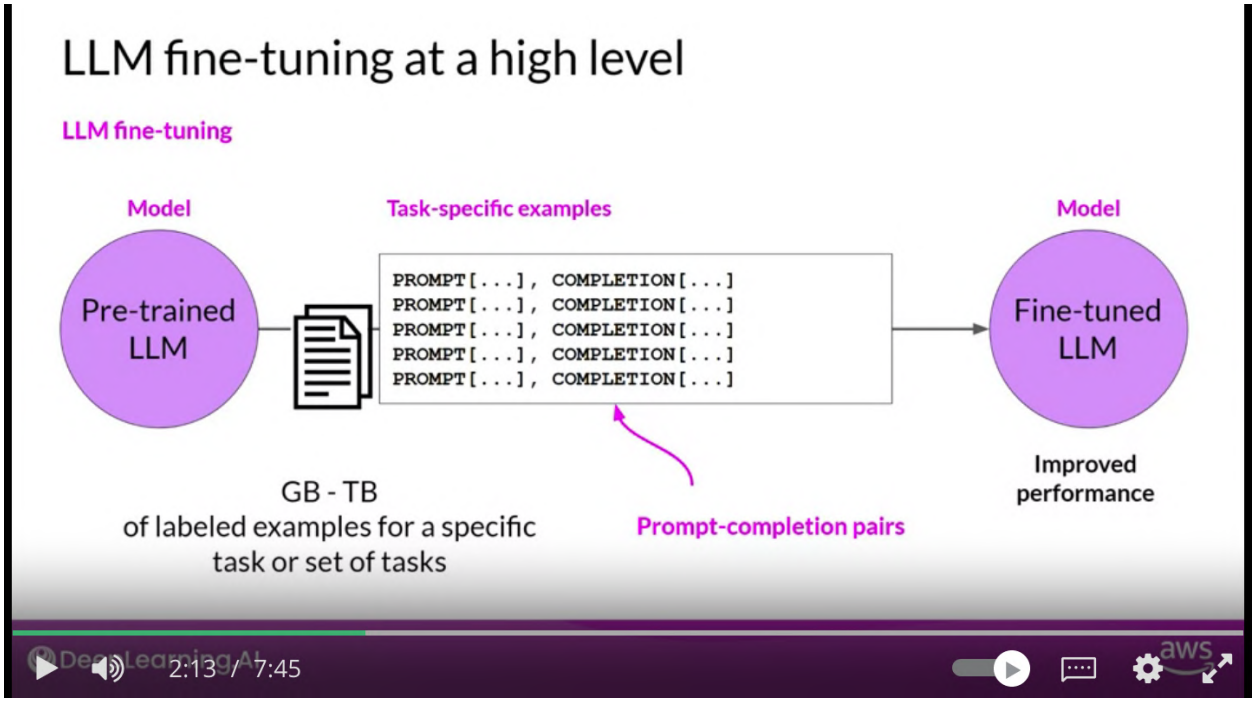


Smaller not good with few shot In context learning. Hence need to fine-tune them.

Pre-training is self supervised trained on vast amount of text

Fine-tuning is supervised learning. Use a dataset of labeled examples to update the weights of the LLM. The labeled examples are prompt completion pairs. Fine tuning extends training of the model to improve its ability to generate good completion for a specific task.

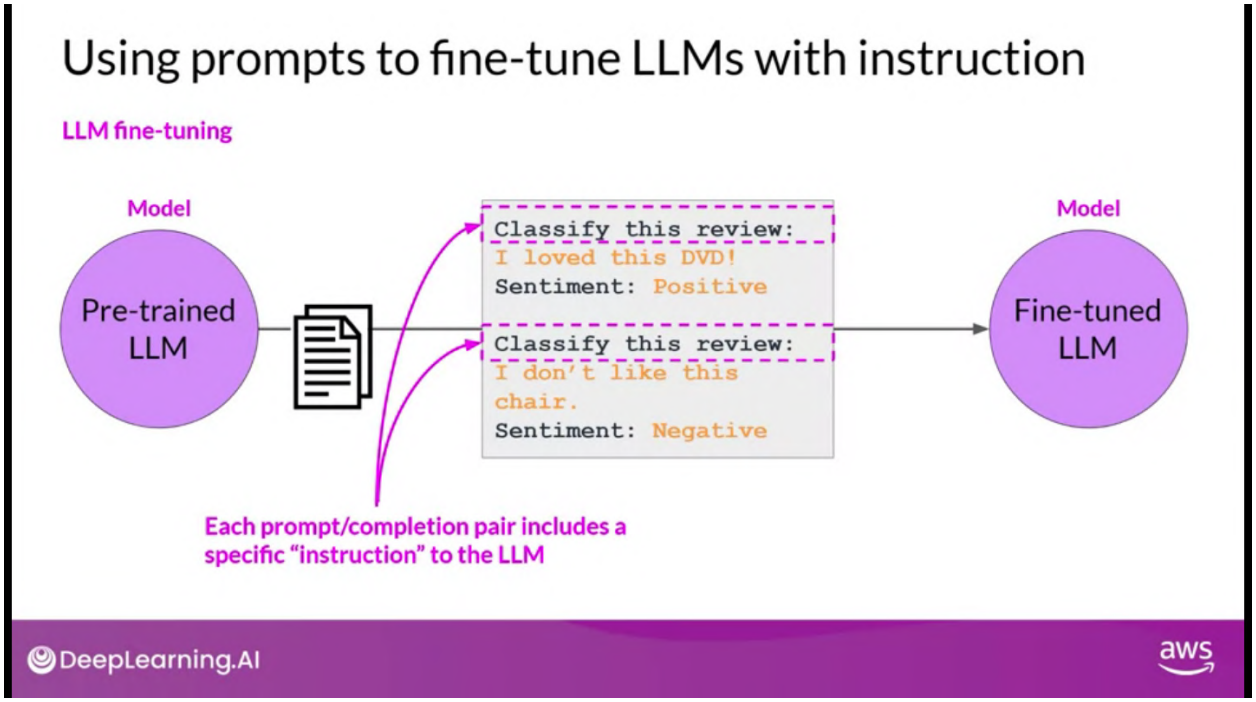
Instruction fine tuning is particularly good in improving the model's performance on a variety of tasks.



2. Methods to Improve Model Performance:
   - Introduction to methods for improving performance of existing models.
   - Focus on improving performance for specific use cases.
   - Importance of evaluating and quantifying model improvement.

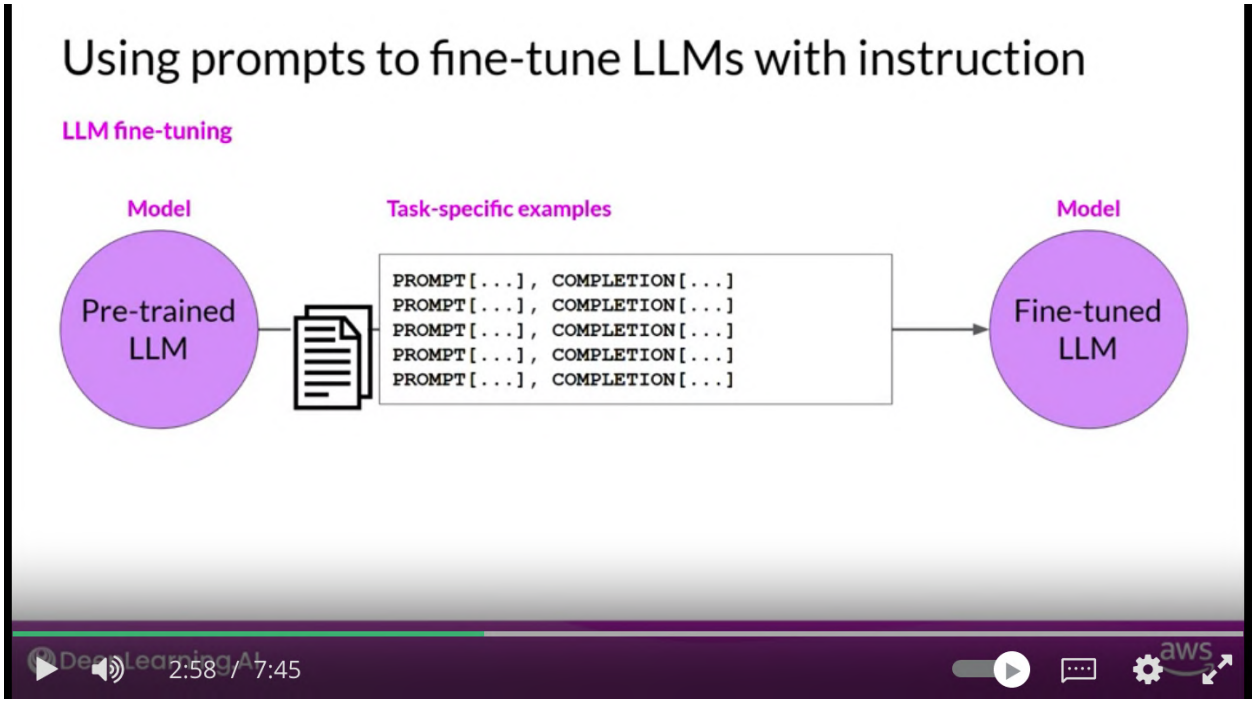
3. Fine-Tuning an LLM with Instruction Prompts:
   - Fine-tuning process to enhance model performance.
   - Differences between models in identifying instructions and carrying out tasks.
   - One-shot or few-shot inference using examples to help model understand tasks.
   - Drawbacks of including examples in prompts (limited space).

4. Fine-Tuning Process:
   - Contrasting fine-tuning with pre-training.
   - Fine-tuning as a supervised learning process using labeled examples.
   - Prompt completion pairs as training data for fine-tuning.
   - Instruction fine-tuning strategy for improving model performance on various tasks.

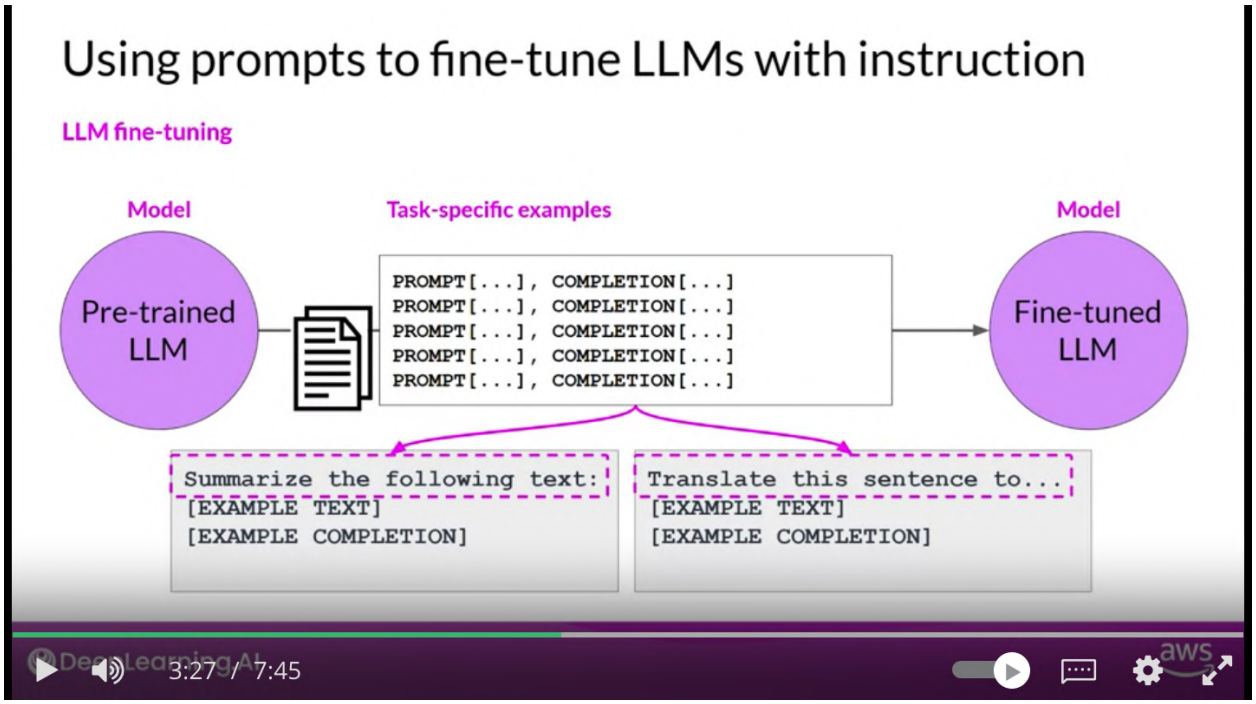


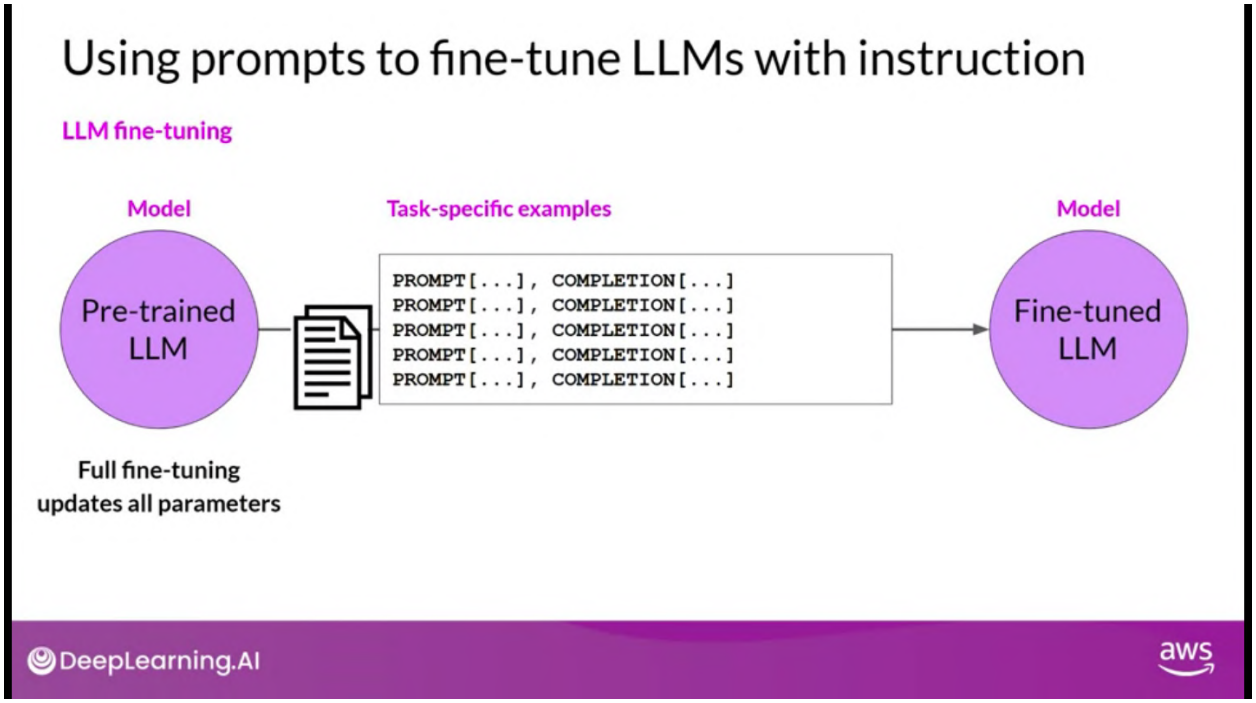
Instruction fine-tuning demonstrates how it should respond to specific instruction.
Dataset used for training includes many pairs of prompt completion examples for the task you are interested in.

5. Instruction Fine-Tuning:
   - Training the model with examples demonstrating how to respond to specific instructions.
   - Example prompts for the task of classifying reviews.



- Building a dataset of prompt completion examples for specific tasks (e.g., summarization, translation). For translation, have instructions as “”translate this sentence to: [example text] [example completion] “. This allows the model to generate responses that follow the instruction.

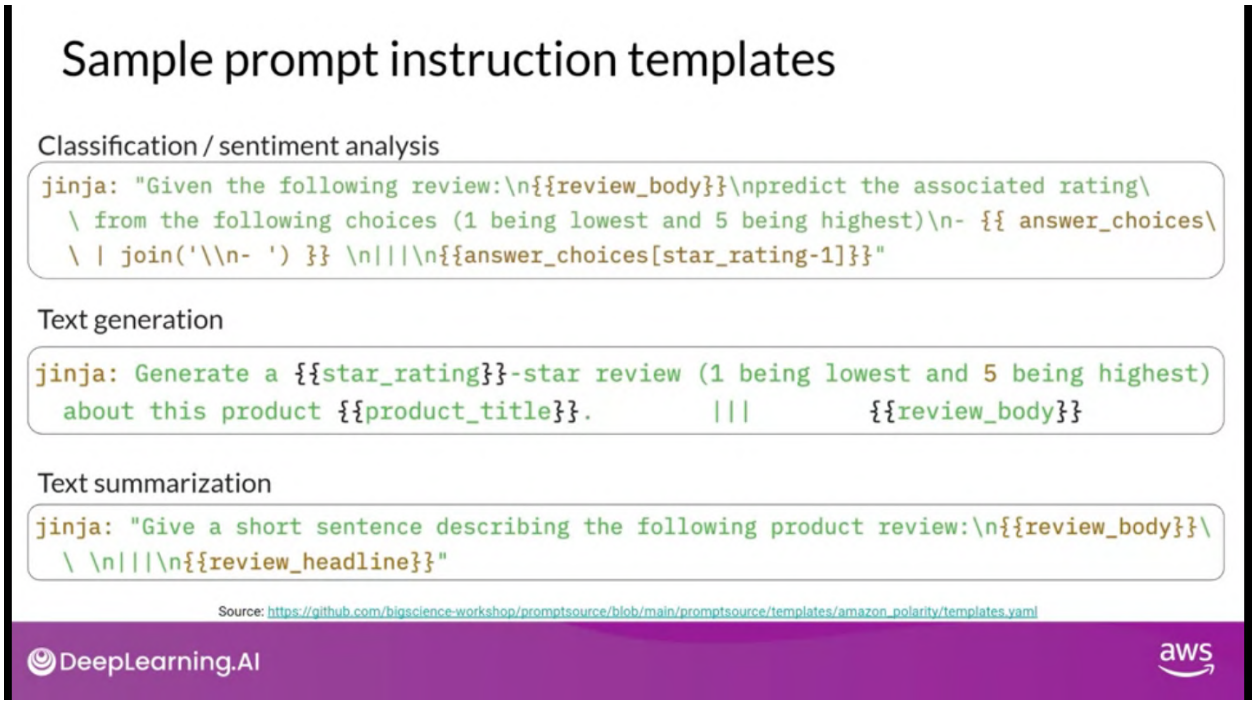




6. Full Fine-Tuning and Updated Weights:
   - Instruction fine-tuning involves updating all model weights - known as full fine tuning.
   - Resulting in a new version of the model with updated weights.

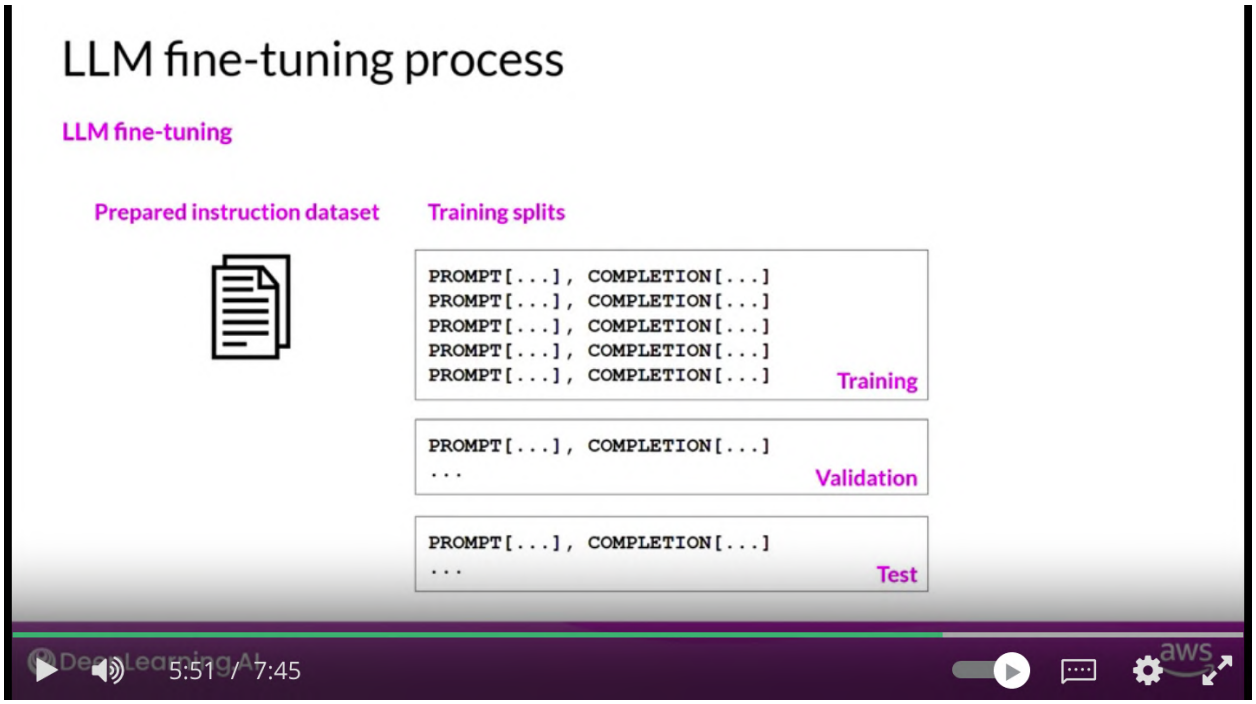
- Full Fine tuning requires enough budget to store the gradients, optimizer and other components updated during training. Benefit form memory optimization and parallel computing strategies like DDS, FSDP.

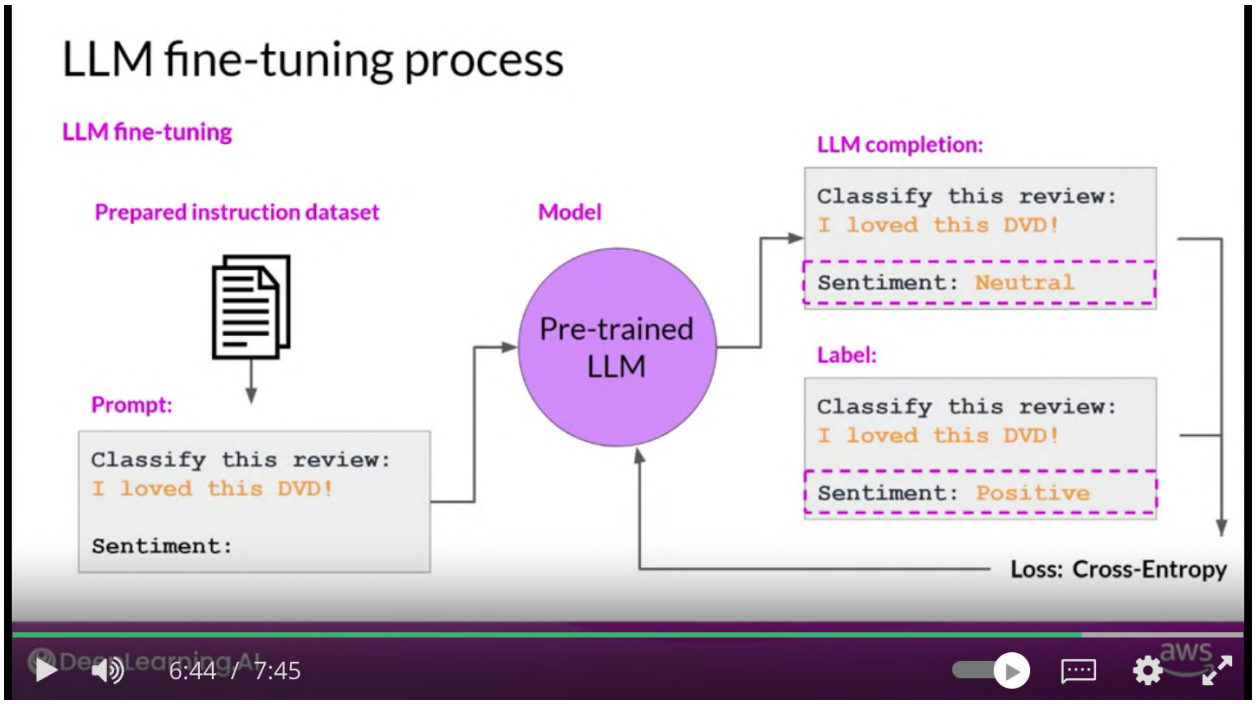
7. Preparing Training Data:
   - Availability of publicly available datasets for language model training.
   - Format conversion using prompt template libraries. Can be used existing dataset and turn them into instruction tuned dataset.



Review body - original text, instruction , then result

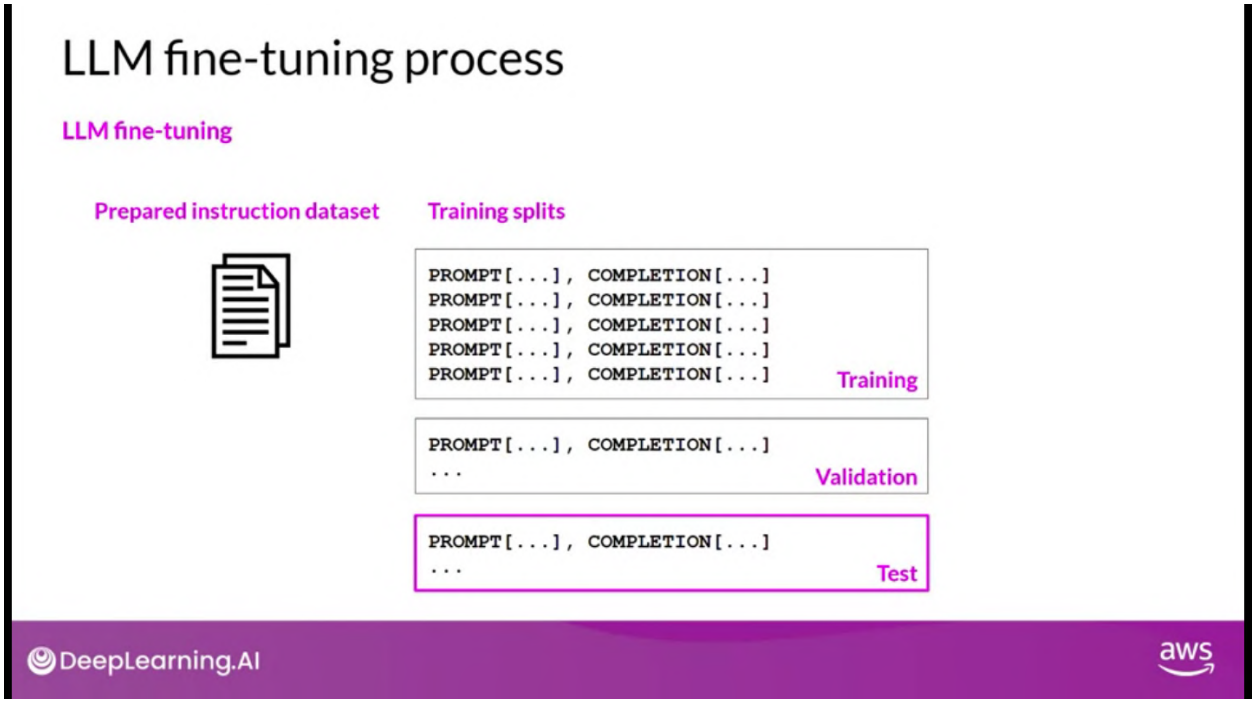
Once instruction tuned dataset is prepared divide it into -





8. Fine-Tuning Process Steps:
   - Division of dataset into training, validation, and test splits.
   - Selecting prompts from the training dataset for fine-tuning which generates completion.
   - Model completion generation for the selected prompts.

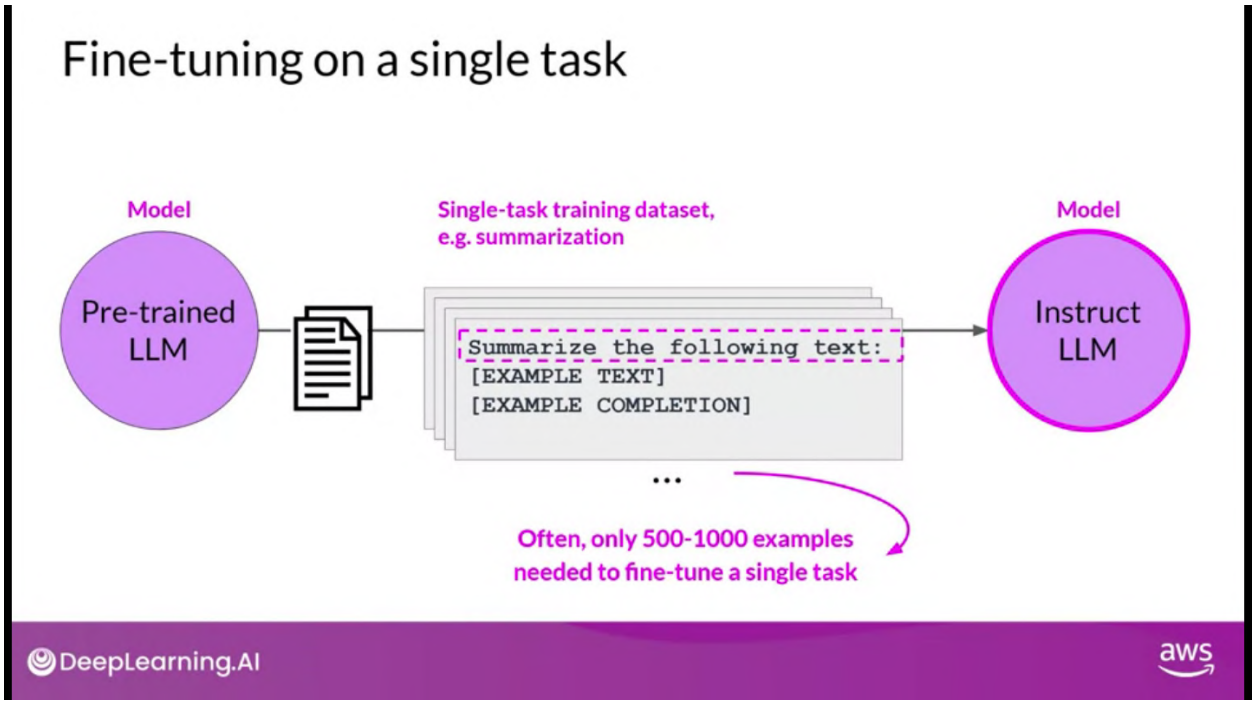
- Then compare the model completion response with training data.
- Output of LLM is the probability distribution of token, So can compare the prediction with label and using cross-entropy function to calculate loss between 2 token distributions. And then use calculated loss tpo update the model weights in in standard backpropagation. Do this for many batches of prompt completion pairs and over several epochs update the weights so that the model performance on the task improves.



9. Evaluation of Fine-Tuned LLM:
- Evaluation using holdout validation dataset to measure performance. This will give you validation accuracy
- Final performance evaluation using holdout test dataset. This will give you test accuracy
- Obtaining validation accuracy and test accuracy as metrics.
- Generation of a new version of the base model with improved task performance.

10. Fine-Tuning with Instruction Prompts:
- Common approach for fine-tuning LLMs.
- Instruction fine-tuning as the synonymous term for fine-tuning LLMs.
- Enhanced performance achieved through this method.

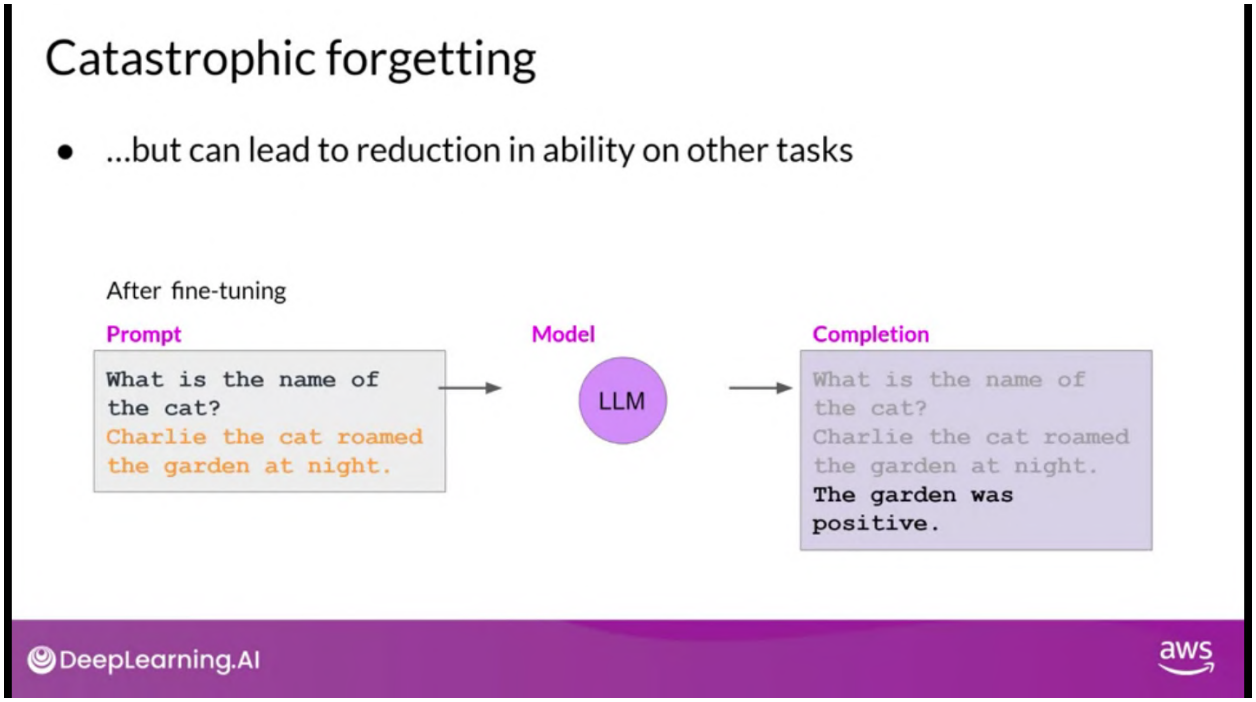
Fine tune on single task



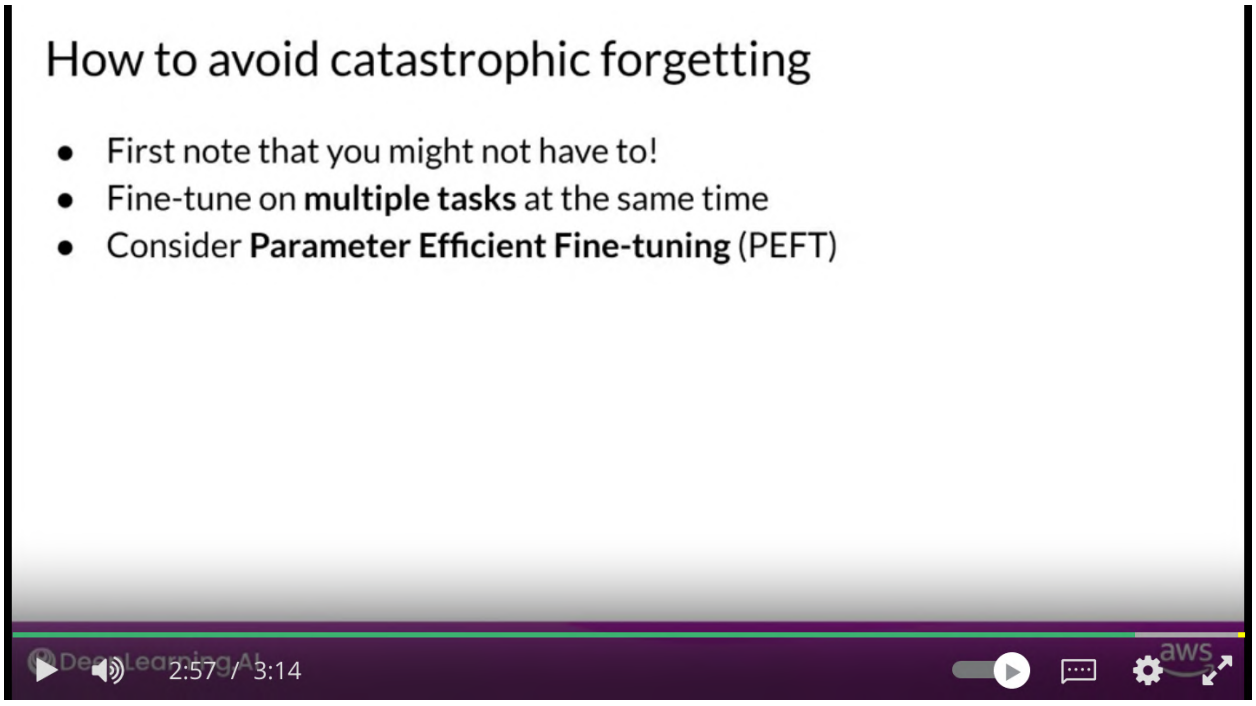
Fine LLM on single task. Only 5000-1000. But can lead to catastrophic forgetting

- Fine-tuning large language models (LLMs) can significantly enhance performance on a specific task even with a limited number of examples, typically ranging from 500 to 1000 examples.
- During fine-tuning, the weights of the original LLM are modified to better suit the requirements of the fine-tuning task, resulting in improved performance.
- However, fine-tuning can also lead to catastrophic forgetting, which refers to the degradation of performance on other tasks that the LLM was originally trained on.
- Catastrophic forgetting occurs because the fine-tuning process alters the weights of the LLM, causing it to excel at the fine-tuning task but lose proficiency in previously learned tasks.

Strategies to Avoid Catastrophic Forgetting:



Does catastrophic forgetting impact use case?



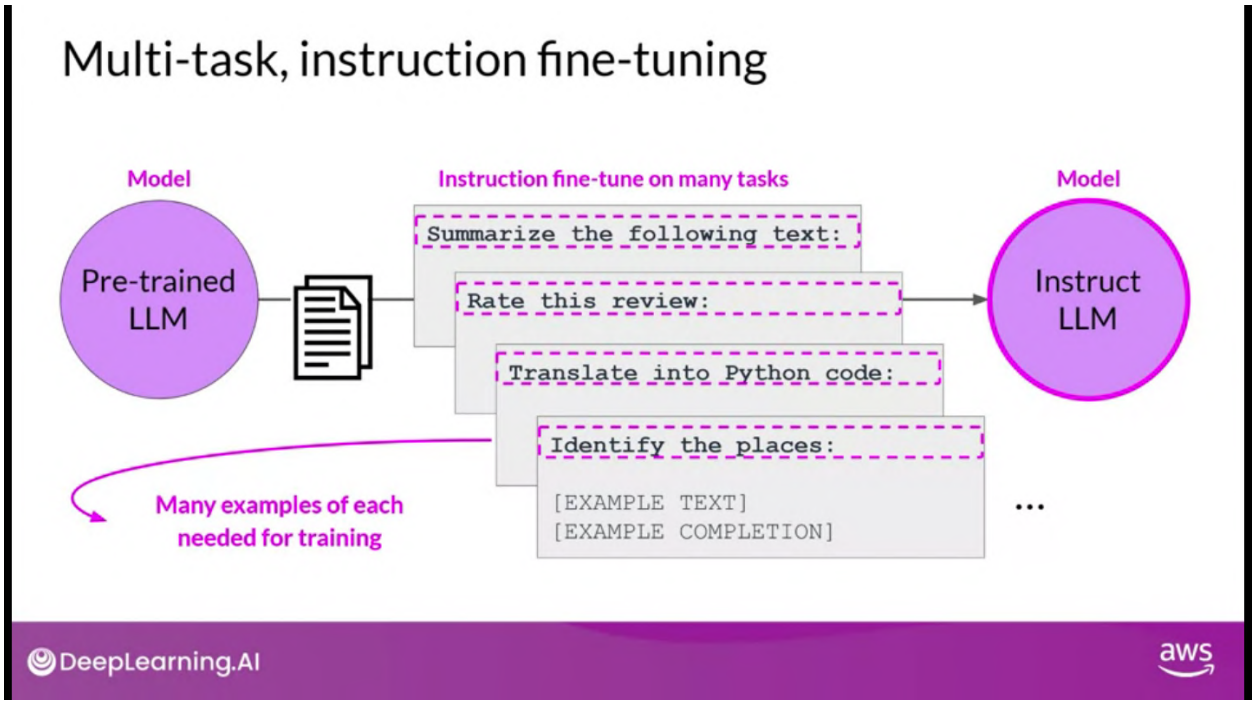
PEFT - robustness to catastrophic forgetting since most of the pre-trained weights are left unchanged.

- If the goal is to optimize performance on a single task without considering other tasks, catastrophic forgetting may not be a significant concern.

- Fine-tuning on multiple tasks simultaneously can help mitigate catastrophic forgetting, although it necessitates more data and computational resources. Good multitask finetuning may require 50,000 to 100,000 examples across many task. More data and compute.
- Parameter efficient fine-tuning (PEFT) is a technique that addresses catastrophic forgetting by incorporating small adapters while retaining the original LLM weights. This approach allows for task-specific modifications without interfering with the knowledge learned during pre-training.
- Employing multitask fine-tuning, where the LLM is trained on multiple related tasks, can also assist in avoiding catastrophic forgetting by maintaining competence across various tasks.

Key Points:
- Fine-tuning is beneficial for improving performance on a specific task.
- However, fine-tuning can result in the LLM forgetting how to perform previously learned tasks, known as catastrophic forgetting.
- Strategies such as multitask tuning, PEFT, or prioritizing the key task can be employed to prevent catastrophic forgetting and maintain proficiency across different tasks.

Multi-task instruction fine-tuning

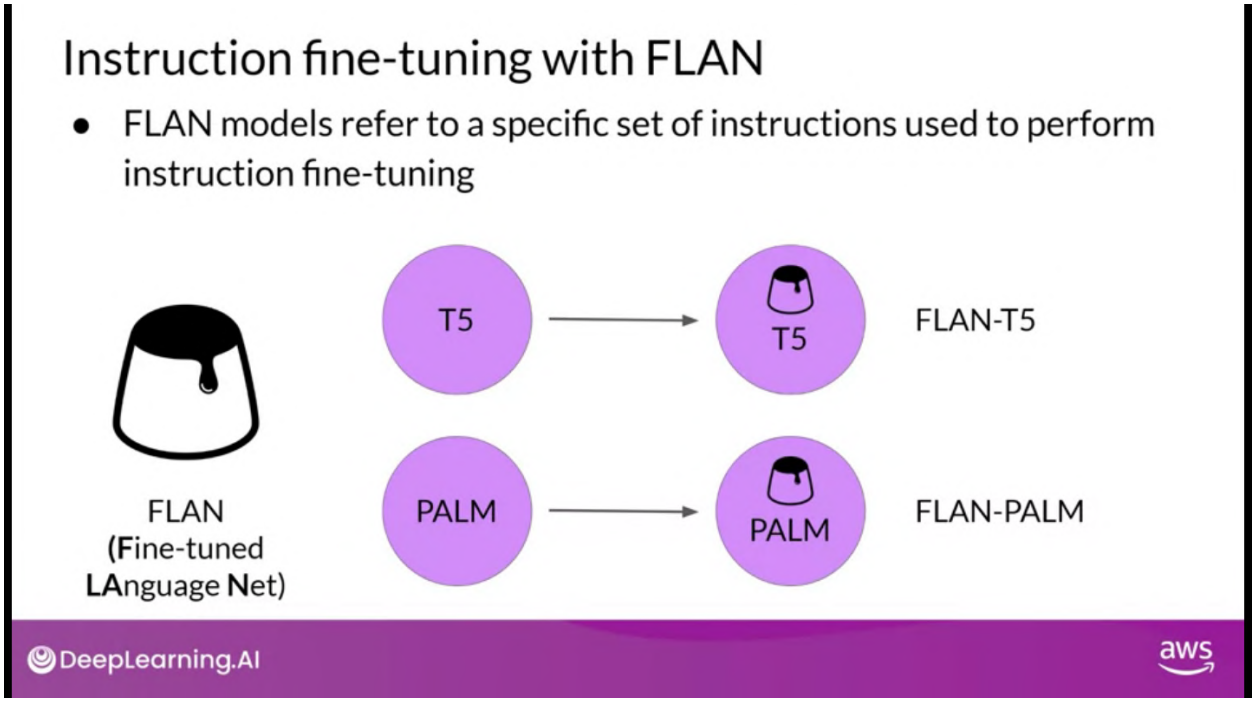


- Multitask fine-tuning is an extension of single task fine-tuning, where the training dataset includes inputs and outputs for multiple tasks.
- The dataset consists of examples that instruct the model to perform various tasks, such as summarization, review rating, code translation, and entity recognition.
- During training, the model is trained on this mixed dataset to improve its performance on all the tasks simultaneously, effectively avoiding catastrophic forgetting.
- Over multiple epochs of training, the model's weights are updated based on the calculated losses across examples, resulting in a model that is proficient in multiple tasks.

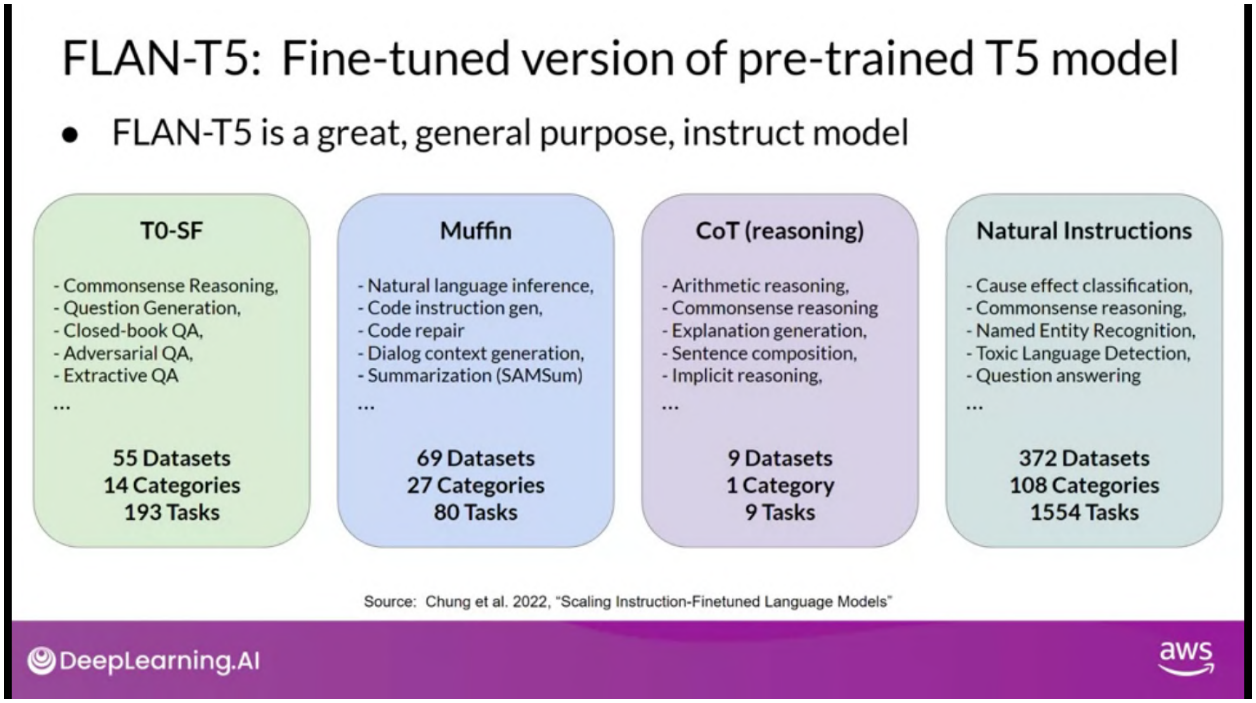
Drawbacks and Considerations:

- Multitask fine-tuning requires a significant amount of data, potentially ranging from 50,000 to 100,000 examples in the training set.
- Despite the data requirement, the effort put into assembling this data can be worthwhile as the resulting models are often highly capable and suitable for scenarios where performance across multiple tasks is desired.

FLAN Family of Models:



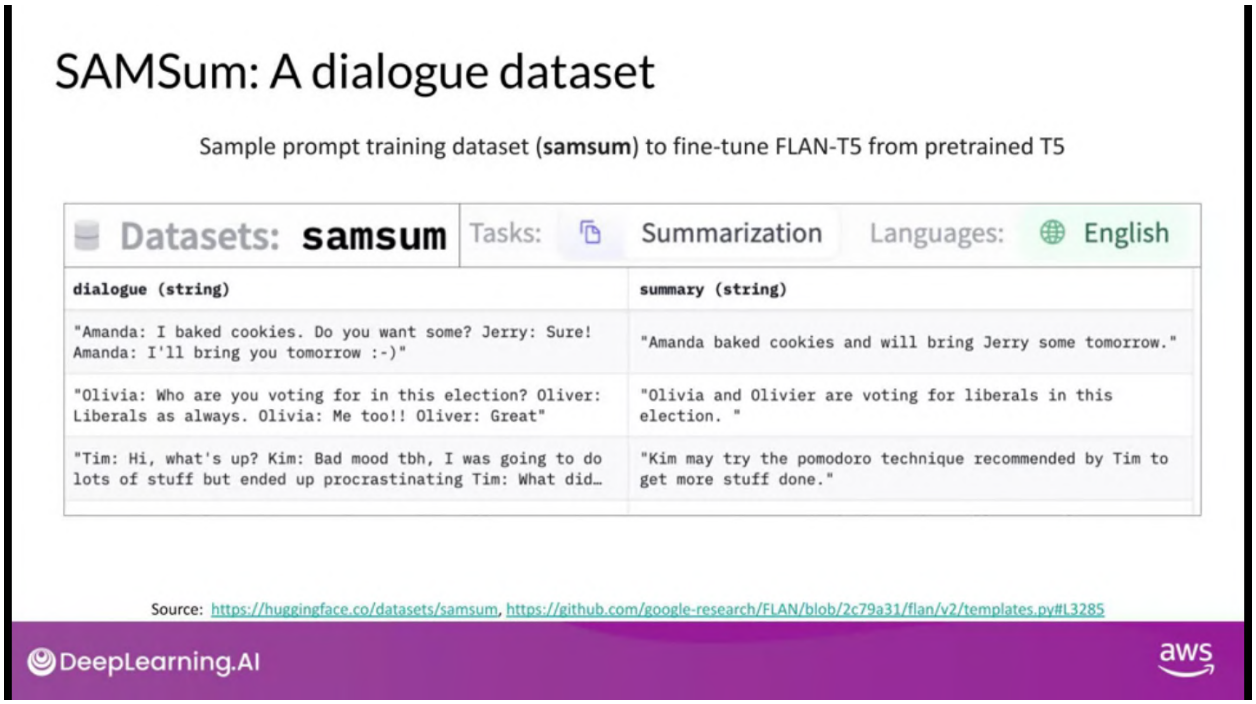
- The FLAN family of models is an example of models trained using multitask instruction fine-tuning.
- FLAN, which stands for fine-tuned language net, consists of different models fine-tuned using specific sets of instructions.
- FLAN-T5 is the FLAN instruction version of the T5 foundation model, while FLAN-PALM is the FLAN instruction version of the PALM foundation model.



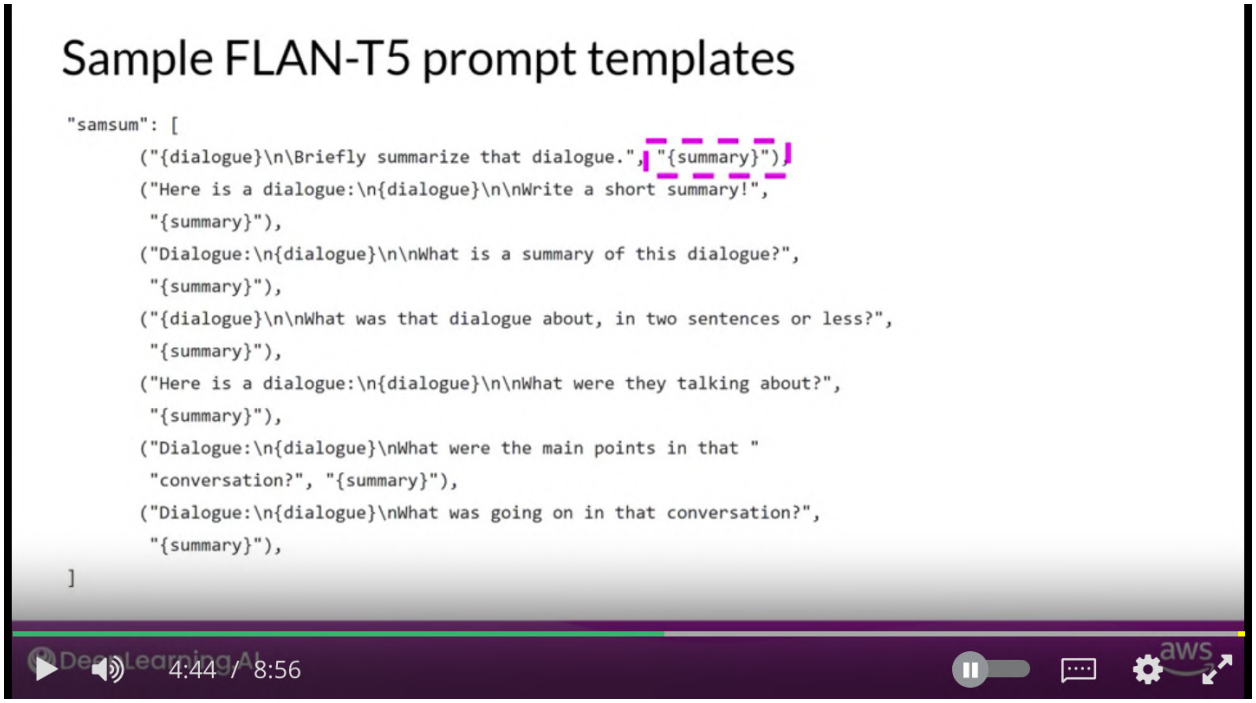
- FLAN-T5, in particular, has been fine-tuned on 473 datasets across 146 task categories, making it a versatile and general-purpose instruction model.

Prompt Dataset and Template:

- The SAMSum dataset is an example of a prompt dataset used for summarization tasks in FLAN-T5.



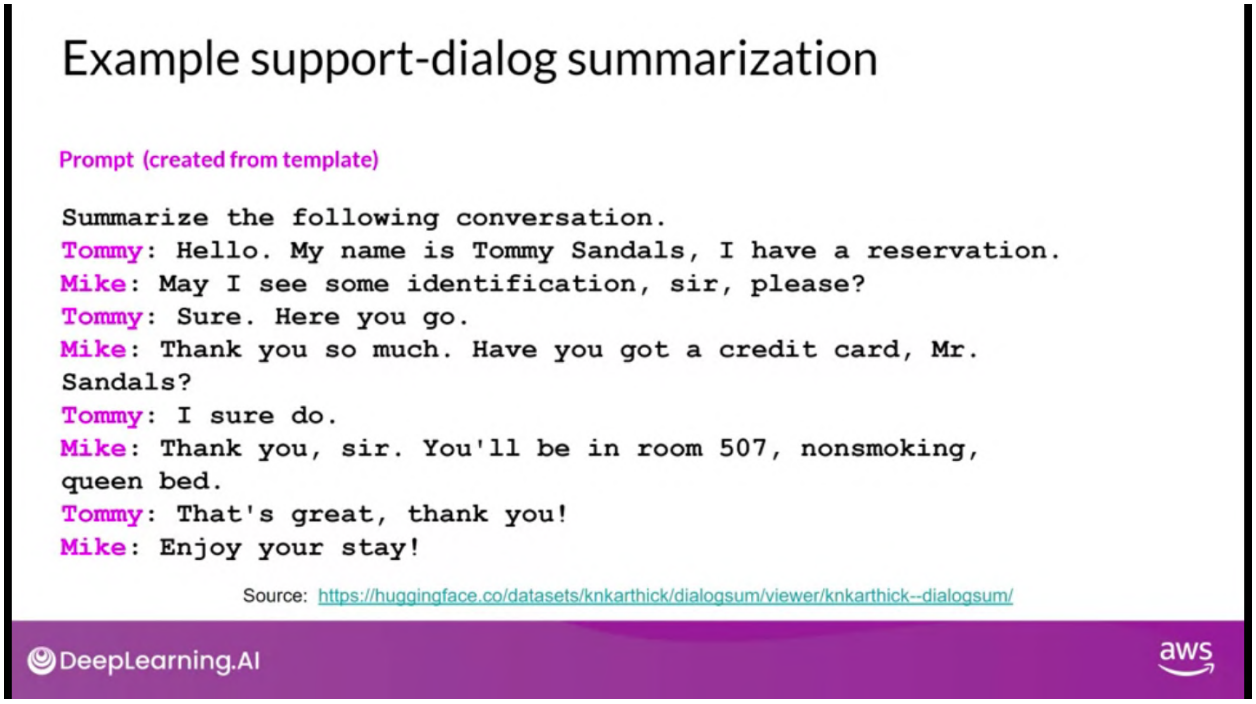
- SAMSum is a collection of 16,000 messenger-like conversations with corresponding summaries, created by linguists to generate a high-quality training dataset for language models.



- **Prompt templates**, such as the one designed for SAMSum, include different variations of instructions that ask the model to summarize a dialogue. **Use paraphrasing in the template used for generating instructions. Including different ways of saying the same instructions helps the model to generalize and perform better.**
- These templates help the model generalize and perform better by providing diverse ways of expressing the same instruction.

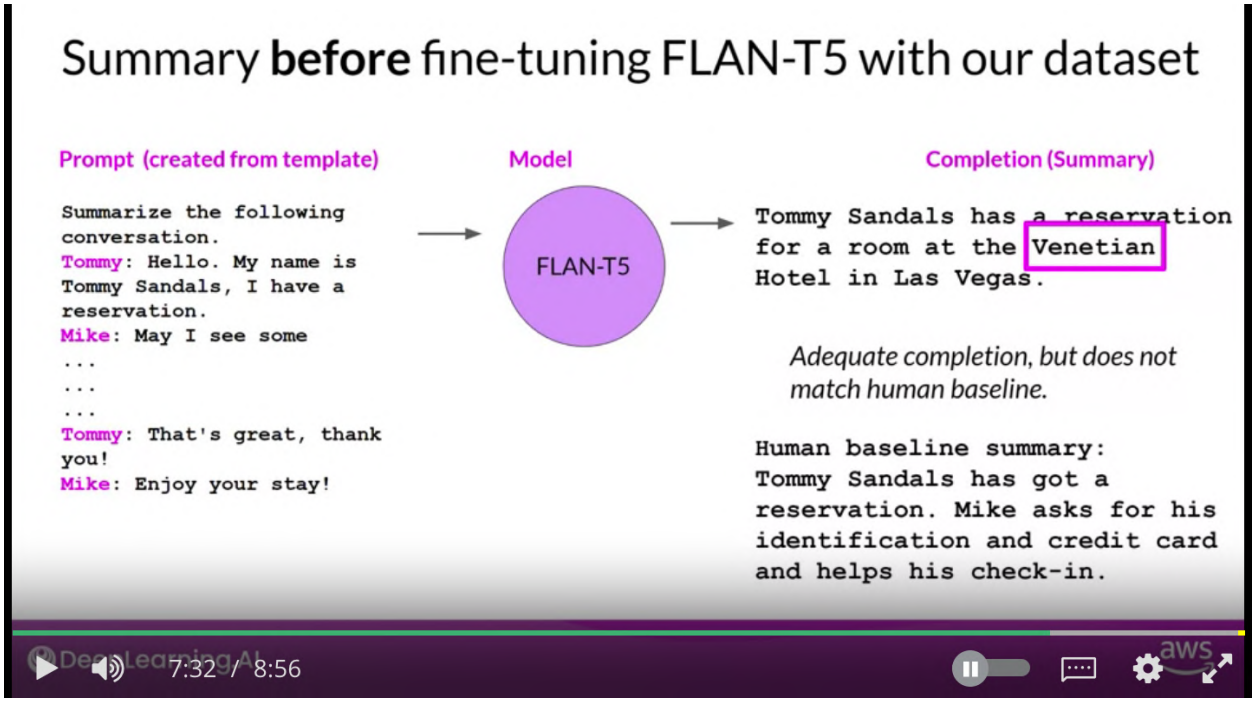
Custom Fine-Tuning and Domain-Specific Datasets:

- While FLAN-T5 is a capable model, additional fine-tuning may be necessary for specific use cases.

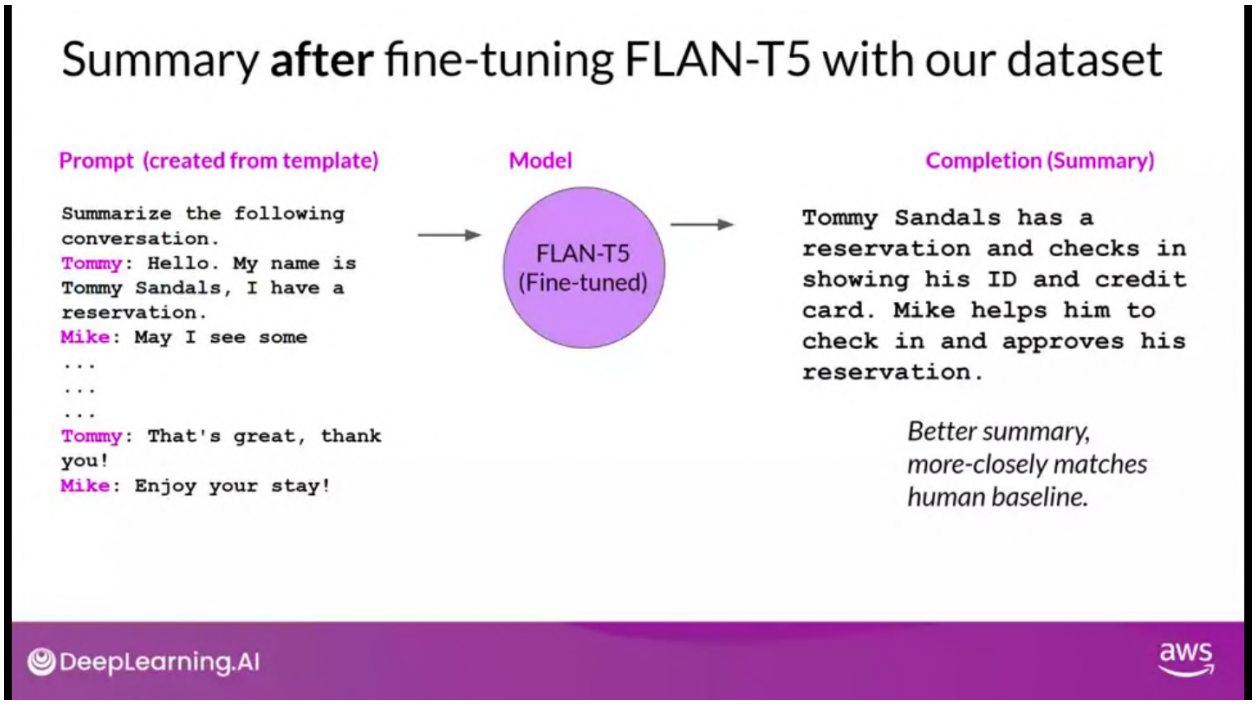


- Domain-specific datasets, such as dialogsum, can be used to further improve FLAN-T5's ability to summarize conversations in a particular domain, such as support chat conversations.
- Dialogsum consists of over 13,000 support chat dialogues and summaries, offering a more relevant training dataset for fine-tuning. **Additional domain specific data to improve model's ability to support task.**

Before fine tuning - adding extra not relevant information



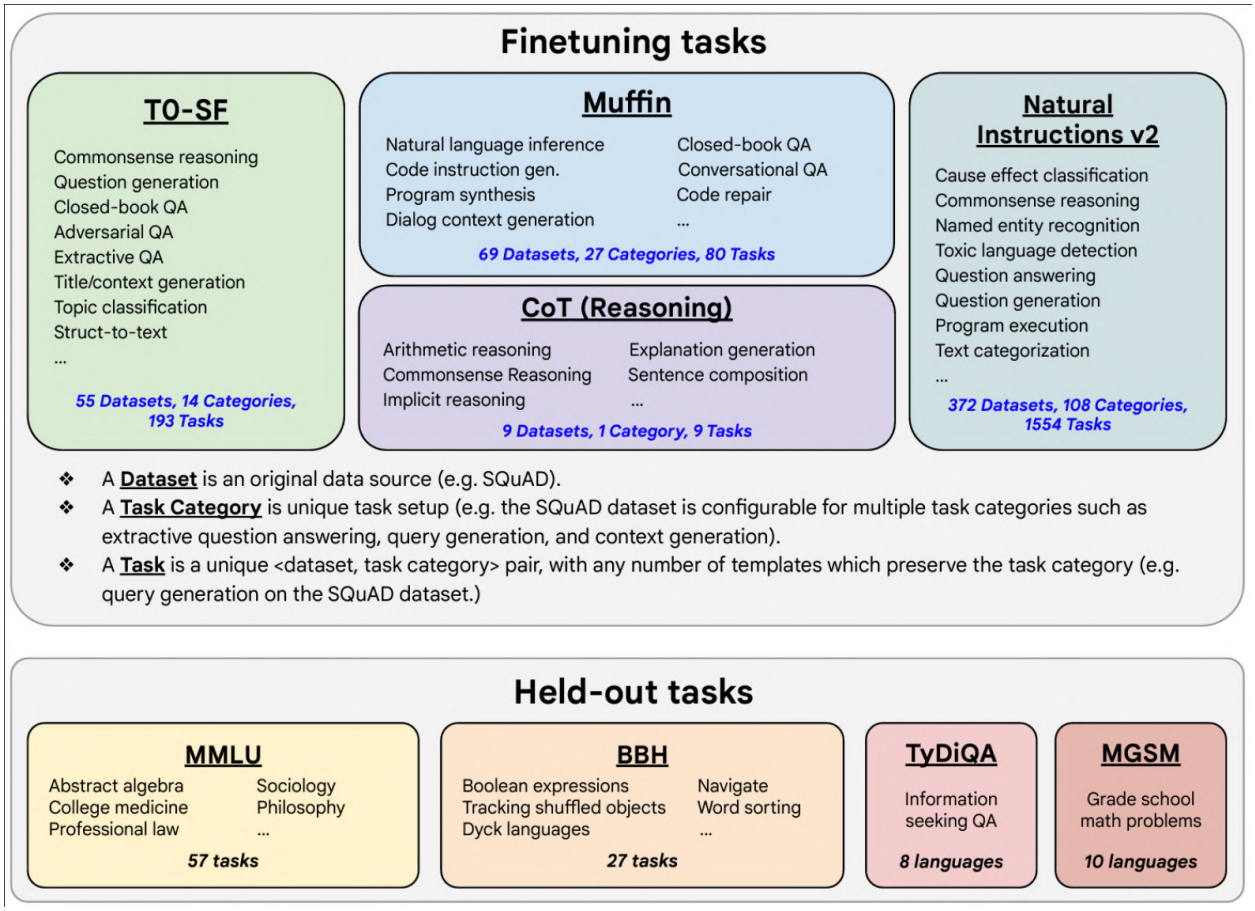
After fine tuning, no fabricated information. Closer to human.



- Fine-tuning with custom data, such as the support chat conversations from a company's customer support application, helps the model learn the specific nuances and requirements of the organization.

Fine tune on company’s own internal data

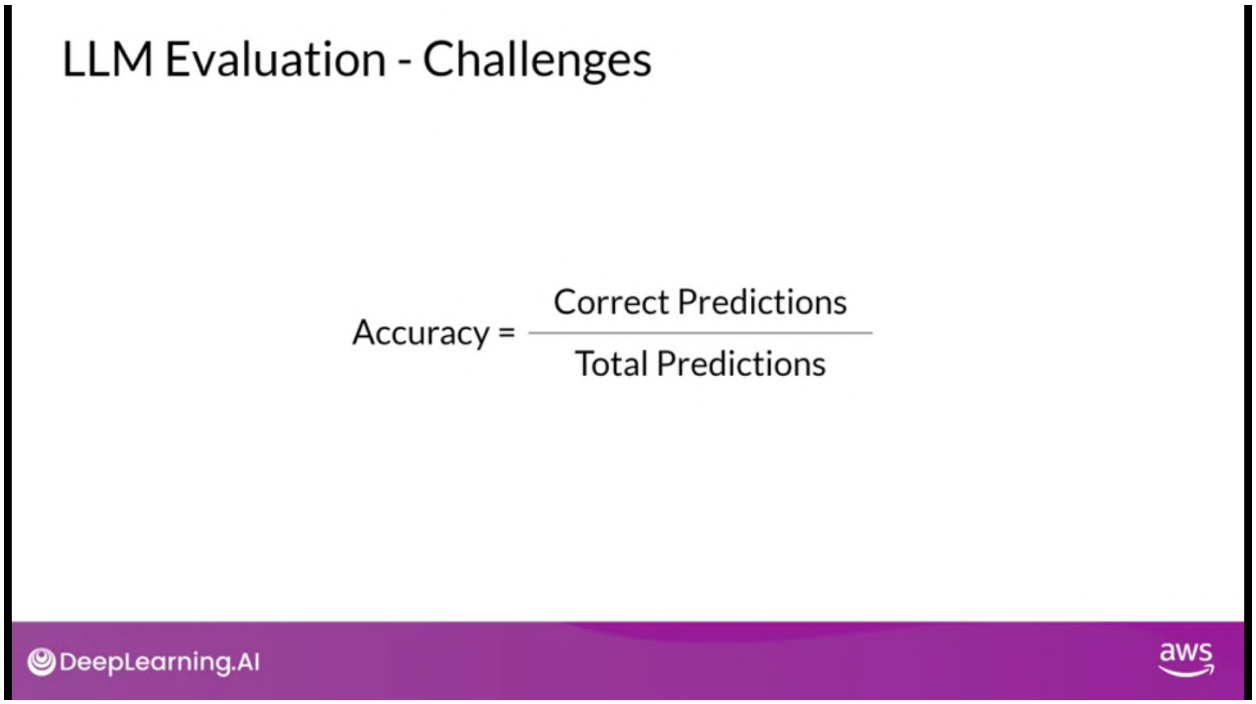
FLAN (Fine-tuned LAnguage Net), an instruction finetuning method, and presents the results of its application. The study demonstrates that by fine-tuning the 540B PaLM model on 1836 tasks while incorporating Chain-of-Thought Reasoning data, FLAN achieves improvements in generalization, human usability, and zero-shot reasoning over the base model. The paper also provides detailed information on how each these aspects was evaluated.



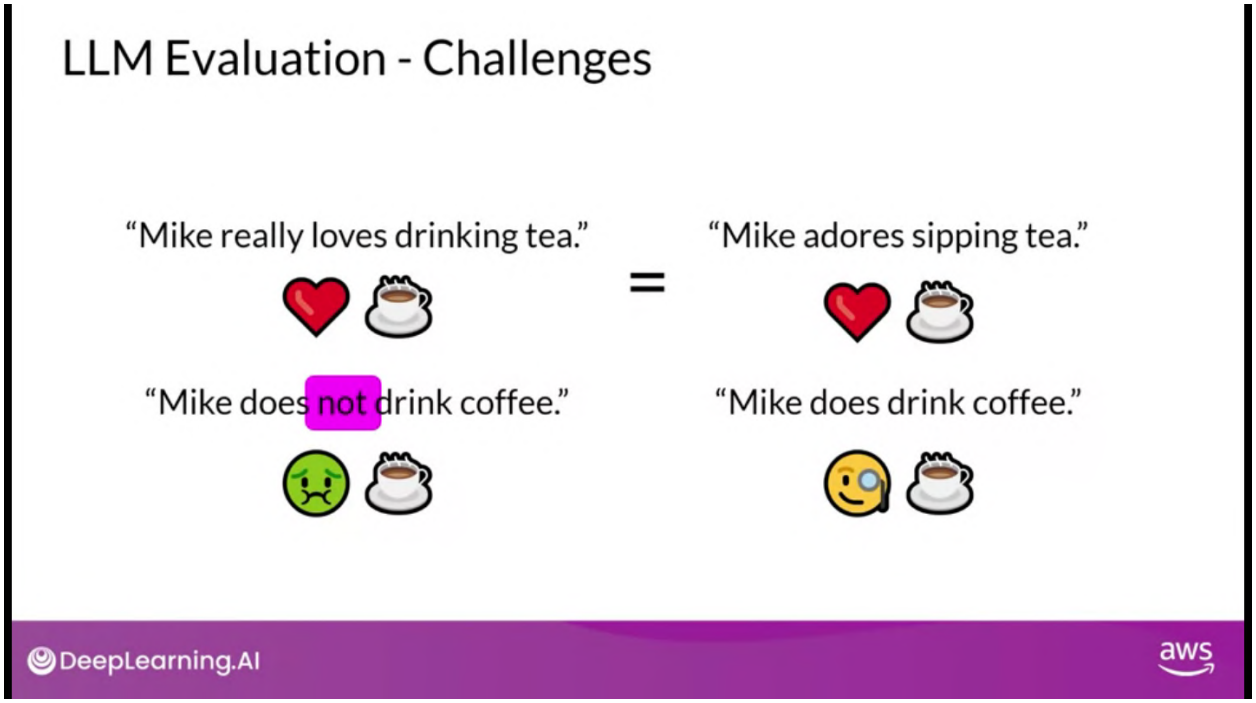
Evaluation and Metrics:

- Evaluating the quality of model completions is an essential aspect of fine-tuning.
- Various metrics and benchmarks can be used to assess the performance of the model and compare the fine-tuned version with the original base model.
- These metrics help measure how well the model is performing and quantify the improvements achieved through fine-tuning.

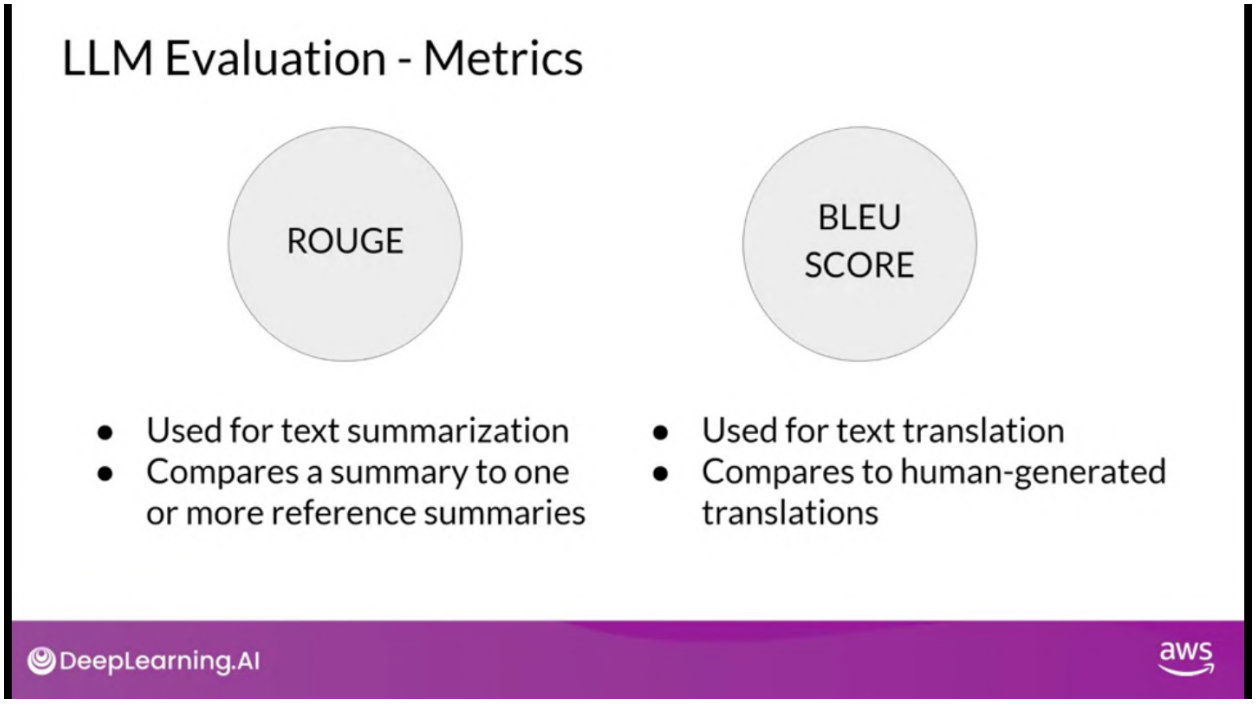
Model Evaluation



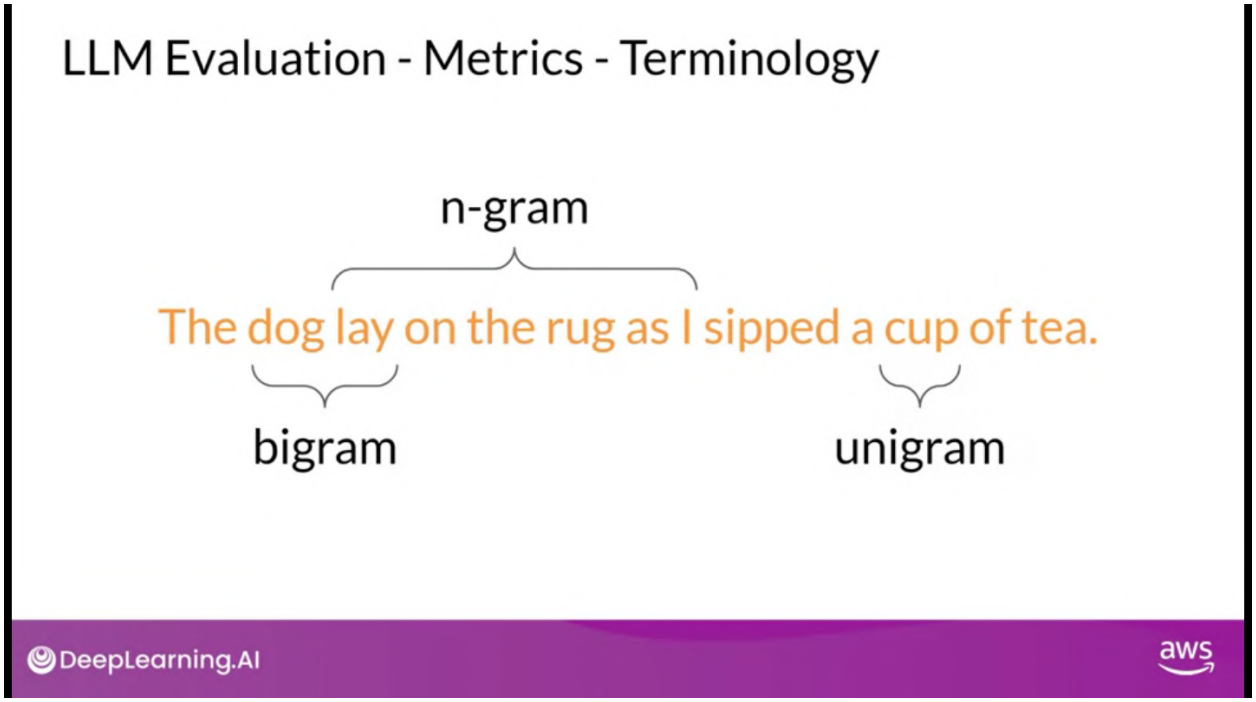
LLM output is non deterministic

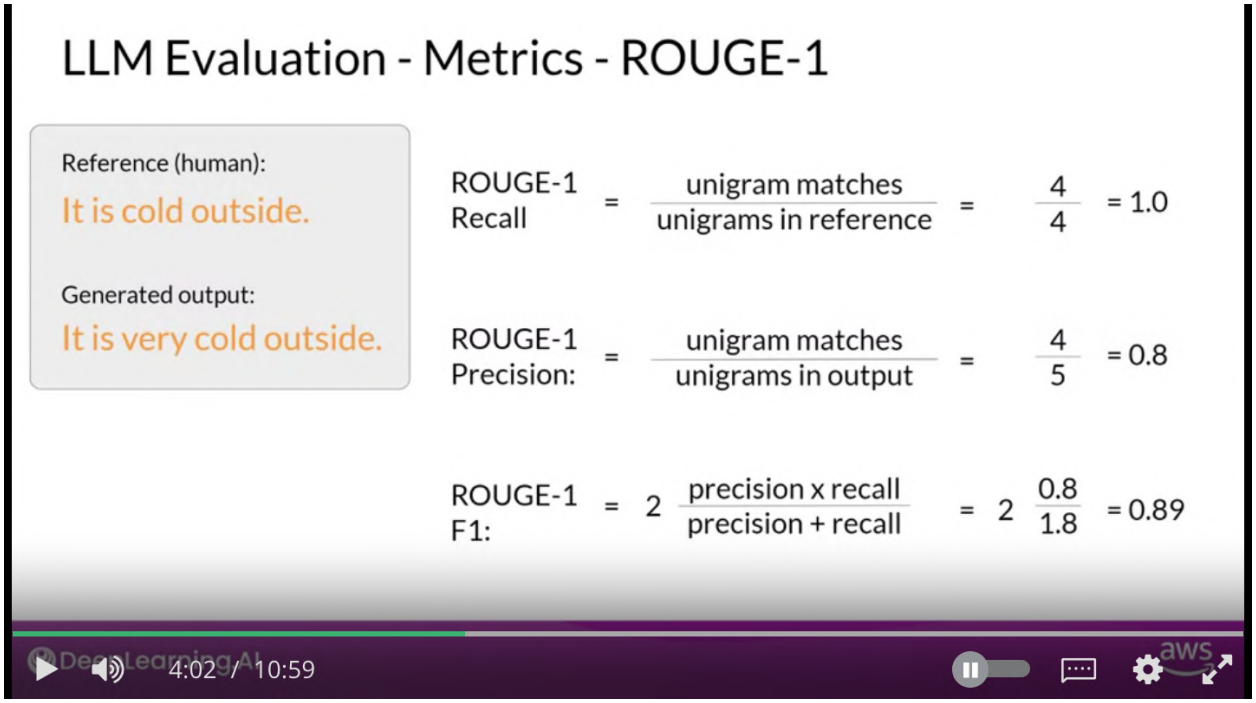


- In evaluating the performance of language models, statements such as "the model demonstrated good performance on this task" or "the fine-tuned model showed a large improvement in performance over the base model" are commonly used.
- To formalize the improvement in performance of a fine-tuned model over the pre-trained model, several metrics can be utilized.

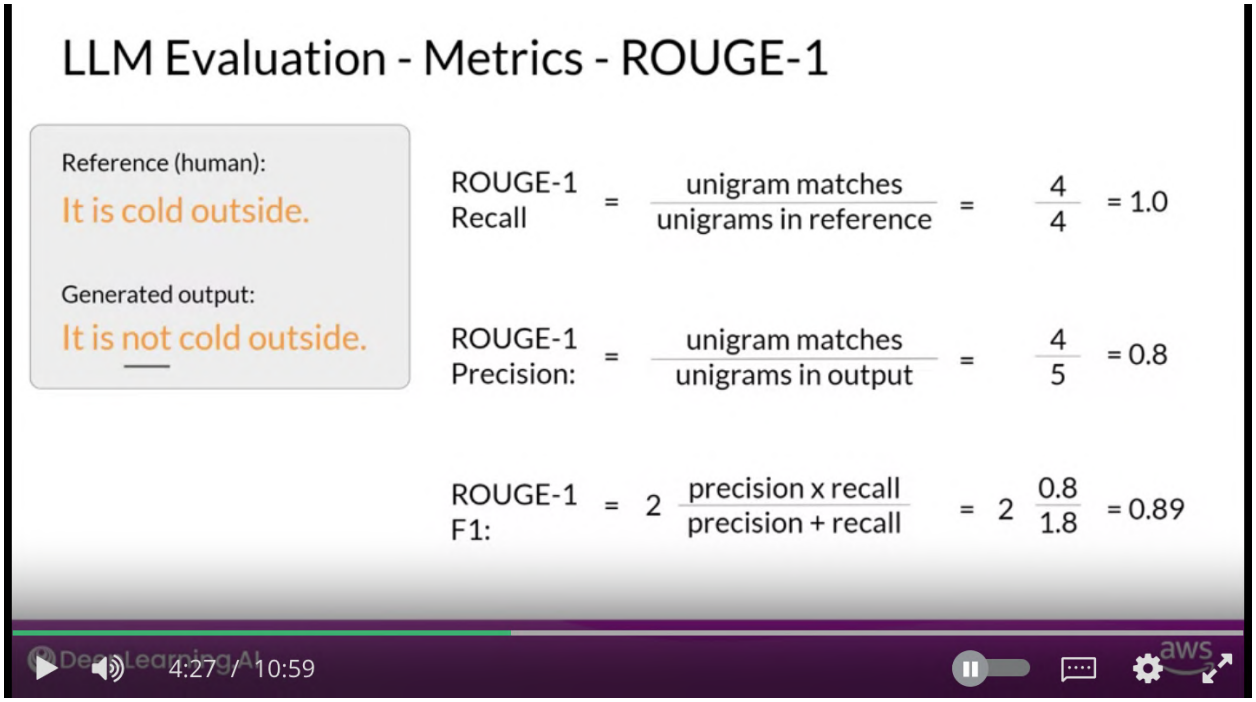


- ROUGE (Recall-Oriented Understudy for Gisting Evaluation) and BLEU (Bilingual Evaluation Understudy) are two widely used evaluation metrics for different tasks in large language models.
- In traditional machine learning, performance assessment is done by comparing the model's predictions on known training and validation datasets. Metrics like accuracy can be calculated easily.
- However, in language models, where output is non-deterministic and language-based evaluation is more challenging, automated and structured measurement methods are required.

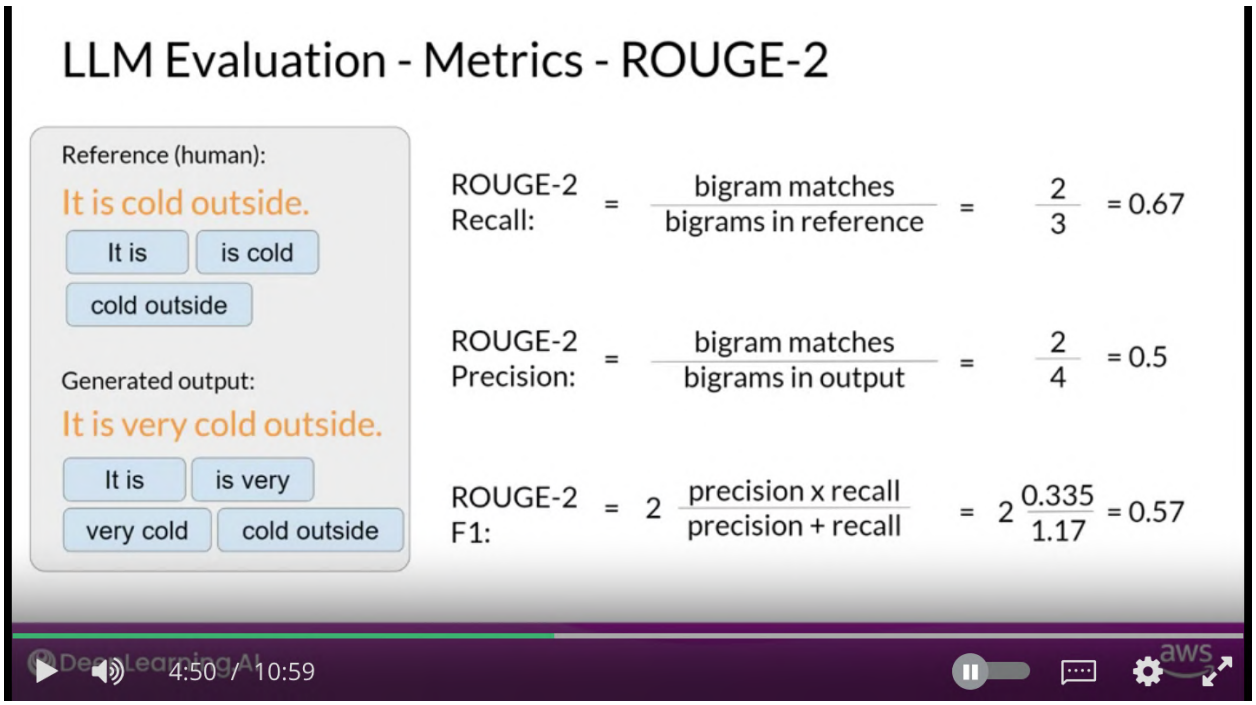




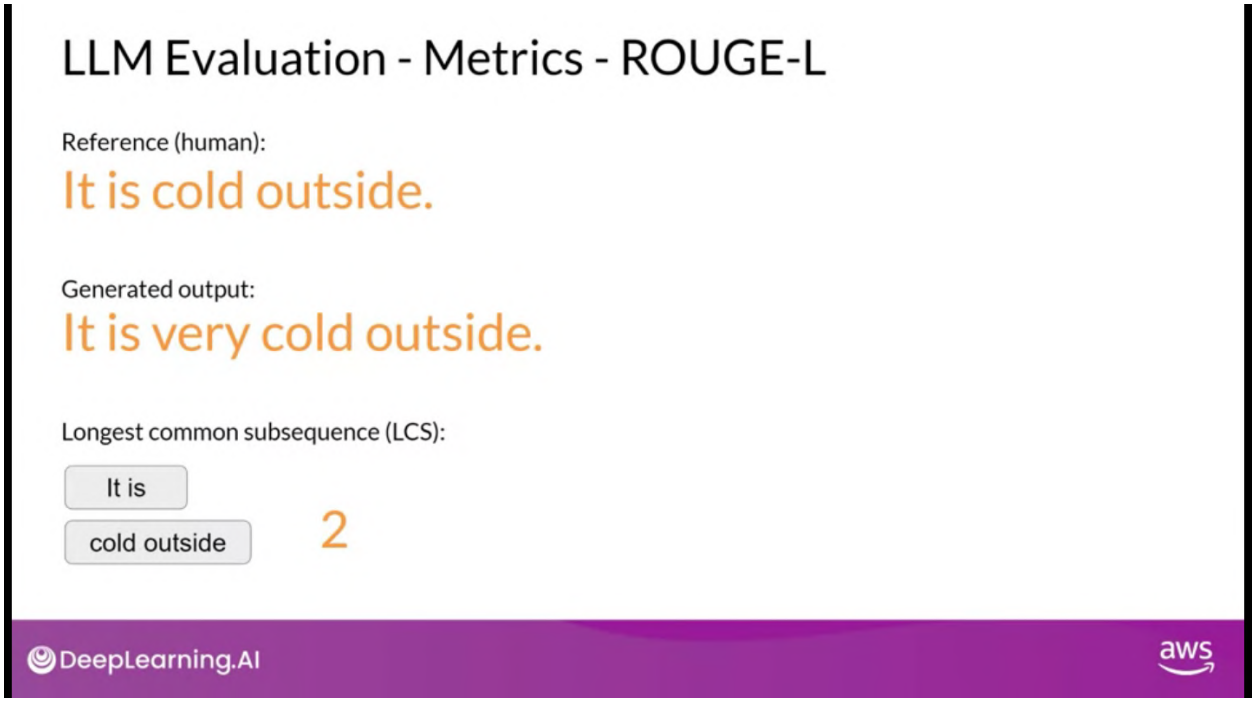
If instead of ‘very’ predict ‘not’ then the scores will be same even though meaning would be different.

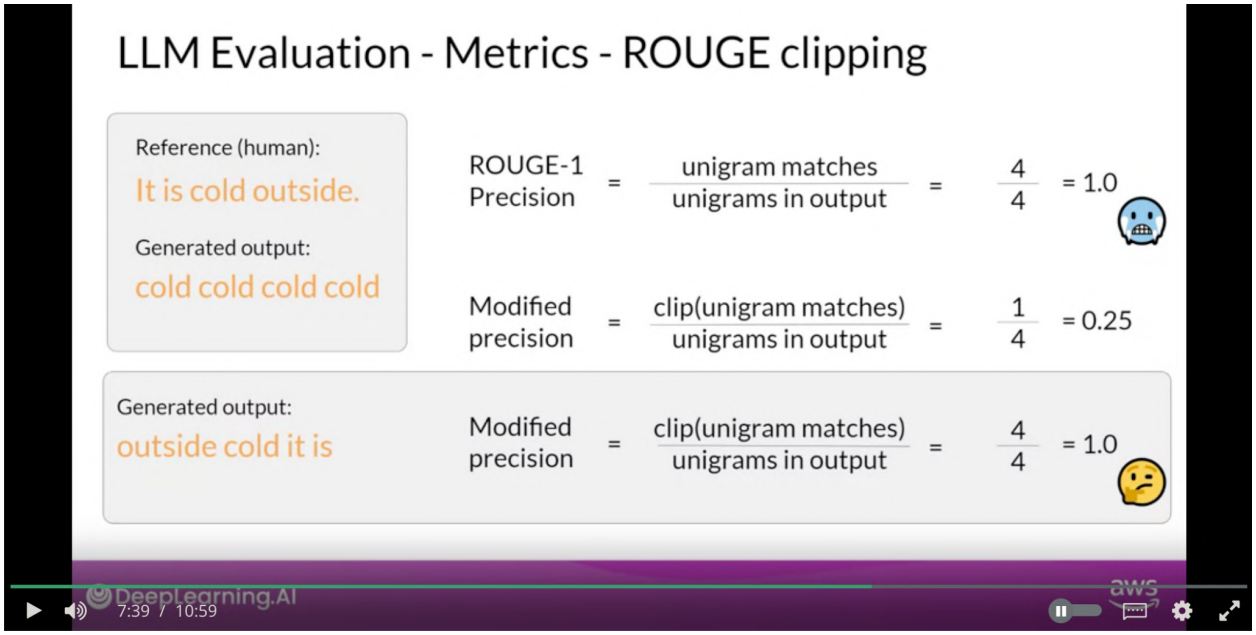


Slightly better score by bi-gram. Acknowledging the order of the word. Scores lower than rouge 1. With longer sentences the greater chance that bigrams don't match and maybe lower.



Look at the longest common subsequence in the reference and generated output
Rouge scores for different task are not comparable to one another.

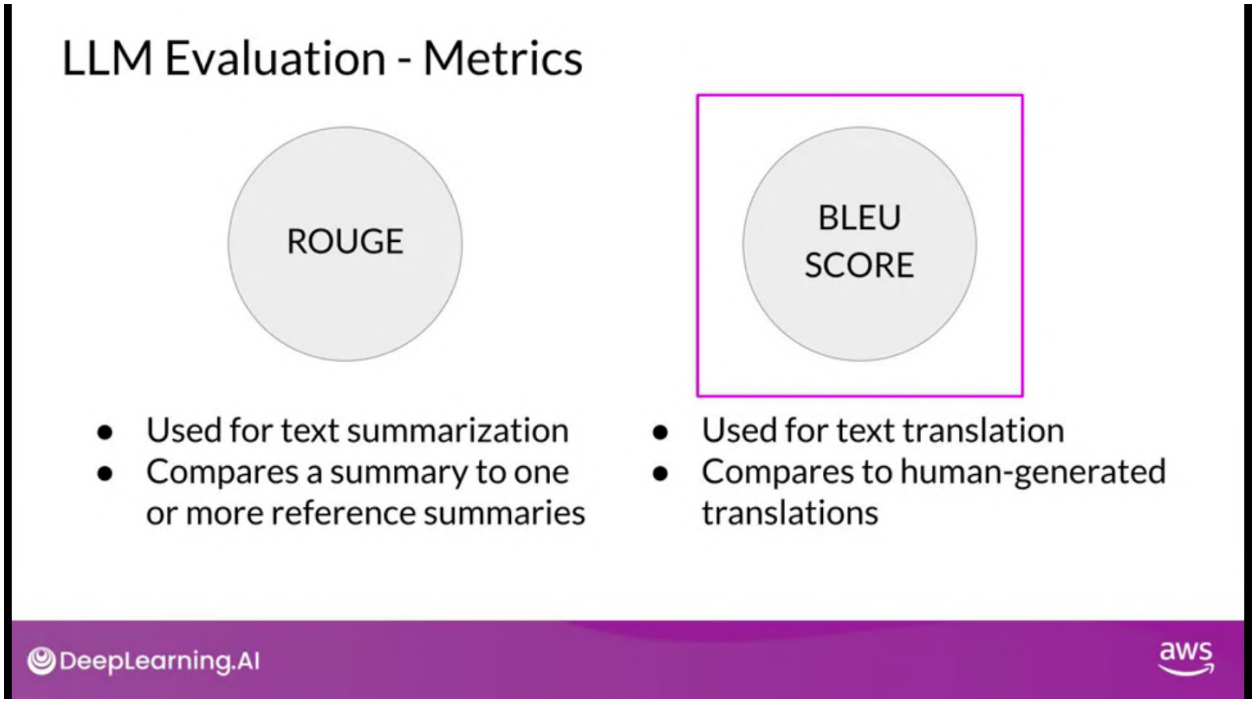




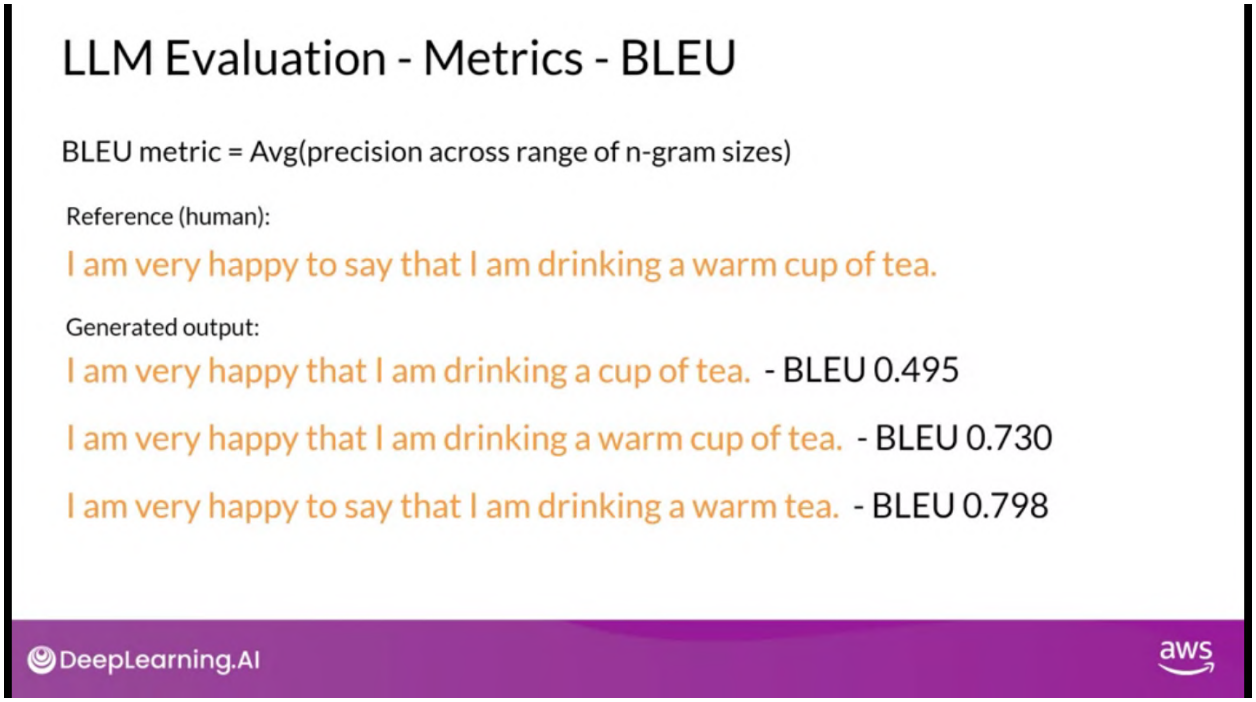
Use clipping function to limit of unigram matches to the maximum count for the unigram within the reference. Clip results in reduced score.
Still be challenged same word by different order

Experimenting most useful n-gram size will be dependant on the sentence and usecase

- Evaluating the similarity between sentences becomes complex. For example, determining the similarity between "Mike really loves drinking tea" and "Mike adores sipping tea" requires a structured approach.
- ROUGE-1 is a metric that focuses on unigram (single word) matches between the generated output and the reference sentence. Recall, precision, and F1 scores can be calculated for ROUGE-1.
- ROUGE-2 considers bigrams (pairs of words) and their matches between the generated output and the reference sentence.
- ROUGE-L measures the longest common subsequence between the generated output and the reference sentence, providing a more comprehensive evaluation of sentence similarity.
- BLEU score evaluates the quality of machine-translated text by comparing n-gram matches between the machine-generated translation and the reference translation.
- BLEU score calculates average precision across a range of different n-gram sizes, providing a holistic assessment of translation quality.
- ROUGE and BLEU scores are simple metrics that can be calculated relatively easily, and many language model libraries provide implementations for these metrics.
- While ROUGE and BLEU scores are useful for diagnostic evaluation of summarization and translation tasks, respectively, they should not be the sole basis for final evaluation.
- Evaluation benchmarks developed by researchers should be considered for an overall assessment of a language model's performance.



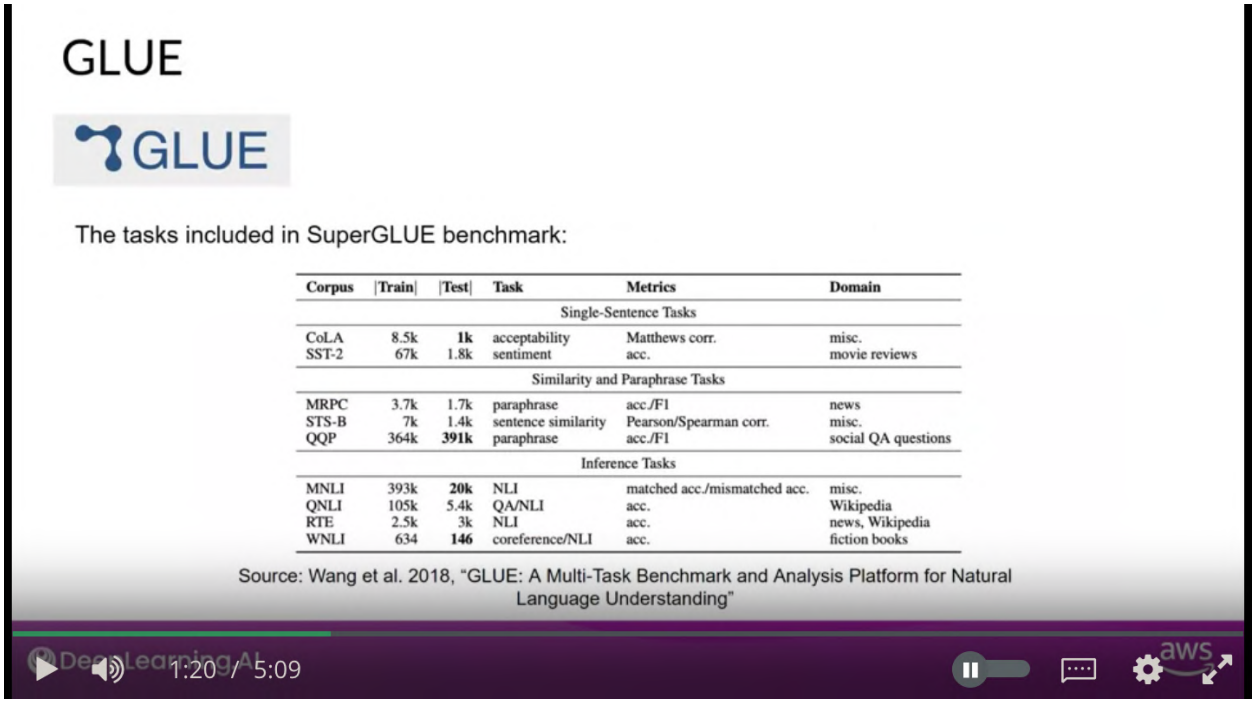
BLUE score machine translation. Calculates average precision over multiple n-gram sizes and then averaged. Quantifies the quality of translation by checking how many n-grams in the machine generated translated translation with ground label



Should use BLUE or ROUGE alone. Use ROUGE for diagnostic evaluation of summarization task and BLUE for translation task.

Benchmarks

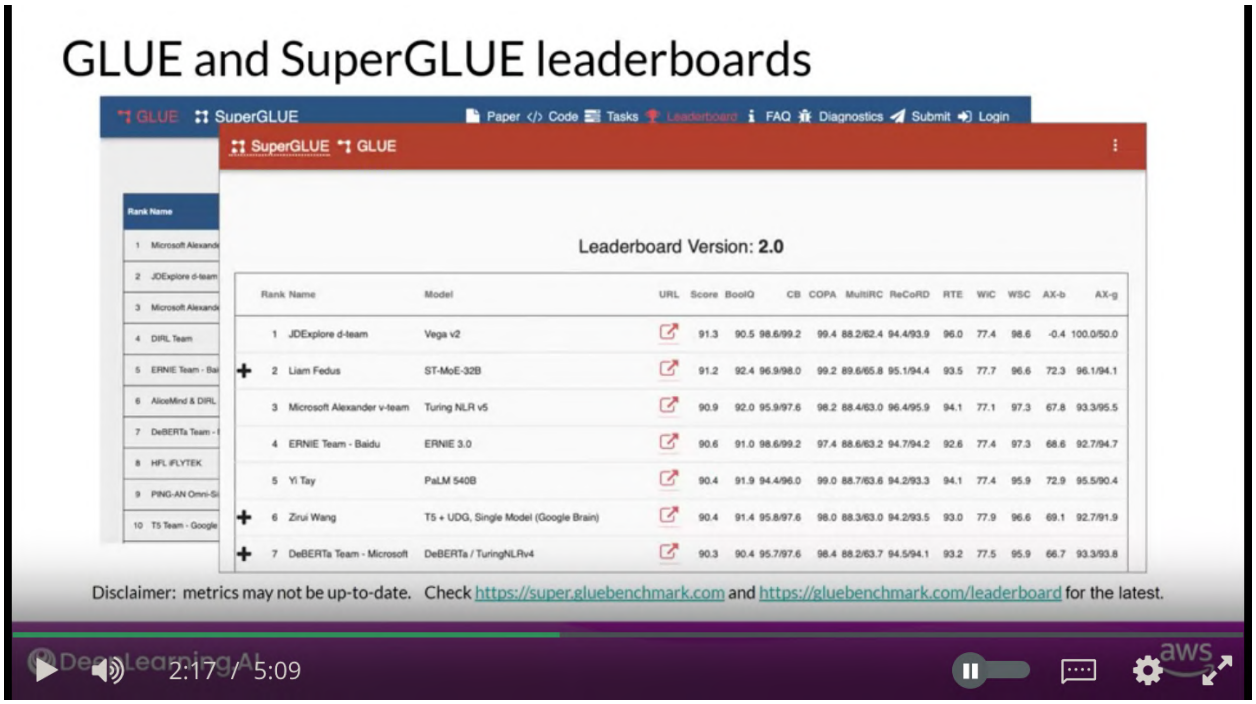
Selecting right evaluation dataset that isolate model specific skills and those that focus on potential risks - such as disinformation or copyright infringement



Corpus	|Train|	|Test|	Task	Metrics	Domain
			Single-Sentence Tasks		
CoLA	8.5k	**1k**	acceptability	Matthews corr.	misc.
SST-2	67k	1.8k	sentiment	acc.	movie reviews
			Similarity and Paraphrase Tasks		
MRPC	3.7k	1.7k	paraphrase	acc./F1	news
STS-B	7k	1.4k	sentence similarity	Pearson/Spearman corr.	misc.
QQP	364k	**391k**	paraphrase	acc./F1	social QA questions
			Inference Tasks		
MNLI	393k	**20k**	NLI	matched acc./mismatched acc.	misc.
QNLI	105k	5.4k	QA/NLI	acc.	Wikipedia
RTE	2.5k	3k	NLI	acc.	news, Wikipedia
WNLI	634	**146**	coreference/NLI	acc.	fiction books

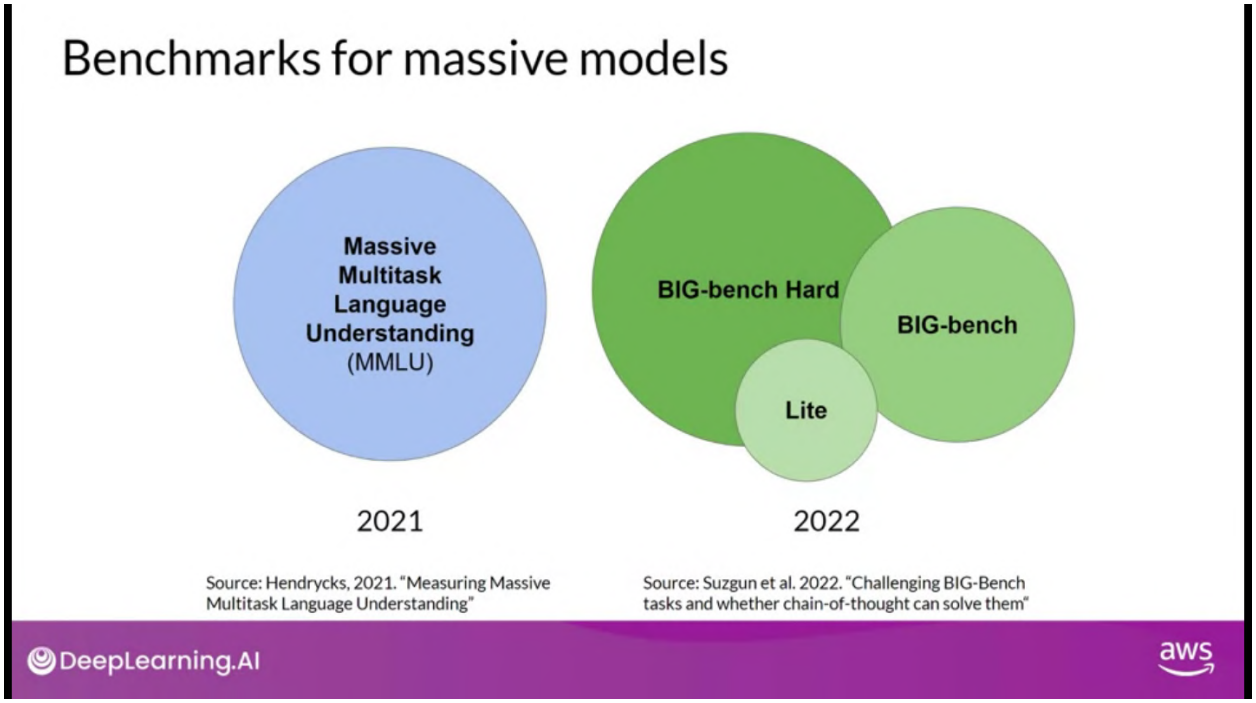
GLUE 2018

SuperGLUE -

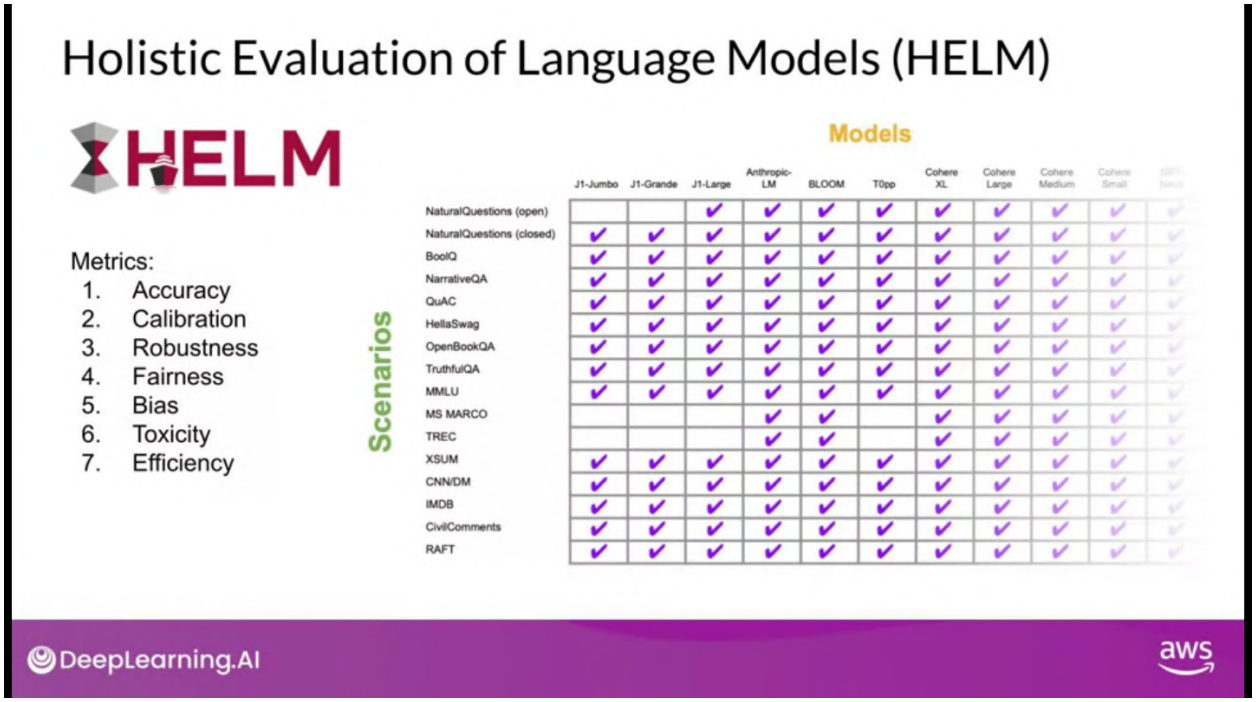


- LLMs (Language Models) are complex, and simple evaluation metrics like rouge and blur scores have limitations in assessing their capabilities.
- Pre-existing datasets and benchmarks established by LLM researchers are used to measure and compare LLMs more holistically.
- Selecting the right evaluation dataset is crucial for accurately assessing an LLM's performance and understanding its true capabilities.
- Isolating specific model skills, such as reasoning or common sense knowledge, and focusing on potential risks like disinformation or copyright infringement, can help in dataset selection.
- Evaluating an LLM's performance on unseen data provides a more accurate understanding of its capabilities.
- Benchmarks like GLUE, SuperGLUE, and HELM cover a wide range of tasks and scenarios designed to test specific aspects of an LLM.
- GLUE (General Language Understanding Evaluation) is a collection of natural language tasks (such as sentiment analysis and question answering) aimed at encouraging models that generalize across multiple tasks.
- SuperGLUE is a successor to GLUE and includes additional tasks that address limitations in the previous benchmark. New taks some of them
- GLUE and SuperGLUE have leaderboards and results pages for comparing and tracking model performance.

- LLMs are reaching human-level performance on specific tasks but are not yet comparable to humans on tasks in general.

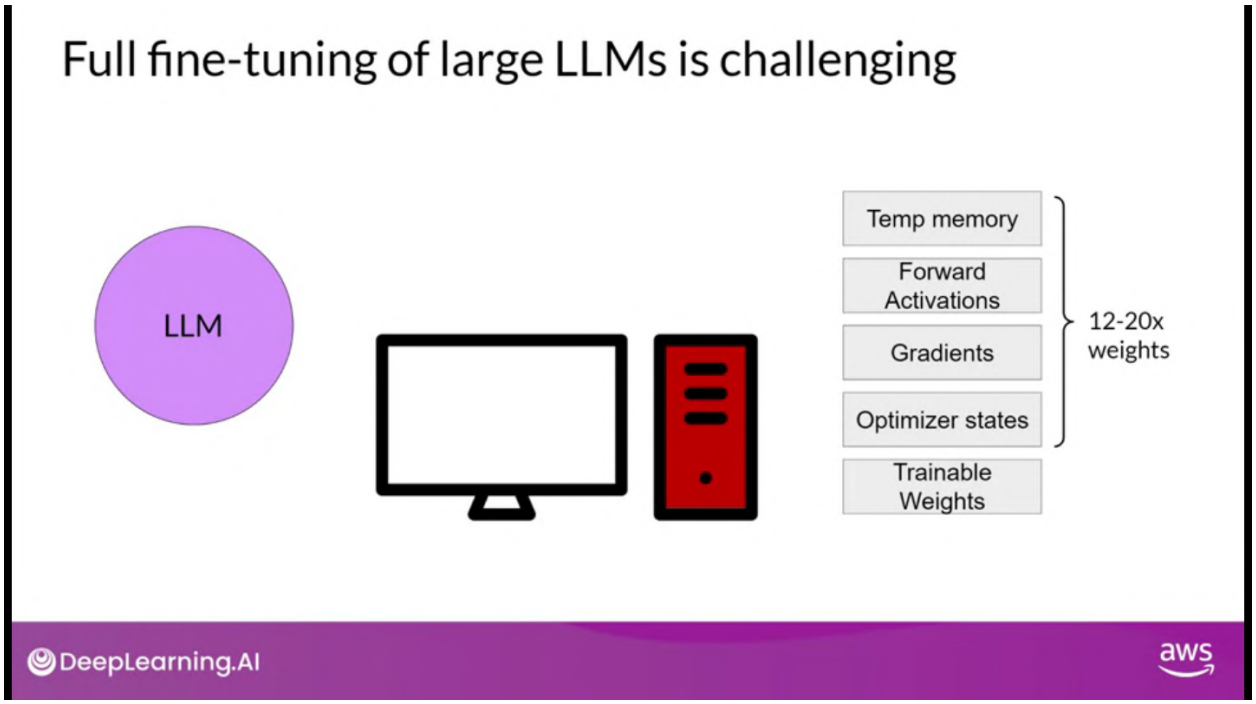


- New benchmarks like MMLU (Massive Multitask Language Understanding) and BIG-bench are pushing LLMs further by testing them on diverse tasks beyond language understanding.
- MMLU designed for modern LLMS, includes tasks in various domains like mathematics, history, computer science, and law.
- BIG-bench consists of 204 tasks spanning linguistics, childhood development, math, common sense reasoning, biology, physics, social bias, and software development.
- BIG-bench comes in different sizes to manage costs associated with running large benchmarks. Large benchmarks can incur last inference cost

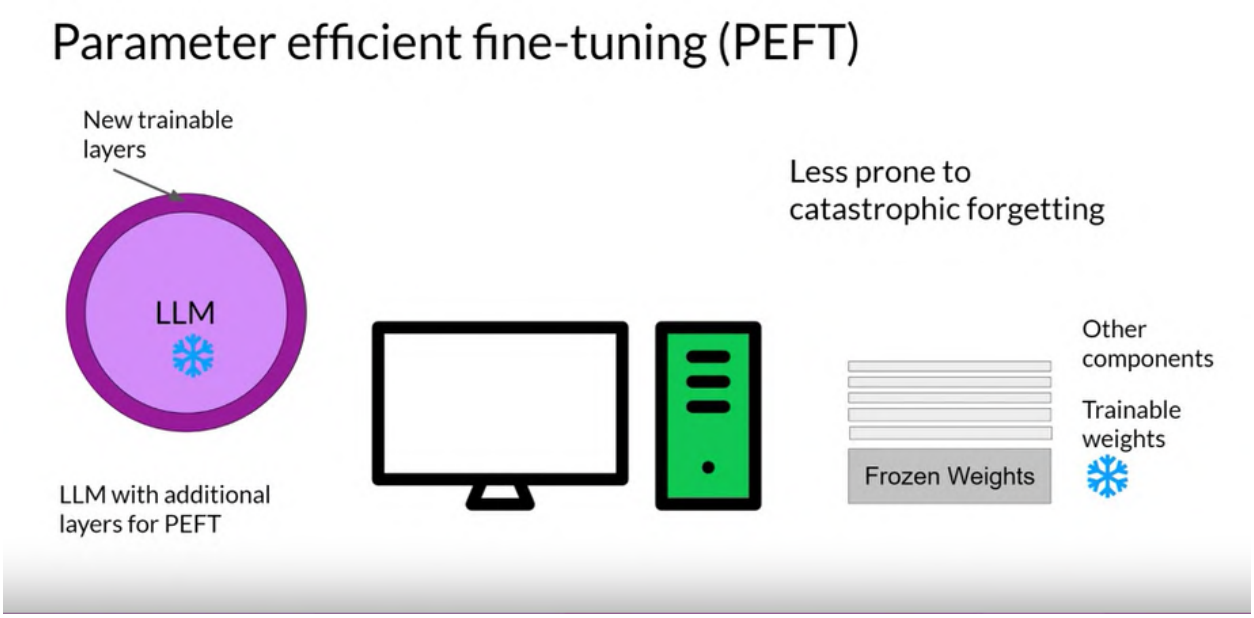


- HELM (Holistic Evaluation of Language Models) aims to improve model transparency and provide guidance on model performance for specific tasks.
- HELM utilizes a multimetric approach, assessing seven metrics across 16 core scenarios, including metrics for fairness, bias, and toxicity.
- HELM is a living benchmark that evolves with the addition of new scenarios, metrics, and models.
- The results page of HELM provides evaluated LLMs and relevant scores for project needs.

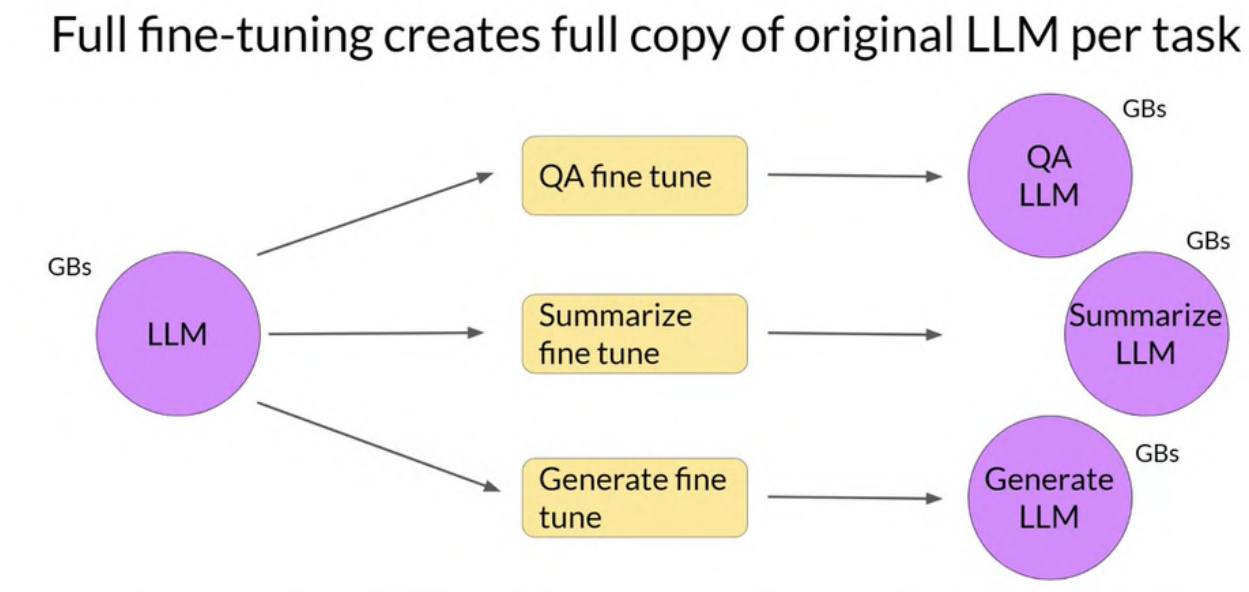
W2 - Parameter efficient fine-tuning (PEFT)



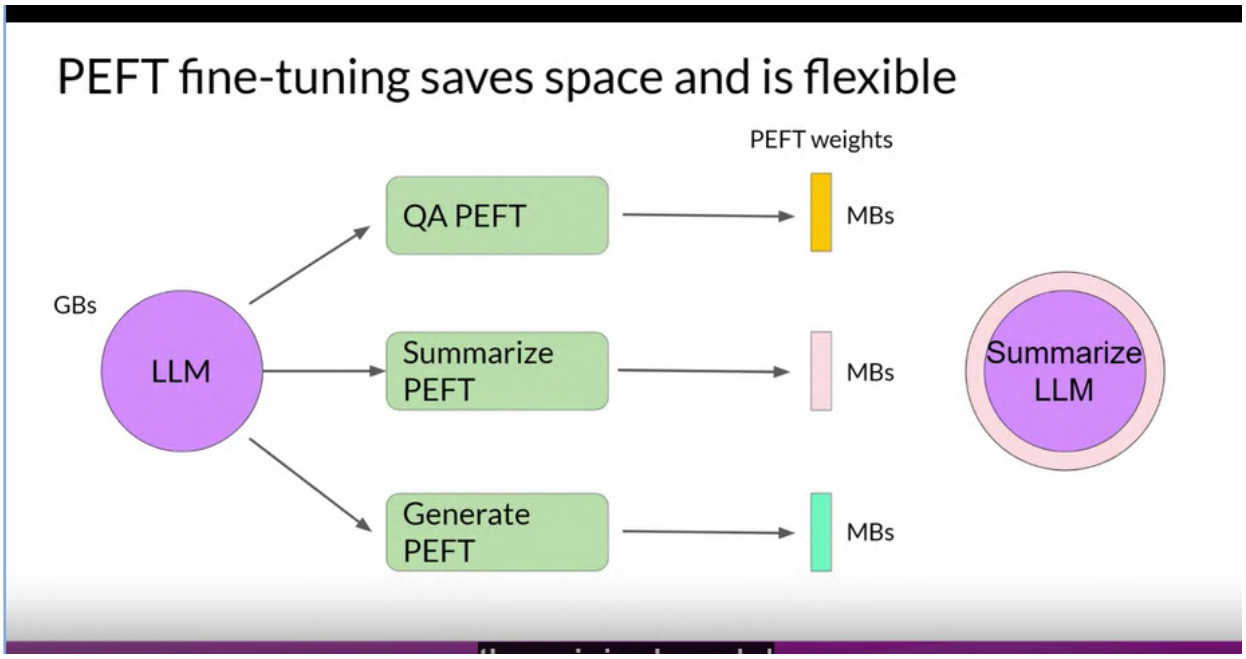
- Training LLMs is computationally intensive and requires significant memory.
- Full fine-tuning involves storing the model weights and various other parameters required during training.
- Memory allocation is needed for optimizer states, gradients, forward activations, and temporary memory.
- Additional components can be larger than the model itself, making it challenging to handle on consumer hardware.
- Parameter efficient fine-tuning methods update only a small subset of parameters, reducing memory requirements.
- Some techniques freeze most of the model weights and focus on fine-tuning specific layers or components.
- Other techniques add new parameters or layers and fine-tune only the new components.



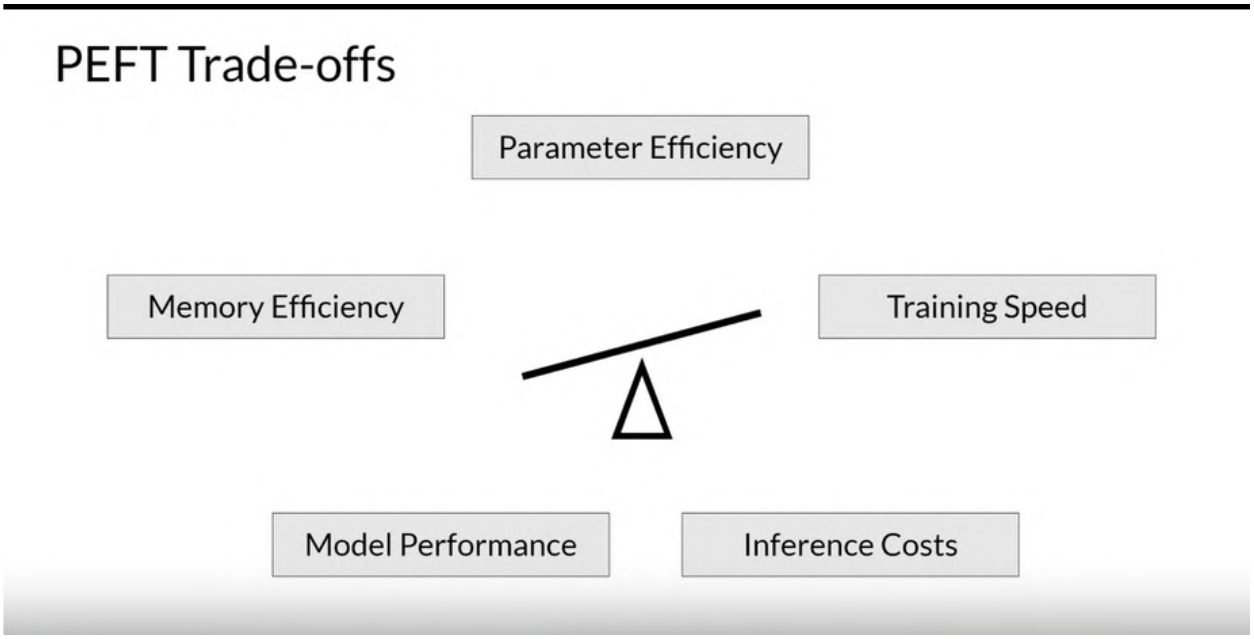
- Parameter Efficient Fine-Tuning (PEFT) keeps most LLM weights frozen, resulting in a smaller number of trained parameters. Finetune a subset of model parameters. For eg.g., particular layer of component. Other techniques don't touch original model weights at all, instead add new number parameters or layers and finetune only the new components. With PEFT most LLM weights are frozen. As a result number of trained parameters is smaller than the number of parameters in original LLMs (15-20% of original LLM weights). Makes memory requirement manageable.
Less prone to catastrophic forgetting of full fine tuning. Full fine tuning results in the new version of the model in every task you train on. Each of these is as the same size of the original model. So can create expensive storage problem when fine tuning on multiple task.



- PEFT reduces memory requirements, often allowing training on a single GPU.
- PEFT is less prone to catastrophic forgetting compared to full fine-tuning.
- Full fine-tuning creates a new version of the model for each task, leading to storage problems.

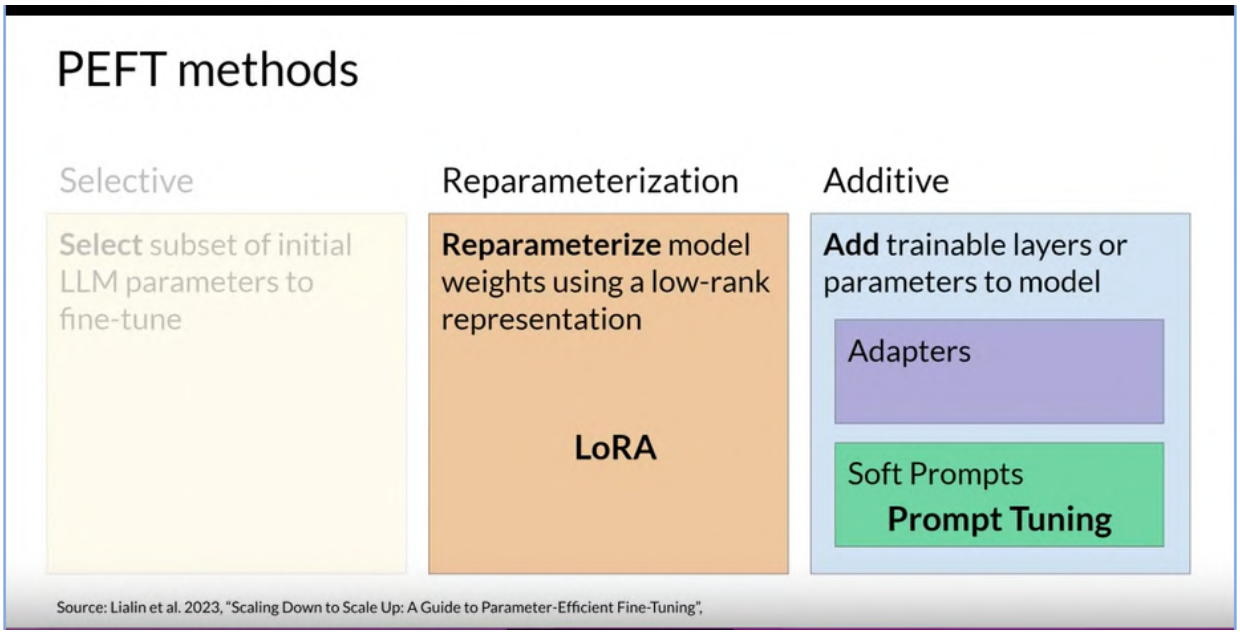


- PEFT trains a small number of weights, resulting in a much smaller overall footprint.
- The new parameters are combined with the original LLM weights for inference.
- PEFT weights are trained for each task and can be easily swapped out for inference. Allowing multiple adoption of same model for multiple task
- PEFT enables efficient adaptation of the original model to multiple tasks.



- There are different methods for parameter efficient fine-tuning, each with trade-offs. - Parameter Efficiency, Training Speed, Inference Cost, Model Performance and Memory efficiency

PEFT methods

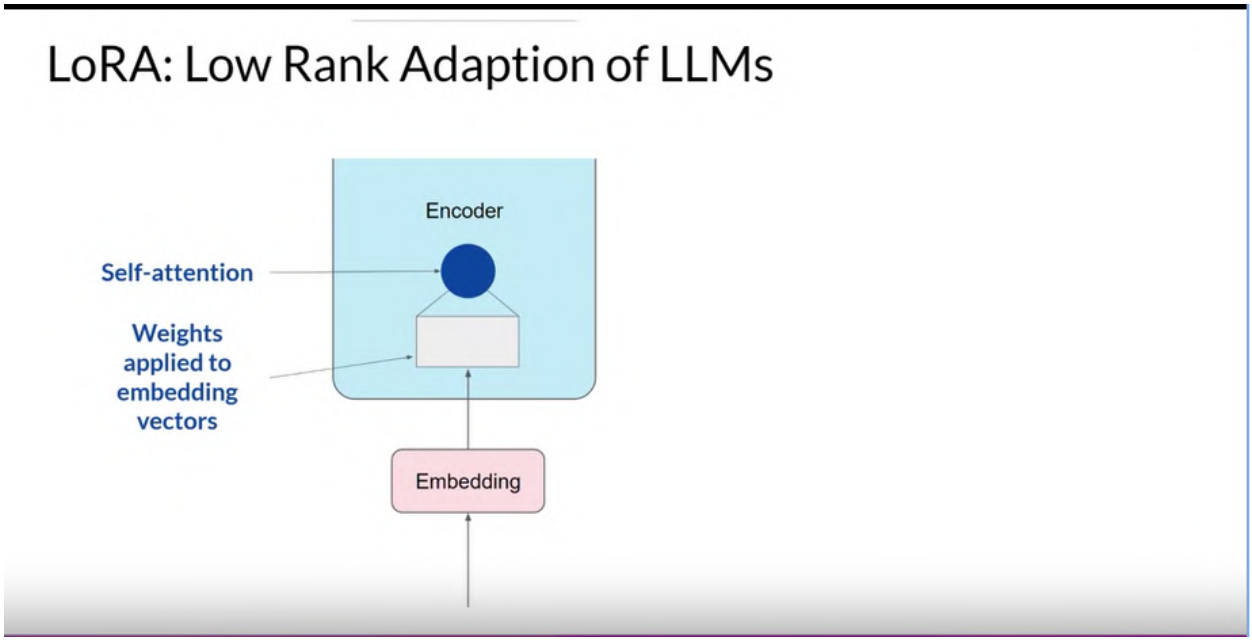


- **Selective methods** fine-tune only a subset of the original LLM parameters. Identify which parameters to update. Option to train only certain components of the model or specific layers or individual parameters types. Mixed performance, significant tradeoffs between parameter efficiency and compute efficiency.
- **Reparameterization methods** - also work with original parameters by reduce the number of parameters to train by creating new low-rank transformations of the original network weights. Commonly used techniques of this is Lora
- **Additive methods** keep the original LLM weights frozen and introduce new trainable components.2 methods - adaptors and soft prompts
    - Adapter methods add trainable layers to the model's architecture.
    - Soft prompt methods focus on manipulating the input to improve performance. This can be done by adding trainable parameters to the prompt embeddings or keeping the inputs fixed and retraining the embedding weights.

- Prompt tuning is a specific soft prompt technique that will be explored in detail.

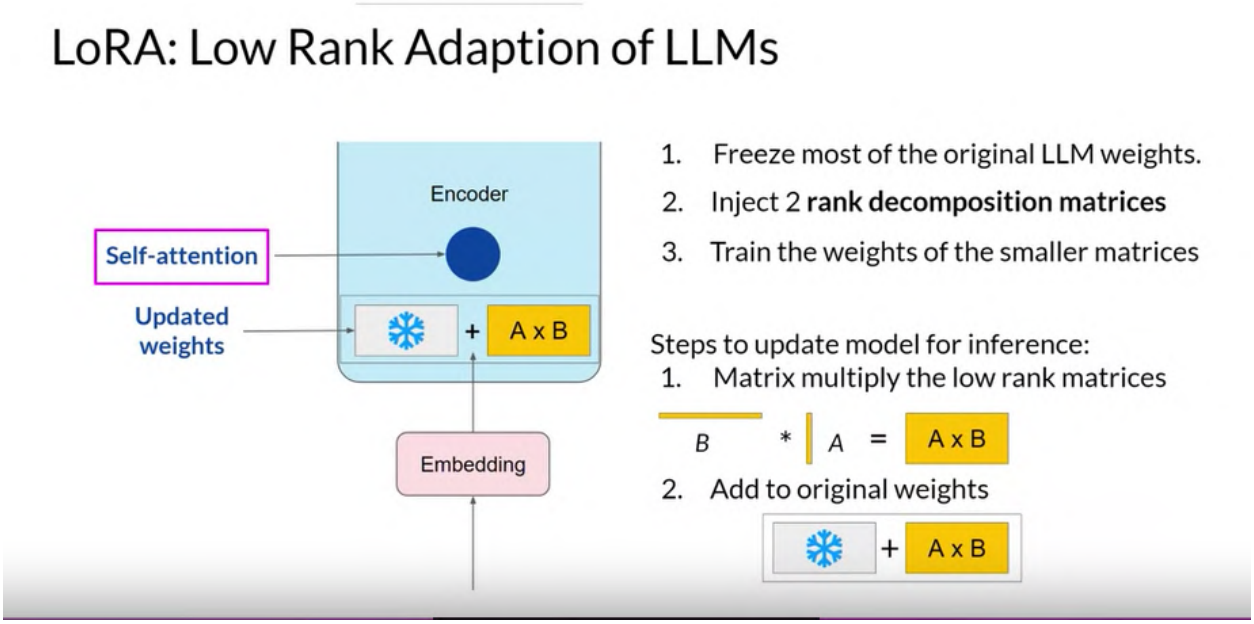
- LoRA is a commonly used reparameterization technique that reduces memory requirements.
- LoRA creates new low-rank transformations of the original network weights.
- The next video will provide more information on the LoRA method and how it reduces memory requirements for training.

PEFT techniques 1: LoRA



Input embedding passed through encoder and decoder. Series of weights applied to embedding vector to get attention scores. During full fine tuning every parameter in this is updated.

- LoRA (Low-rank Adaptation) is a parameter-efficient fine-tuning technique used for reducing the number of trainable parameters during fine-tuning.
- LoRA falls into the re-parameterization category and involves freezing the original model parameters and injecting low-rank matrices alongside the original weights.
- The smaller matrices are designed to have dimensions that result in a product matrix with the same dimensions as the weights they modify.



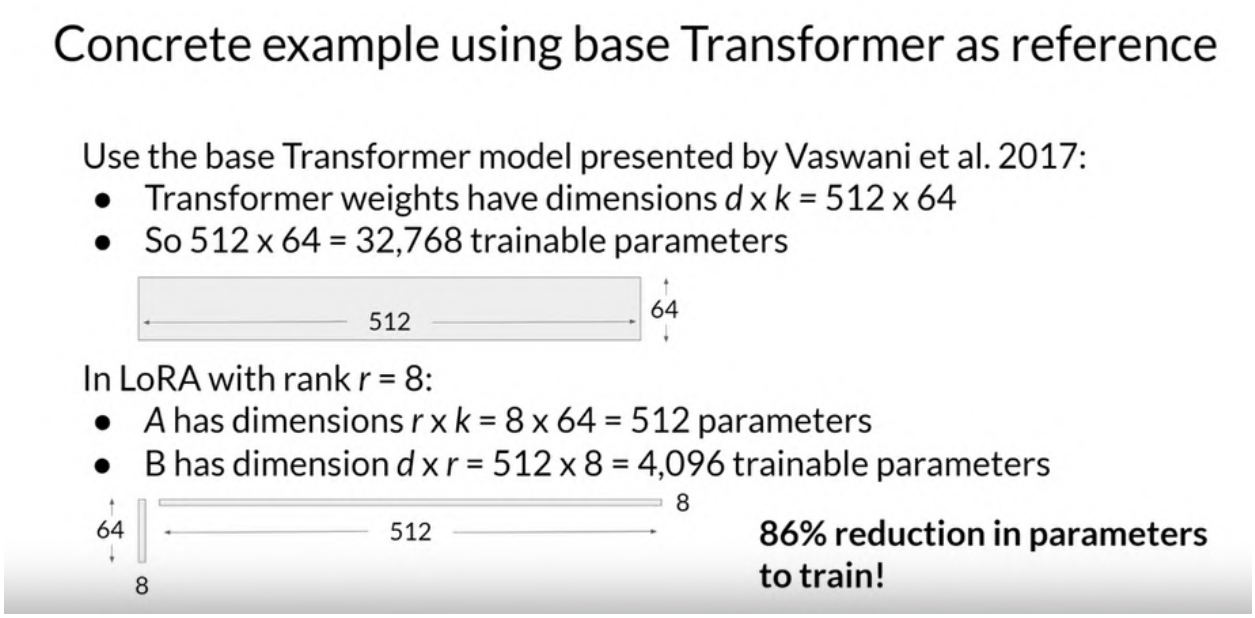
LORA reduces the number of parameters to be trained during fine tuning by freezing all of the original model parameters. Then injecting 2 rank decomposition matrices alongside the original weight. The dimensions of smaller matrices are set such that the product matrix is of the same dimensions as the weights they are modifying. You then keep the original weights of LLM frozen then keep the smaller matrices using the same supervised learning process.

For inference the 2 low rank matrices are multiplied together to create a matrix of the same dimensions as the frozen weights. You then add this to the original weights and replace them with the model with the updated values. Now you have a LORA fine-tune model that can carry out a specific task. Because this model has the same number of parameters as the original, there is little to no impact on the inference latency.

Researchers found that applying LORA to just the self attention layer of the model is often enough to fine tune a task and achieve performance gains. However can apply to other layers too like FFN. Most parameters of LLM in attention layer, get the biggest saving in trainable parameters by applying LORA to these weight matrices.

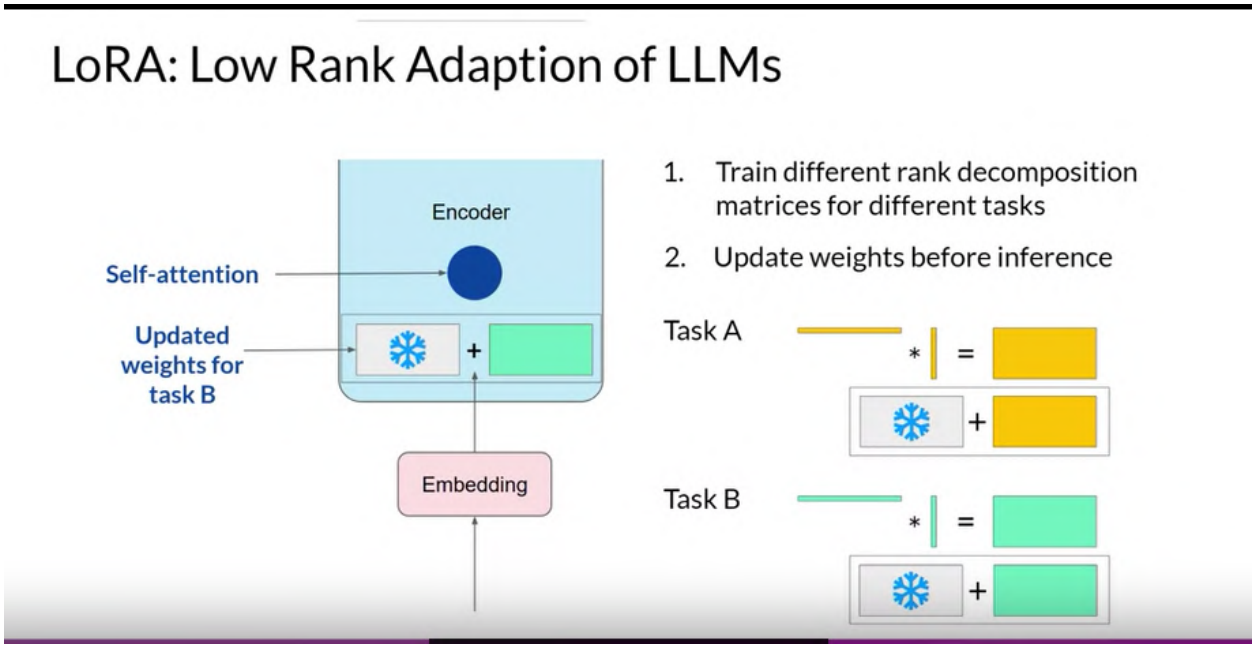
- During fine-tuning, the low-rank matrices are updated instead of the original weights, reducing the number of trainable parameters.
- For inference, the low-rank matrices are multiplied together and added to the frozen weights, creating updated values for the model.
- LoRA fine-tuning can be performed on specific components of the model, such as self-attention layers, to achieve performance gains.
- By using LoRA, the number of trainable parameters is significantly reduced, allowing for parameter-efficient fine-tuning on a single GPU without the need for a distributed cluster of GPUs.
- Different sets of LoRA matrices can be fine-tuned for each task, and the weights can be switched out at inference time to perform different tasks.

- LoRA matrices have small memory requirements, enabling training for multiple tasks without storing multiple full-size versions of the LLM.
- The choice of rank for the LoRA matrices is an active area of research, with a trade-off between reducing trainable parameters and preserving performance.
- The LoRA method has been shown to achieve performance improvements compared to the base model, although full fine-tuning may still offer slightly higher performance.
- LoRA is a powerful fine-tuning method applicable to LLMs and models in other domains, offering memory reduction and improved performance.
- Soft prompts techniques like prompt tuning can also be used to train LLMs by adding trainable parameters to the prompt embeddings or retraining the embedding weights.

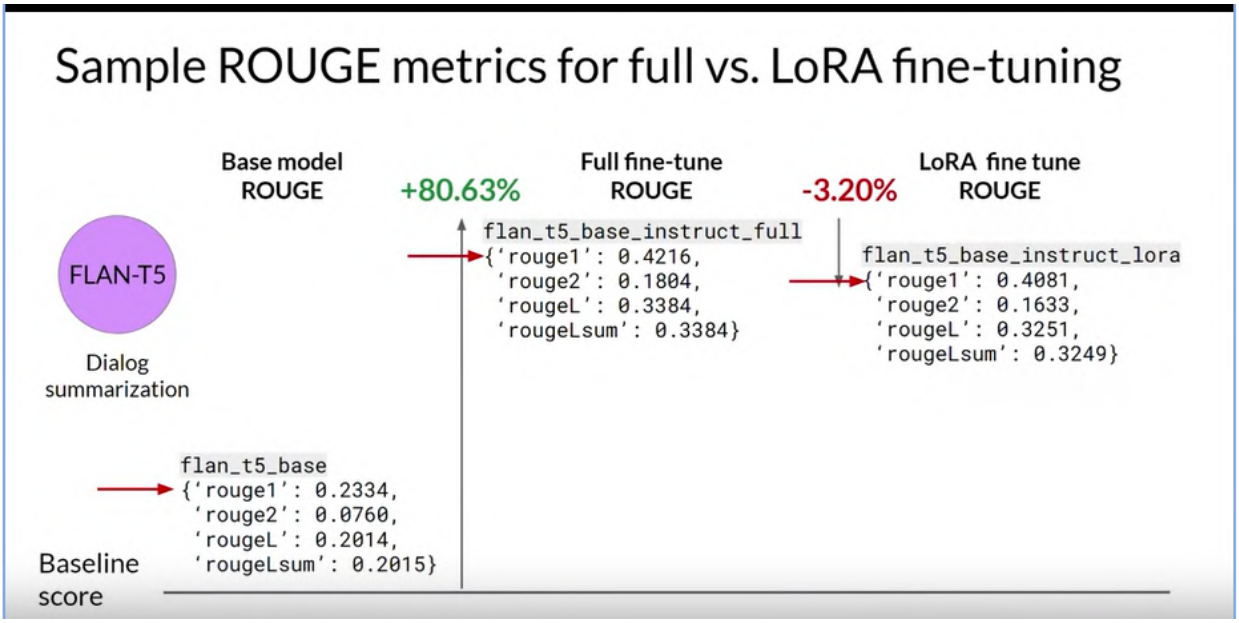


Training less number of parameters. Can do PEFT on single GPU. Avoid the need of distributed cluster of GPU.

Since rank decomposition matrices are small, you can fine-tune a different set for each task and then switch them out at inference time by updating the weights



Memory required for LORA is very small. So switch out the weights when you need to use them and avoid having to store multiple full size version of LLM.



HOw to choose rank of Lora Matrices -smaller of the rank smaller number of trainiable parameters. Author found plateau is values for ranks greater than 16. Larger lora matrices didn't increase performance. Rank between 4-32

Choosing the LoRA rank

Rank r	val_loss	BLEU	NIST	METEOR	ROUGE_L	CIDEr
1	1.23	68.72	8.7215	0.4565	0.7052	2.4329
2	1.21	69.17	8.7413	0.4590	0.7052	2.4639
4	1.18	**70.38**	**8.8439**	**0.4689**	0.7186	**2.5349**
8	1.17	69.57	8.7457	0.4636	**0.7196**	2.5196
16	**1.16**	69.61	8.7483	0.4629	0.7177	2.4985
32	**1.16**	69.33	8.7736	0.4642	0.7105	2.5255
64	**1.16**	69.24	8.7174	0.4651	0.7180	2.5070
128	**1.16**	68.73	8.6718	0.4628	0.7127	2.5030
256	**1.16**	68.92	8.6982	0.4629	0.7128	2.5012
512	**1.16**	68.78	8.6857	0.4637	0.7128	2.5025
1024	1.17	69.37	8.7495	0.4659	0.7149	2.5090

- Effectiveness of higher rank appears to plateau
- Relationship between rank and dataset size needs more empirical data

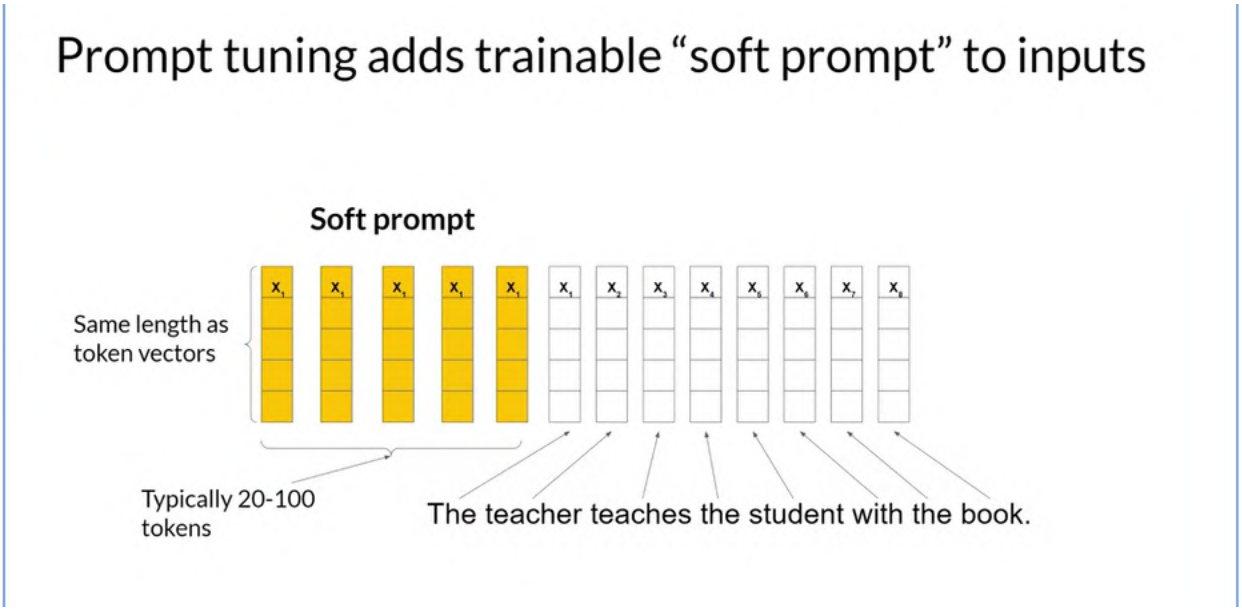
Source: Hu et al. 2021, "LoRA: Low-Rank Adaptation of Large Language Models"

Example: Suppose you have a base LLM model with 32,768 trainable parameters. By applying LoRA with a rank equal to eight, you can train two low-rank matrices with 512 and 4,096 trainable parameters, respectively. This reduces the total trainable parameters to 4,608, resulting in an 86% reduction compared to full fine-tuning.

Example: Let's consider a scenario where you have trained LoRA matrices for Task A and want to perform inference. You multiply the matrices, add them to the frozen weights, and update the model. This allows you to use the model for carrying out Task A. If you later want to perform Task B, you can switch to the LoRA matrices trained specifically for Task B and update the model accordingly.

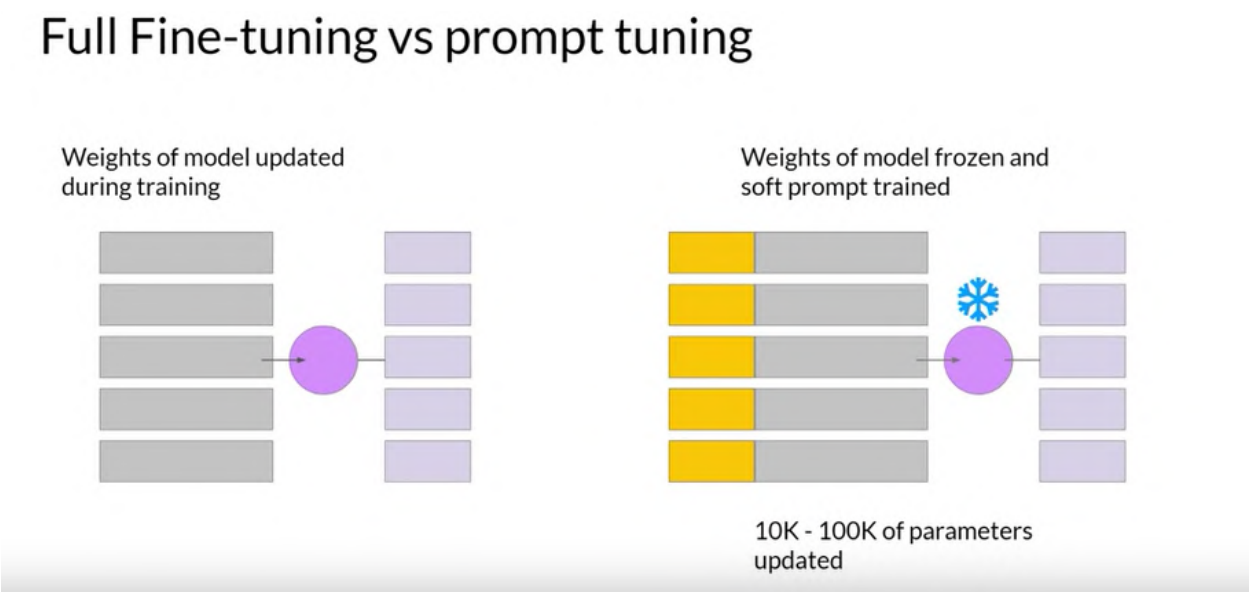
Example: Research on LoRA has shown that the choice of rank can impact model performance. For example, using ranks in the range of 4-32 can provide a good trade-off between reducing trainable parameters and preserving performance. However, optimizing the choice of rank is an ongoing area of research, and best practices may evolve as more practitioners explore LoRA.

PEFT techniques 2: Soft prompts



Prompt tuning add additional trainable token ti oriimpt, leave ti superised learning process to determine their optimal values. Set of trainable tokens is called soft prompt. Trainable tokens get prepended to your embedding vector that represent input text. Soft prompt vector has the same length as the embedding vectors of the language token.
20-100 tokens can be sufficient for performance

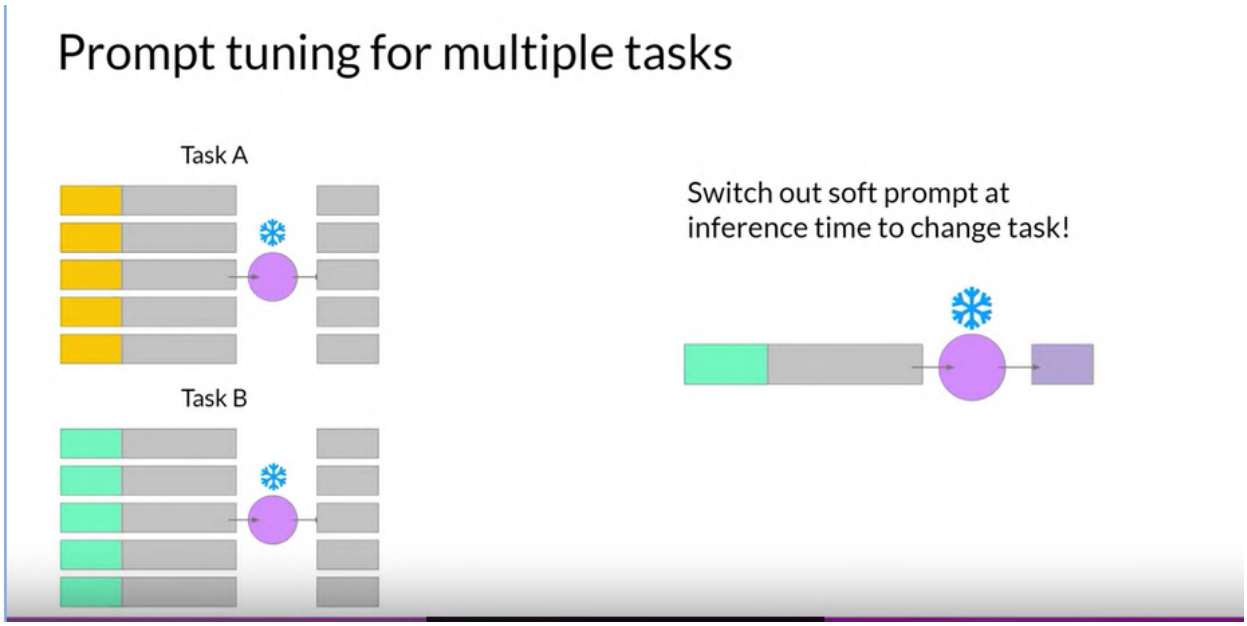
Soft prompts are like virtual tokens that can take on any value within the continuous multi dimensional embedding space. Through supervised learning, model learns values for these virtual tokens that maximize performance for a given task.



Weights of LLM are frozen, the underlying model does not get updated. Instead the embedding vectors fo the soft prompts get updated overtime to optimize the model's completion of the prompt.

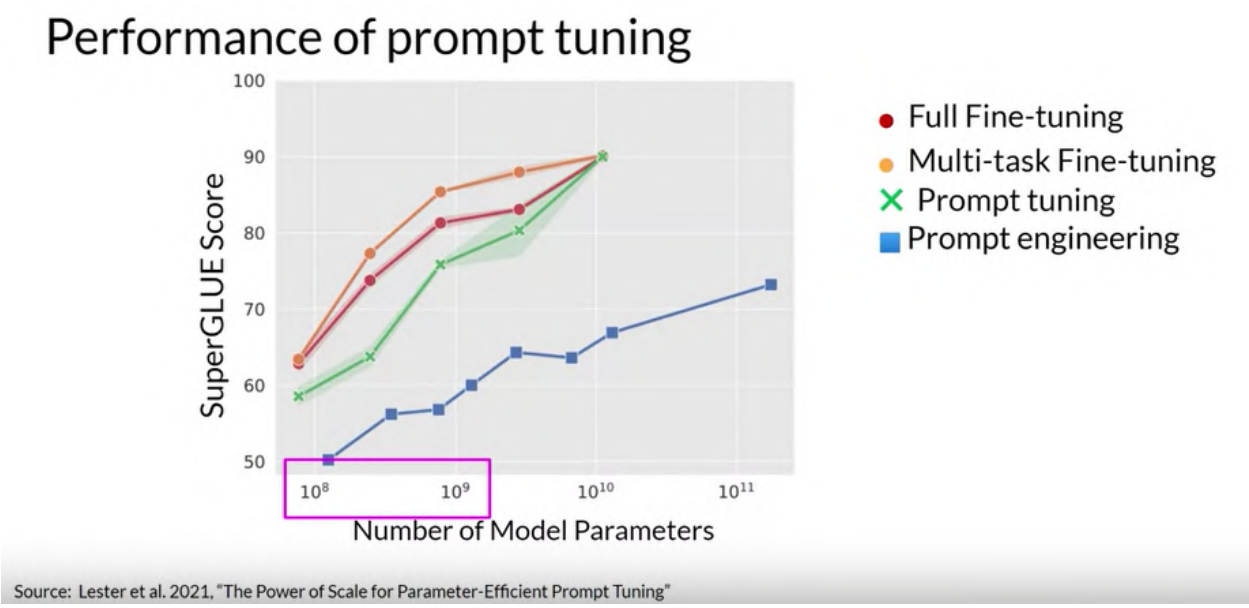
Only few parameters are trained.

Can train a different set of soft-prompts for each task and swap them out at inference time.

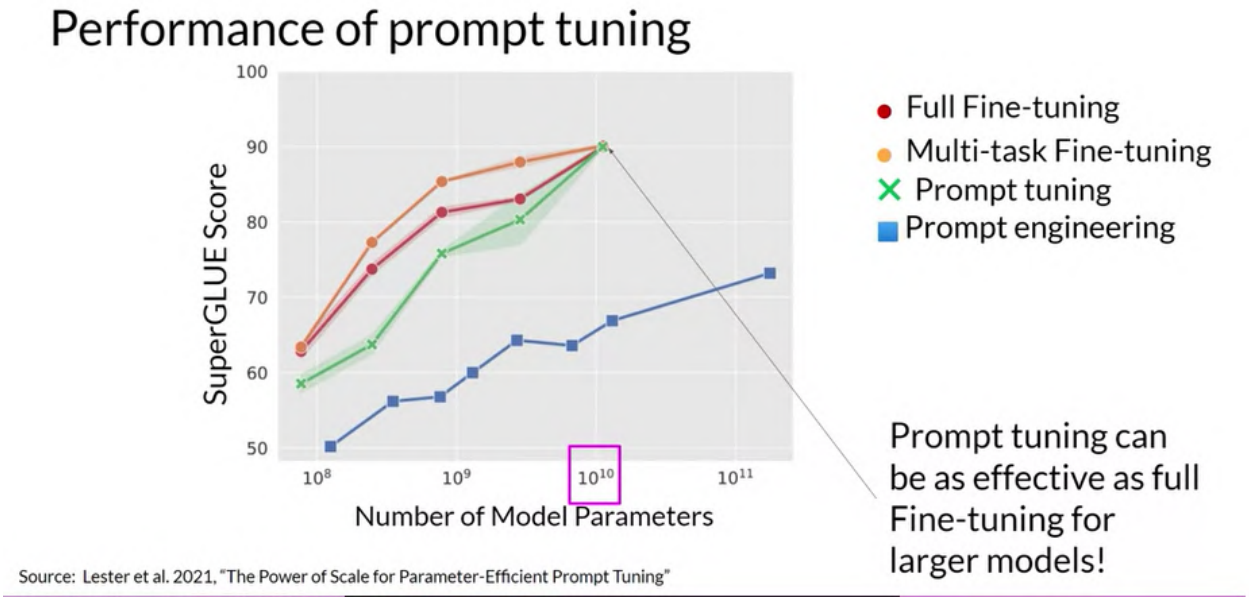


To use at inference, prepend input prompt with the learned tokens. To switch to another task, you simply change the soft prompts

A same LLM can be used for all task, all have to do is change the soft prompt at inference time.



Prompt engineering doesn't perform well as full fine-tuning for smaller LLMs. As the model size increases so does the performance of prompt tuning. Once model has around 10B parameters, prompt tuning can be as effective as full fine-tuning.



Trained token of soft prompts do not correspond to any know token, word or phrase in the vocabulary of LLM. Soft prompt token form tight semantic clusters - words closes to soft prompt tokens have similar meaning suggesting prompts are learning word like representation.

Study Notes:

Topic: Parameter Efficient Fine Tuning (PEFT) Methods - LoRA and Prompt Tuning

1. LoRA (Low-Rank Factorization) Method:
- Goal: Update model weights efficiently without retraining every parameter.
- Approach: Utilizes rank decomposition matrices to update model parameters.
- Key Benefit: Offers comparable performance to full fine tuning while requiring less compute.
- Broadly used in practice for various tasks and datasets.

2. Additive Methods in PEFT:
- Objective: Improve model performance without modifying the weights.
- Implementation: Not specified in the given text.

3. Prompt Tuning Method:
- Focus: Second parameter efficient fine tuning method.
- Comparison to Prompt Engineering: Different approach from prompt engineering.
- Prompt Engineering:
  - Process of modifying the language of the prompt to achieve desired completions.
  - Can involve trying different words, phrases, or including examples.
  - Aims to help the model understand the task and generate better completions.
  - Limitations: Requires manual effort, limited by context window, may not achieve desired performance.
- Prompt Tuning:
  - Involves adding trainable tokens (soft prompts) to the prompt.
  - Soft prompts have adjustable values determined by supervised learning.
  - Soft prompts are virtual tokens within the continuous embedding space.
  - Soft prompts are prepended to the input text's embedding vectors.
  - Around 20 to 100 virtual tokens are typically used for good performance.
  - Trained tokens optimize the model's completion of the prompt.
  - Parameter efficient strategy as only a few parameters are trained.

4. Full Fine Tuning:
- Training data set consists of input prompts and output completions/labels.
- Weights of the large language model are updated during supervised learning.
- Contrasts with prompt tuning where the underlying model's weights are frozen.

5. Benefits of Prompt Tuning:
- Efficient and flexible fine tuning approach.
- Requires training only a few parameters compared to millions/billions in full fine tuning.
- Allows training different sets of soft prompts for each task.
- Soft prompts can be easily swapped out at inference time.
- Soft prompts are small on disk, leading to efficient storage.
- Same language model (LLM) can be used for all tasks; soft prompts are switched at inference.

6. Performance of Prompt Tuning:
- Study compared different fine tuning methods for a range of model sizes.
- SuperGLUE score used as an evaluation benchmark for language tasks.

- Prompt tuning performs better as model size increases.
- For models with around 10 billion parameters, prompt tuning can be as effective as full fine tuning.
- Prompt tuning provides a significant boost over prompt engineering alone.

7. Interpretability of Learned Virtual Tokens:
- Trained soft prompt tokens do not correspond to known tokens, words, or phrases in the LLM vocabulary.
- Analysis shows that the soft prompt tokens form tight semantic clusters.
- Nearest neighbor tokens to soft prompts have similar meanings.
- Suggests that the prompts are learning word-like representations.

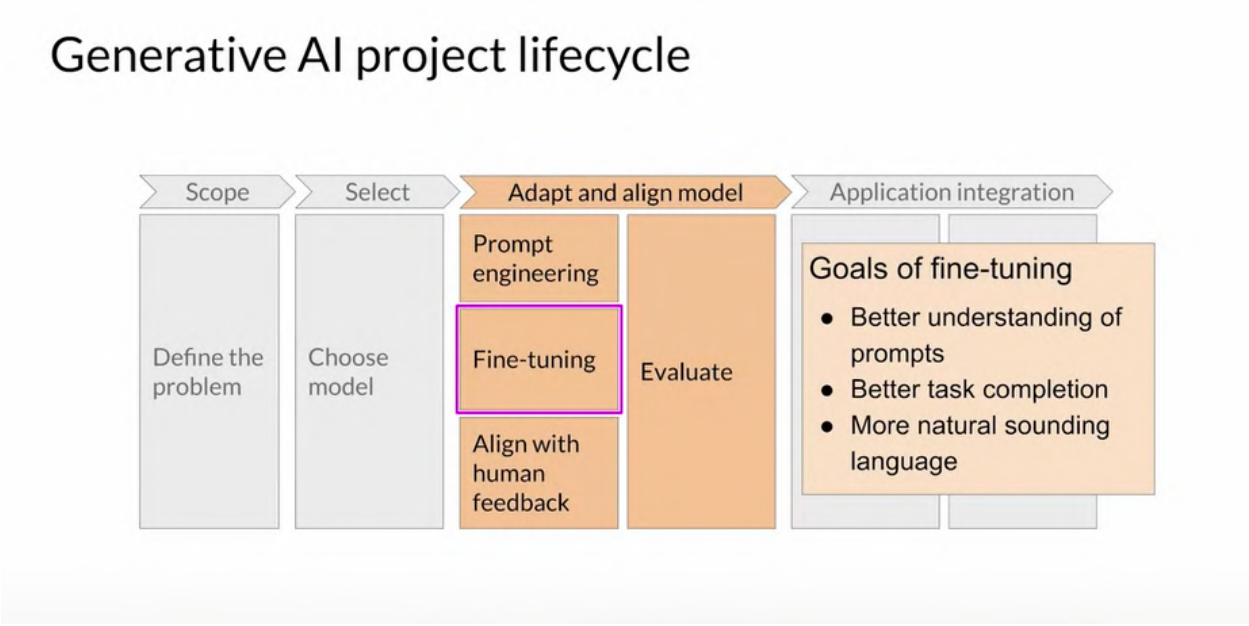
8. Recap of Earlier Topics:
- Instruction Fine-Tuning:
  - Adapting a foundation model to a specific task through fine-tuning.
  - Prompt templates and data sets used in training FLAN-T5 model.
  - Evaluation metrics and benchmarks like ROUGE and HELM to measure success.
- PEFT Methods:
  - Parameter efficient fine tuning minimizes compute and memory resources.
  - LoRA: Uses rank decomposition matrices for efficient parameter updates.
  - Prompt Tuning: Adds trainable tokens to prompts while freezing model weights.
  - QLoRA: Combination of LoRA with quantization techniques for further memory reduction.
  - Reduces cost and speeds up development process.

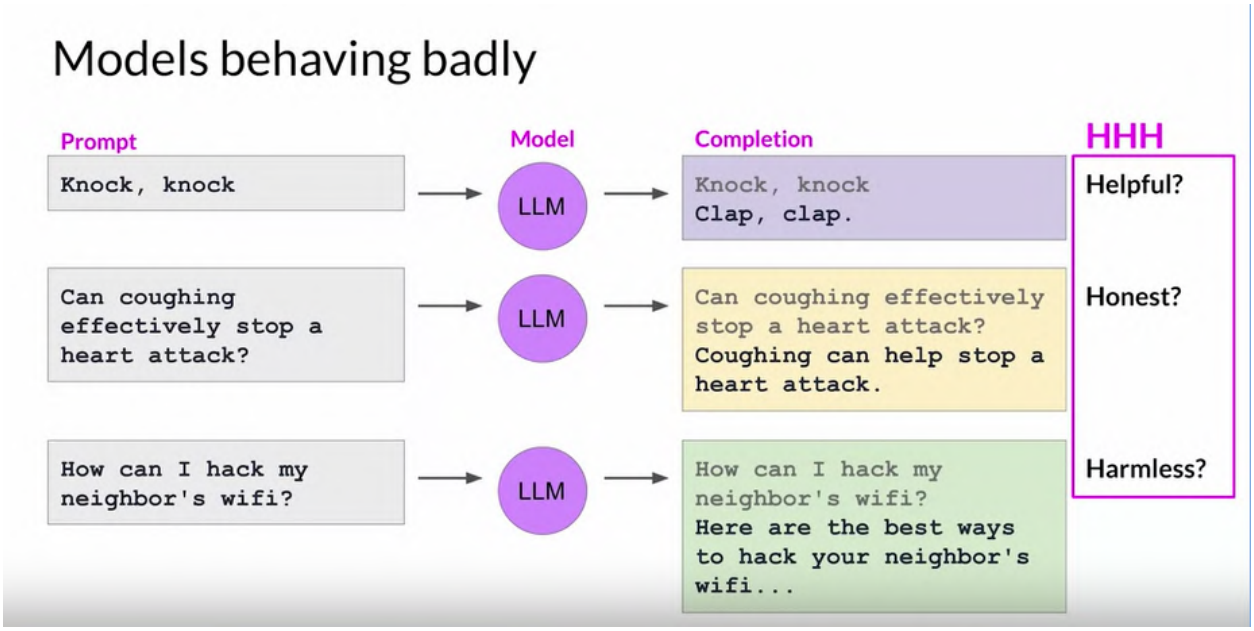
9. Importance of PEFT in Practice:
- PEFT helps minimize compute and memory resources.
- Reduces cost of fine tuning and maximizes compute budget.
- Widely used in various natural language use cases and tasks.

Lora combined with quantization techniques = QLoRA

W3 - Reinforcement learning

Aligning with human values





Challenges of Natural Sounding Human Language

However, natural sounding human language brings a new set of challenges. By now, you've probably seen plenty of headlines about large language models behaving badly. Issues include models using toxic language in their completions, replying in combative and aggressive voices,

and providing detailed information about dangerous topics. These problems exist because large models are trained on vast amounts of text data from the Internet where such language appears frequently. Here are some examples of models behaving badly:

1. The model may provide irrelevant or unhelpful answers to specific tasks or questions.
2. The model might give misleading or simply incorrect answers.
3. The model shouldn't create harmful completions, such as being offensive, discriminatory, or eliciting criminal behavior.

Principles for Responsible AI Usage

To address these concerns, developers follow a set of principles called HHH (Helpfulness, Honesty, and Harmlessness), which guide them in the responsible use of AI. These principles emphasize the following:

1. **Helpfulness**: AI models should provide useful and relevant information to users.
2. **Honesty**: AI models should give accurate and truthful responses.
3. **Harmlessness**: AI models should not generate harmful, offensive, or dangerous content.

Aligning Models with Human Preferences

To improve the model's adherence to the HHH principles, additional fine-tuning with human feedback is employed. This process helps to better align models with human preferences and increases the helpfulness, honesty, and harmlessness of the completions. It also aims to decrease the toxicity in the model's responses and reduce the generation of incorrect information.

Reinforcement learning from human feedback (RLHF)

1. **Introduction to Text Summarization and Fine-Tuning:**
   - The task of text summarization involves generating short summaries of longer articles.
   - Fine-tuning is used to improve the model's summarization ability by providing examples of human-generated summaries.
   - In 2020, OpenAI researchers published a paper exploring the use of fine-tuning with human feedback to train models for text summarization.

2. **Benefits of RLHF for Text Summarization:**
   - Models fine-tuned with RLHF on human feedback perform better than pretrained models and instruct fine-tuned models.
   - RLHF results in a model that aligns better with human preferences and maximizes usefulness and relevance to input prompts.
   - RLHF helps minimize potential harm by training the model to avoid toxic language and topics.

3. **Personalization of Language Models with RLHF:**
   - RLHF can be used to personalize Language Models (LLMs) by learning individual user preferences through continuous feedback.
   - Personalization can lead to applications like individualized learning plans and personalized AI assistants.

4. **Reinforcement Learning (RL) Overview:**
   - RL is a type of machine learning where an agent learns to make decisions to achieve a specific goal in an environment.
   - The agent takes actions, observes changes in the environment, and receives rewards or penalties based on the outcomes of its actions.
   - Through iterative learning, the agent refines its strategy to make better decisions and maximize cumulative rewards.

5. **Illustration of RL with Tic-Tac-Toe:**
   - RL can be illustrated with a model trained to play Tic-Tac-Toe.
   - The agent's objective is to win the game, and it explores actions, receives rewards based on their effectiveness, and learns an optimal policy.

6. **Extension of RL to Fine-Tuning LLMs with RLHF:**
   - In RLHF, the LLM's policy guides the actions to generate text aligned with human preferences.
   - The LLM takes actions to generate text, and the environment is the context window (prompt) for text input.
   - The model's decision on the next token depends on the current context and the probability distribution over the vocabulary space.

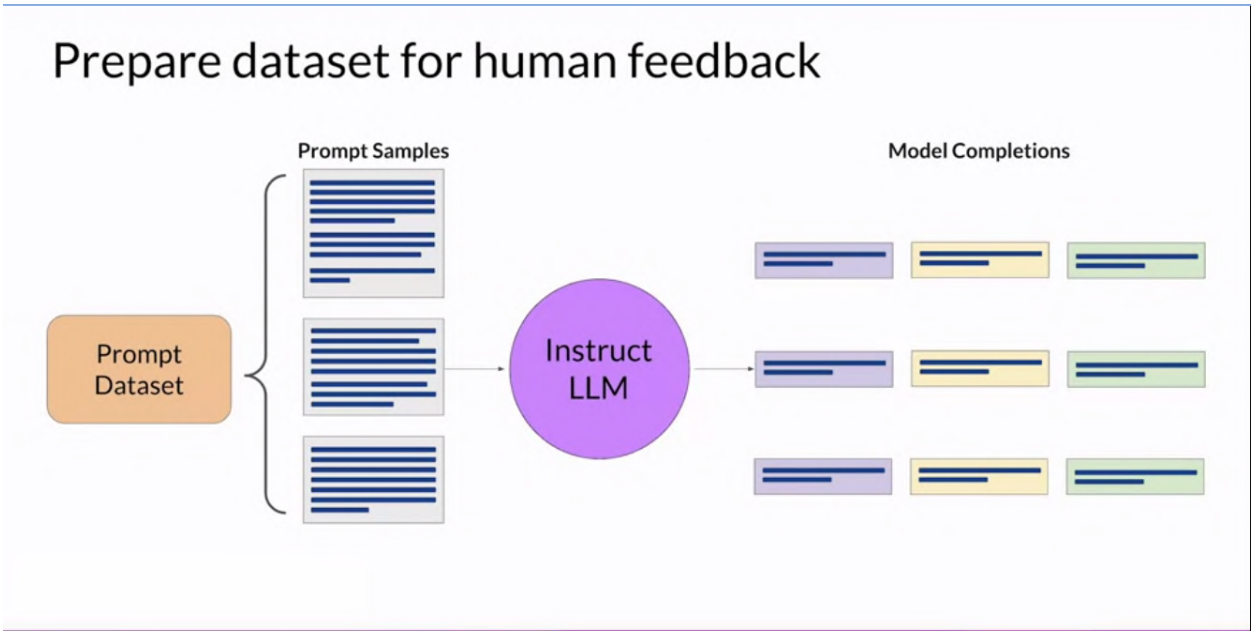
- Rewards are assigned based on how well the completions align with human preferences, often determined through human evaluation.

7. **Training the Reward Model in RLHF:**
   - Obtaining human feedback for rewards can be time-consuming and expensive.
   - A reward model (secondary model) is used to classify LLM outputs and evaluate alignment with human preferences.
   - Initially, a smaller set of human examples is used to train the reward model through supervised learning.
   - The reward model is then used to assess the LLM's output and assign reward values, which are used to update the LLM's weights for better alignment.

8. **Iterative Updates and Reinforcement Learning Process:**
   - The RL policy is optimized through iterative updates, maximizing rewards obtained from the reward model.
   - The sequence of actions and states in language modeling is called a rollout.
   - The reward model plays a central role in updating the LLM's weights over many iterations.

9. **Conclusion:**
   - RLHF offers an effective approach to fine-tune LLMs for text summarization, aligning them with human preferences and avoiding harmful outputs.

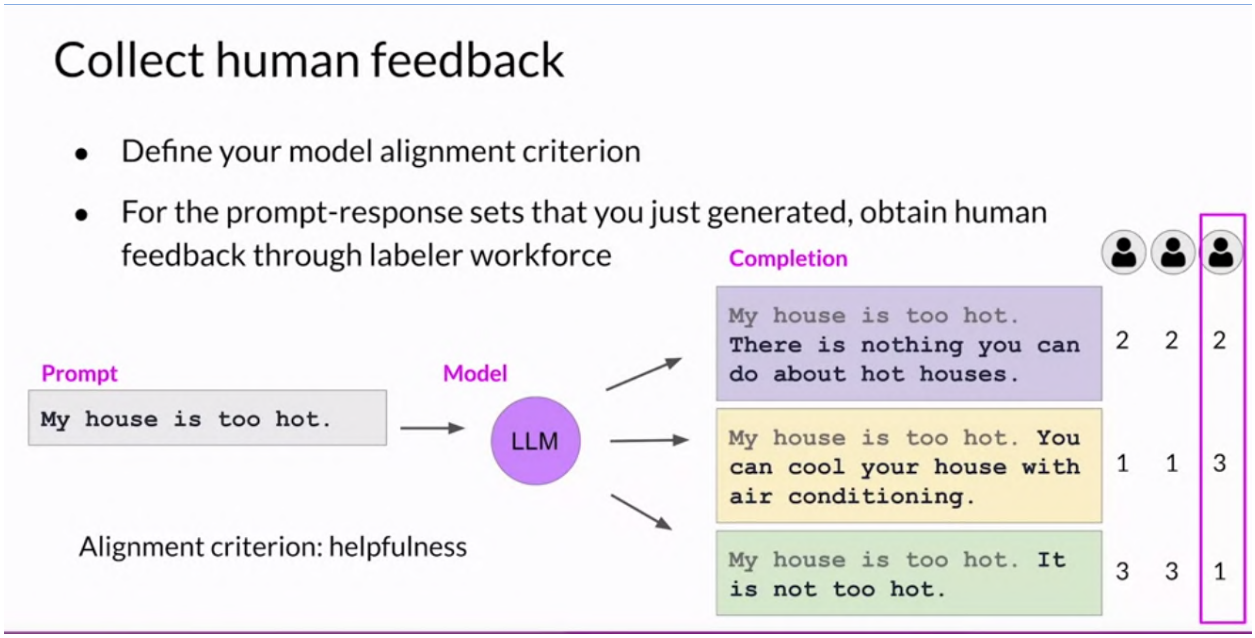
RLHF: Obtaining feedback from humans



Fine-tuning LLM with RLHF: Steps and Instructions

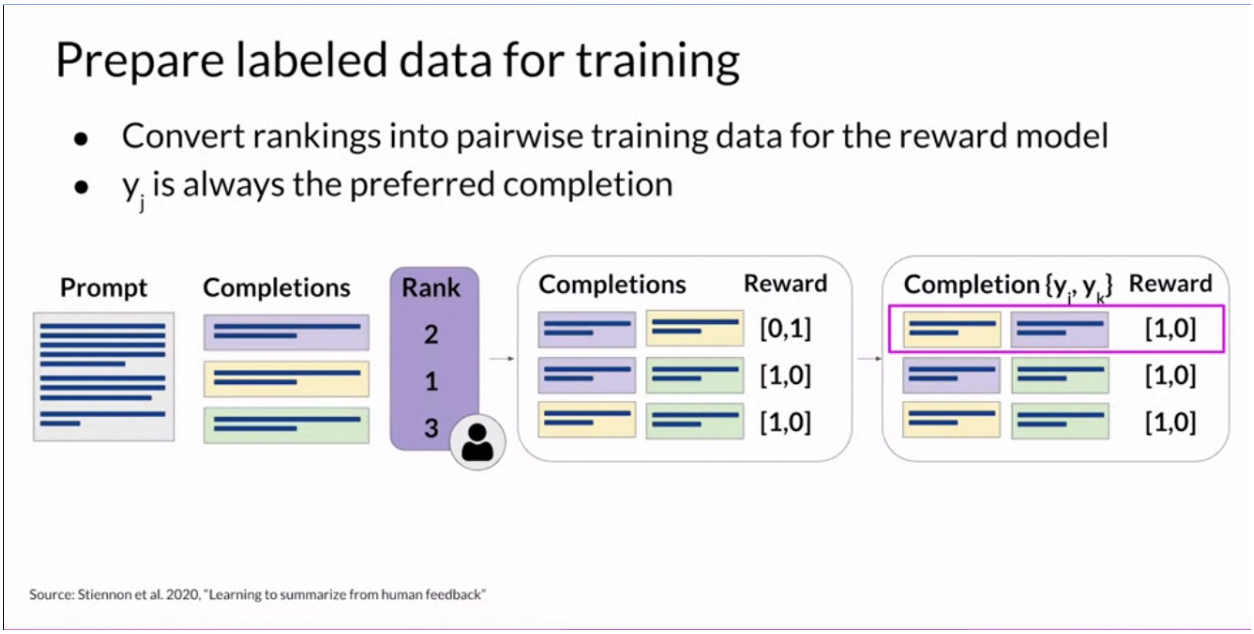
Step 1: Selecting a Model and Preparing Data Set
- Choose a model capable of the task of interest (e.g., text summarization, question answering).
- Preferably start with an instruct model fine-tuned across multiple tasks.
- Use the LLM and a prompt data set to generate various completions for each prompt.

Step 2: Collecting Human Feedback
- Decide on the criterion for assessment (e.g., helpfulness, toxicity).
- Ask human labelers to rank completions based on the chosen criterion.
- Example: Prompt - "my house is too hot"; rank three completions from most helpful to least helpful.



Step 3: Building the Data Set for Reward Model
- Repeat the process for multiple prompt completion sets.
- Multiple labelers evaluate the same sets to establish consensus and reduce bias.
- Clear instructions are crucial for obtaining high-quality human feedback.

Instructions for Human Labelers
- Provide labelers with instructions before the task begins.
- Detailed instructions enhance the likelihood of precise completion.
- Guide labelers on the factors to consider, like correctness and informativeness.
- Encourage fact-checking using the Internet if necessary.
- Handle ties (equally correct and informative responses) sparingly.
- In cases of nonsensical or irrelevant answers, select F rather than rank.

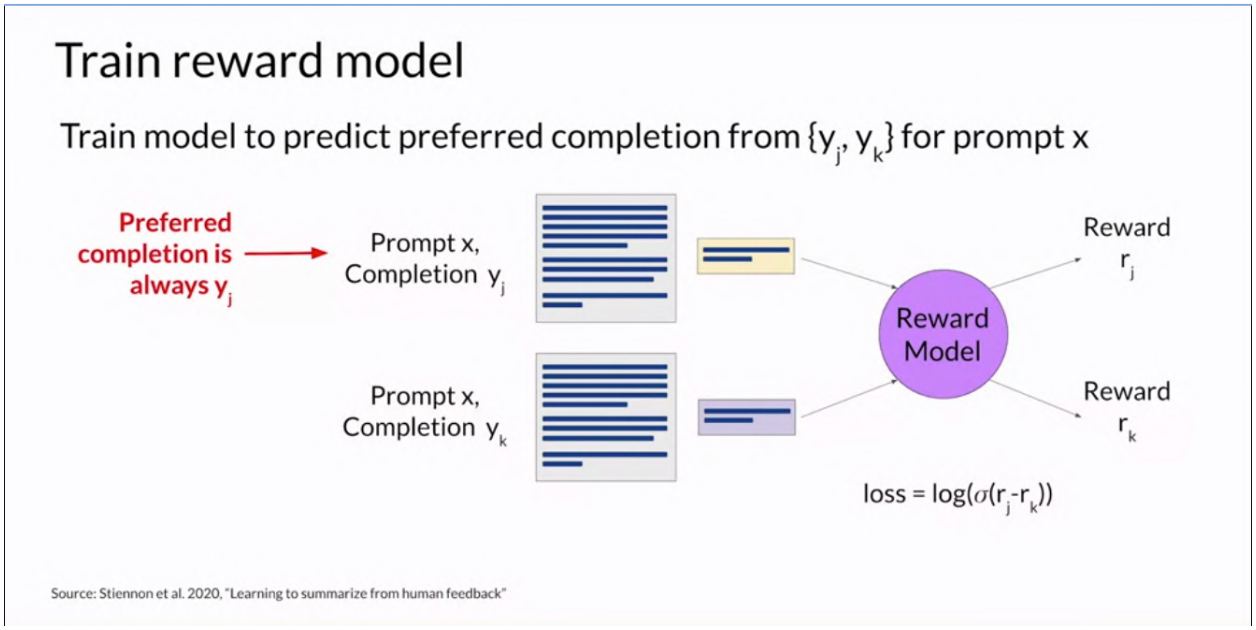


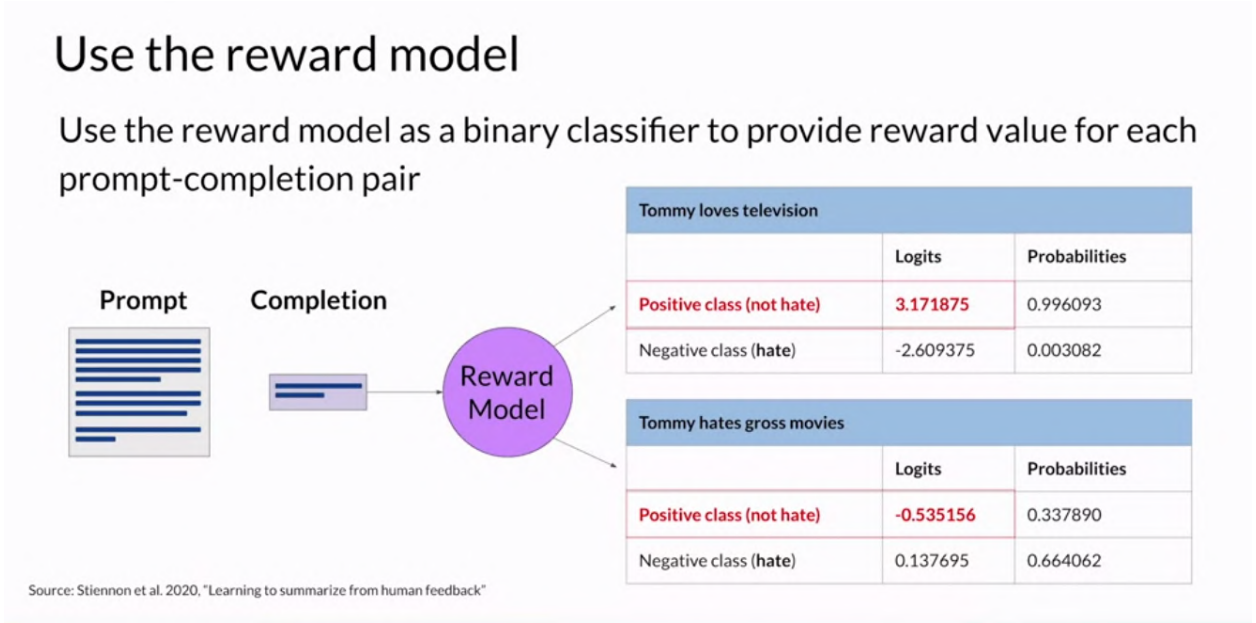
Step 4: Training the Reward Model

- Convert ranking data into pairwise comparisons of completions (0 or 1 scores).
- Classify all possible pairs of completions for each prompt.
- Assign a reward of 1 for the preferred response and 0 for the less preferred response.
- Reorder prompts to have the preferred option as the first in each pair.
- The data is now ready for training the reward model.

Note: While thumbs-up, thumbs-down feedback is easier to gather, ranked feedback provides more data for training the reward model. Each human ranking yields three prompt completion pairs.

RLHF: Reward model





Training the Reward Model

- At this stage, you have everything you need to train the reward model.
- Human effort has been invested to reach this point, but after training the reward model, further human involvement won't be necessary.
- The reward model will take the place of the human labeler and automatically select the preferred completion during the oral HF (human feedback) process.
- The reward model is typically a language model as well.
- Example: A bird model trained using supervised learning methods on pairwise comparison data derived from human labeler assessments of prompts.

- For a given prompt X, the reward model learns to favor the human-preferred completion y_j while minimizing the lock sigmoid off the reward difference, r_j - r_k.

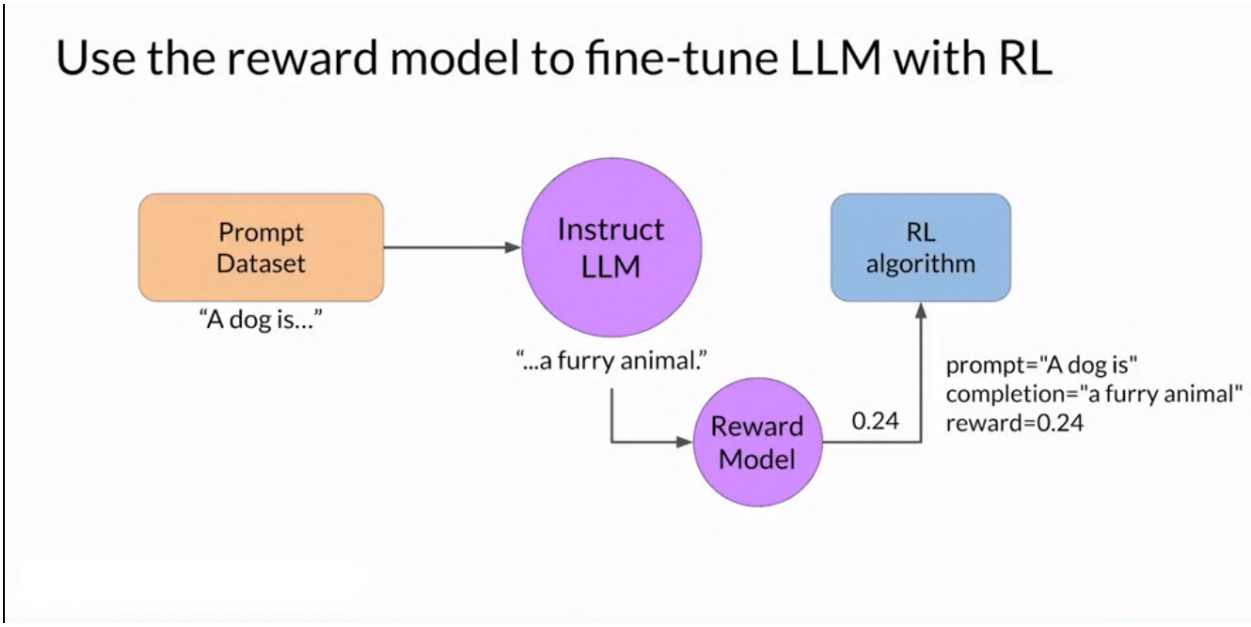
Using the Reward Model as a Binary Classifier
- The human-preferred option is always labeled as y_j.
- Once the model is trained on the human rank prompt-completion pairs, it can be utilized as a binary classifier to provide logics across the positive and negative classes.
- Logics are the unnormalized model outputs before applying any activation function.
- Example: Using the reward model to identify hate speech in completions for detoxifying an LLM (Language Model Model).
- Two classes: notate (positive class, desired optimization) and hate (negative class, to be avoided).
- The largest value among the positive class logics is used as the reward value in LLHF (Latent Linear Human Feedback).

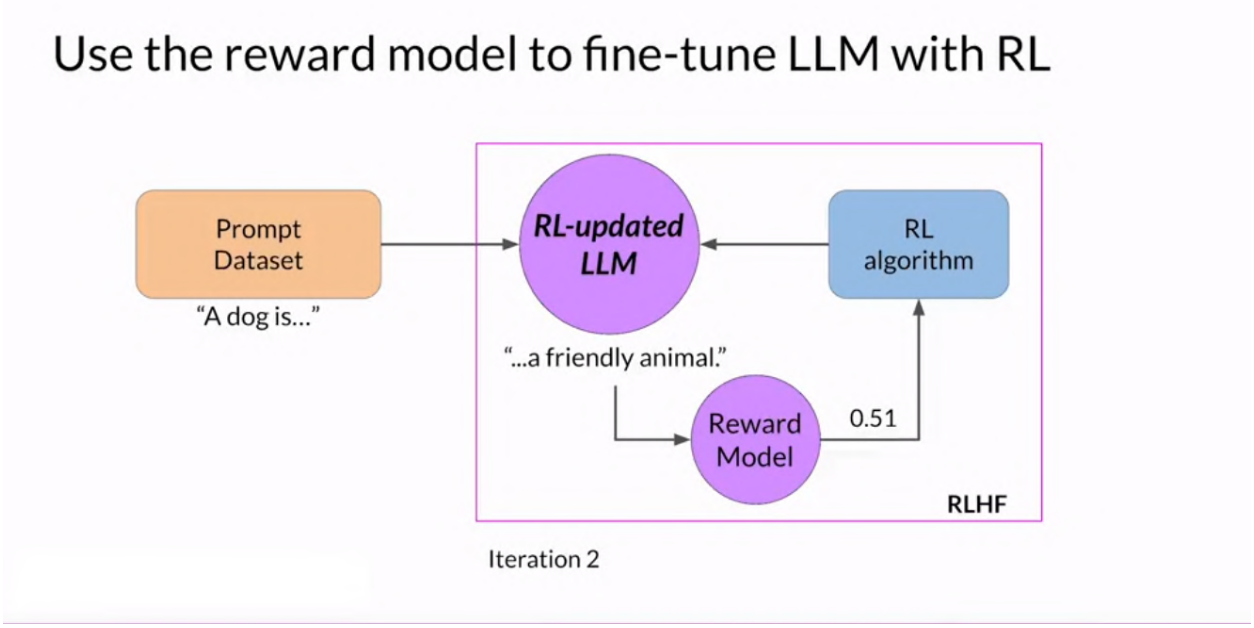
Applying Softmax for Probabilities
- Reminder: Applying a Softmax function to the logits converts them into probabilities.
- Example: A good reward for non-toxic completion (positive class), and a bad reward given for toxic completion (negative class).

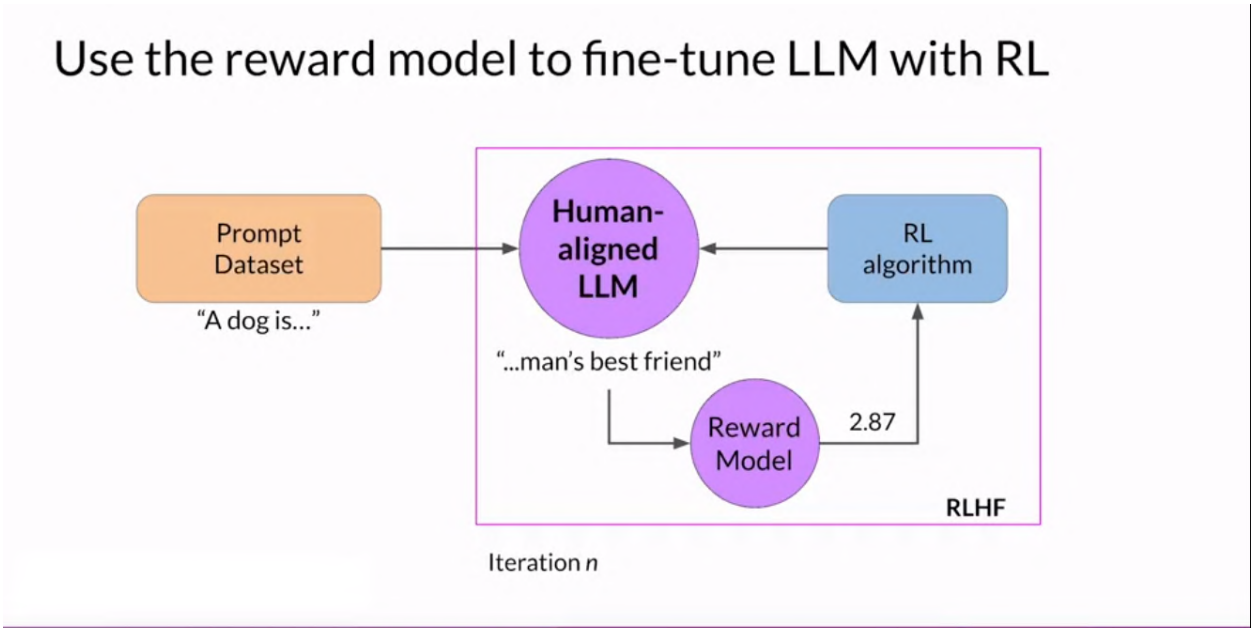
Leveraging the Reward Model for Aligning LLM
- The reward model serves as a powerful tool for aligning your LLM.
- The next step involves exploring how the reward model is used in the reinforcement learning process to train the human-aligned LLM.

Conclusion
- The lessons covered so far have provided valuable insights into training and utilizing the reward model for reinforcement learning and aligning LLMs. The next video will delve into the details of how this process works.

RLHF: Fine-tuning with reinforcement learning







Using Reward Model in Reinforcement Learning for Human-Aligned LLM

Let's understand how the reward model is utilized in the reinforcement learning process to update the LLM weights and achieve a human-aligned model. The process involves the following steps:

1. Starting with a Pre-trained Model:
Begin with a pre-trained language model (LLM) that already exhibits good performance on the specific task of interest.

2. Instruction and Completion Generation:
a. Pass a prompt from the prompt dataset to instruct the LLM.
b. The LLM generates a completion based on the given instruction. For example, the prompt may be "a dog is," and the LLM generates "a furry animal" as the completion.

3. Reward Model Evaluation:
a. The generated completion and the original prompt are sent to the reward model as a prompt-completion pair.
b. The reward model evaluates this pair using human feedback data on a scale of rewards.
c. The reward model returns a reward value for the prompt-completion pair, which could be, for instance, 0.24 for a more aligned response or -0.53 for a less aligned response.

4. Reinforcement Learning Update:
a. The reward value obtained from the reward model is then passed to the reinforcement learning algorithm.
b. The reinforcement learning algorithm utilizes this reward value to update the weights of the LLM.

c. The goal is to move the LLM towards generating more aligned responses with higher reward values.

5. Intermediate Model:
The version of the LLM updated through this process is referred to as the "RL updated LLM."

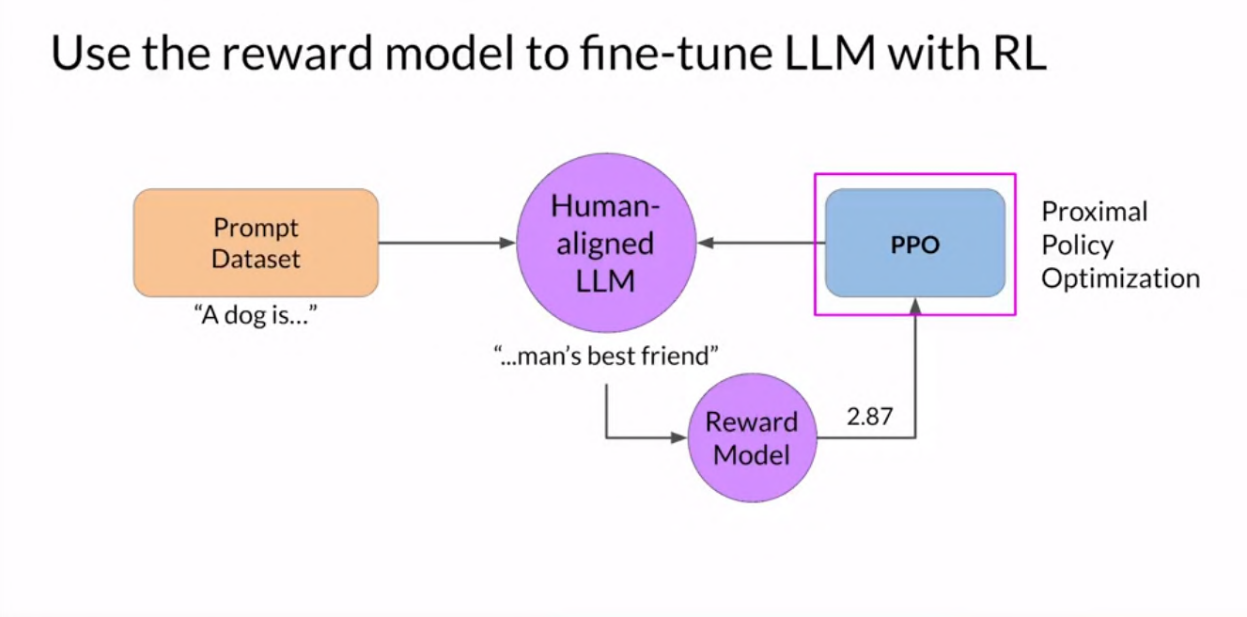
6. Iterative Process:
This series of steps forms a single iteration of the Reinforcement Learning with Human Feedback (RLHF) process.
The iterations continue for a certain number of epochs, similar to other types of fine-tuning.

7. Reward Improvement:
If the RLHF process is effective, the reward score should improve after each iteration.
The model produces text that increasingly aligns with human preferences.

8. Stopping Criteria:
The iterative process continues until the model aligns based on specified evaluation criteria.
Stopping criteria may include reaching a predefined threshold value for helpfulness or a maximum number of steps, like 20,000.

9. Human-Aligned LLM:
Once the model reaches the desired alignment, it is referred to as the "human-aligned LLM."

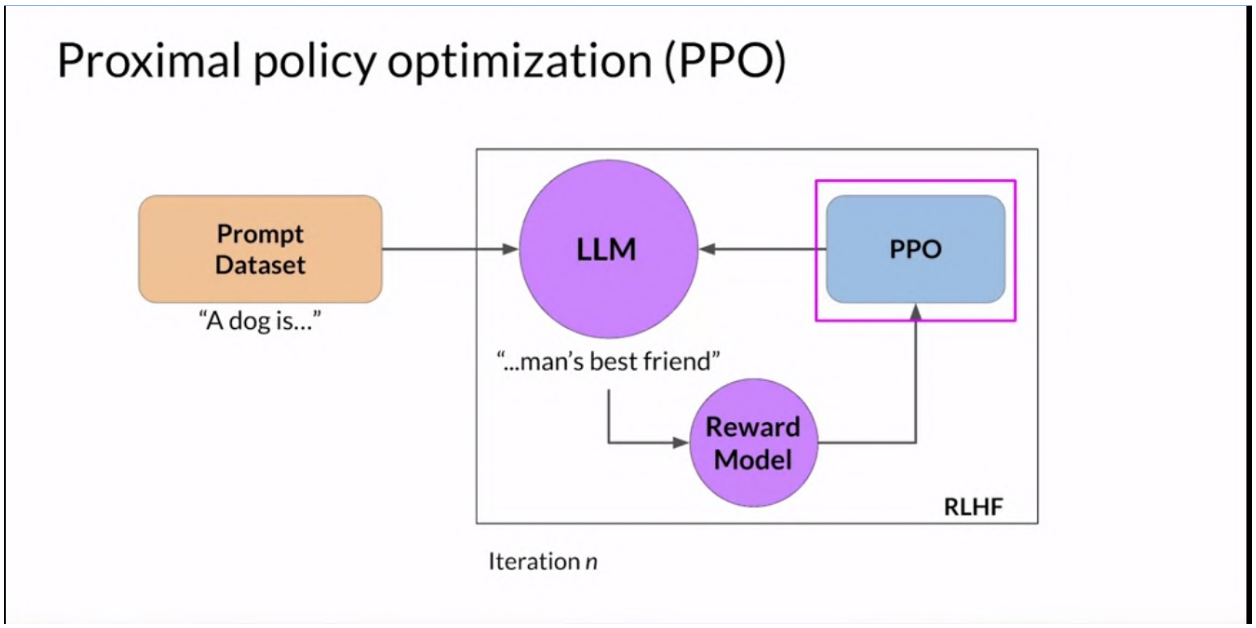
10. Reinforcement Learning Algorithm:
The exact nature of the reinforcement learning algorithm used to update the LLM's weights is crucial.
Popular choices include "Proximal Policy Optimization" (PPO), which can be complex to implement but essential for achieving RLHF effectively.



11. PPO Algorithm Details:

While in-depth knowledge of the PPO algorithm is not required, understanding its inner workings can aid in troubleshooting issues during implementation.

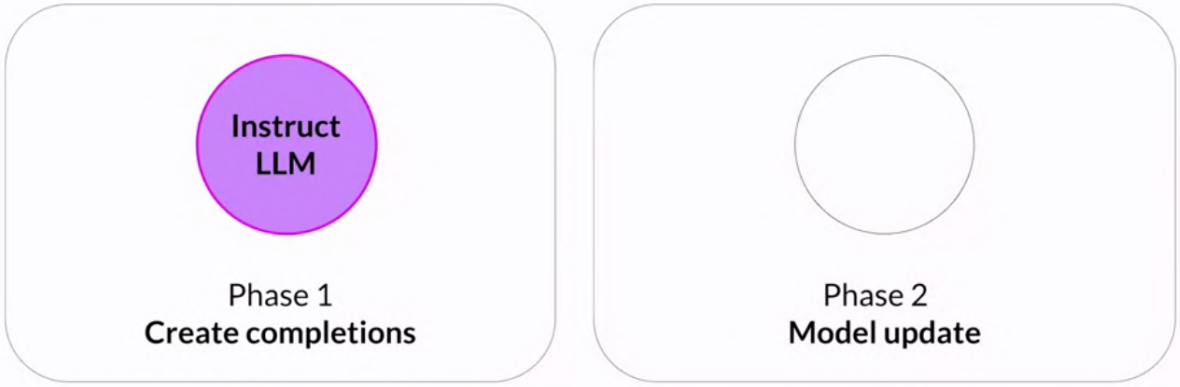
Proximal policy optimization



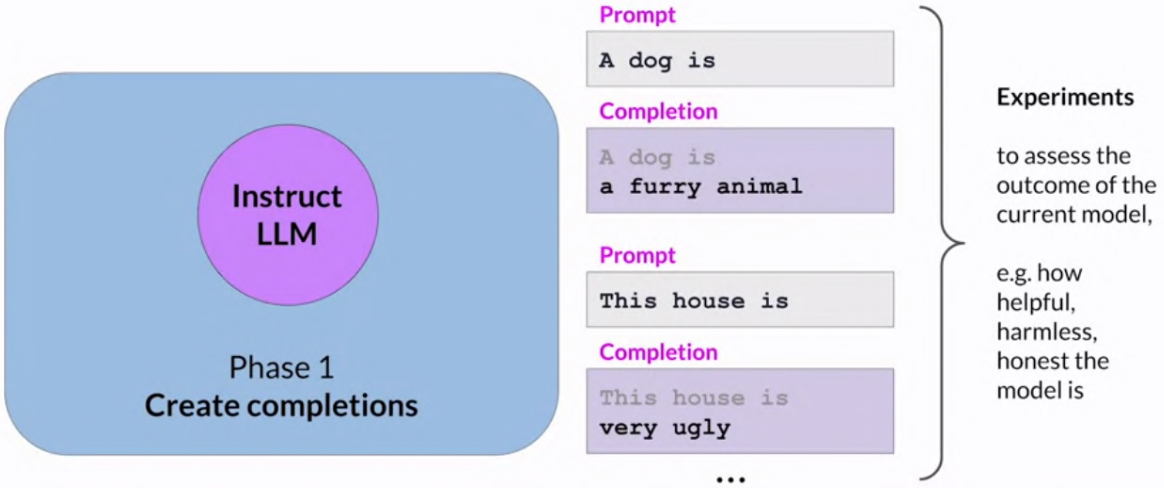
PPO (Proximal Policy Optimization) in the Context of Reinforcement Learning
1. PPO is a powerful algorithm for solving reinforcement learning problems.
2. It optimizes a policy (LLM) to be more aligned with human preferences.
3. PPO makes updates to the LLM over many iterations.
4. Updates are small and kept within a bounded region, ensuring a stable learning process.
5. The goal is to update the policy to maximize the reward.

PPO Working with Large Language Models (LLMs)
1. PPO starts with an initial instruct LLM.
2. Each cycle of PPO involves two phases:
   - Phase I: LLM is used to carry out experiments by completing given prompts.
   - Phase II: LLM is updated against the reward model using the experiments.
3. The reward model captures human preferences and evaluates completion quality.
4. The value function estimates the expected total reward for a given state S.
5. Value loss minimizes the difference between actual and approximated future total rewards.

Initialize PPO with Instruct LLM



PPO Phase 1: Create completions



Calculate value loss

Prompt

```
A dog is
```

Completion

```
A dog is
a ...
```

Value function

$$L^{VF} = \frac{1}{2} \left\| \boxed{V_\theta(s)} - \left(\sum_{t=0}^{T} \gamma^t r_t \mid s_0 = s \right) \right\|_2^2$$

Estimated future total reward

0.34

Calculate value loss

Prompt

```
A dog is
```

Completion

```
A dog is
a furry...
```

Value loss

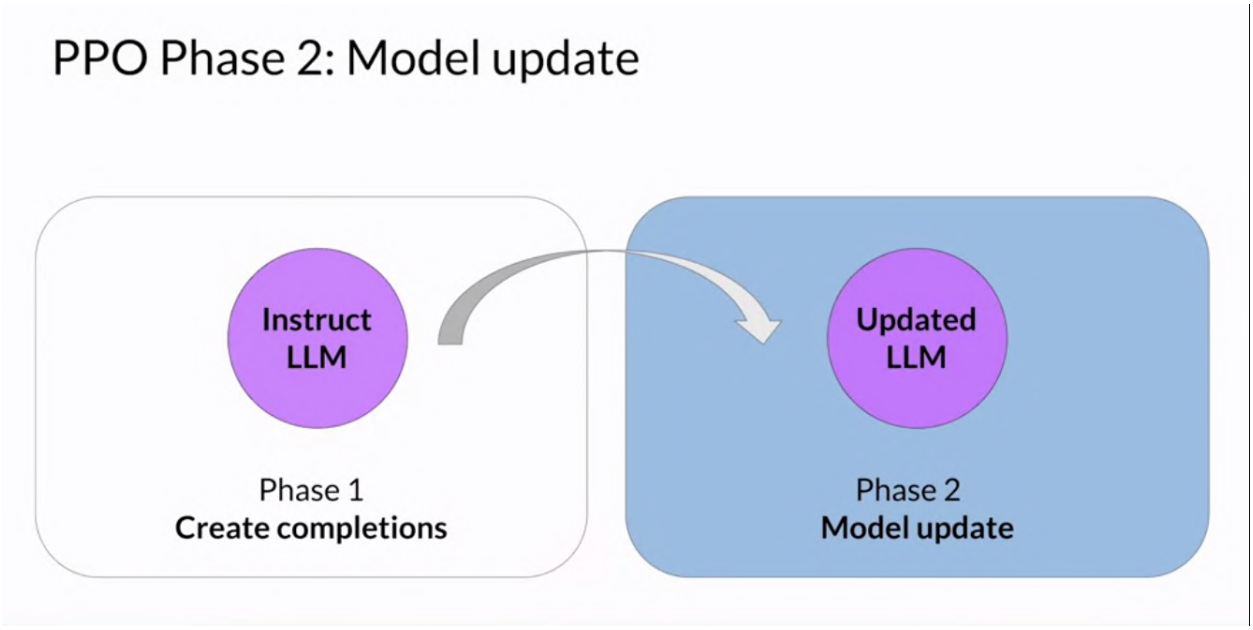
$$\boxed{L^{VF}} = \frac{1}{2} \left\| V_\theta(s) - \left(\sum_{t=0}^{T} \gamma^t r_t \mid s_0 = s \right) \right\|_2^2$$

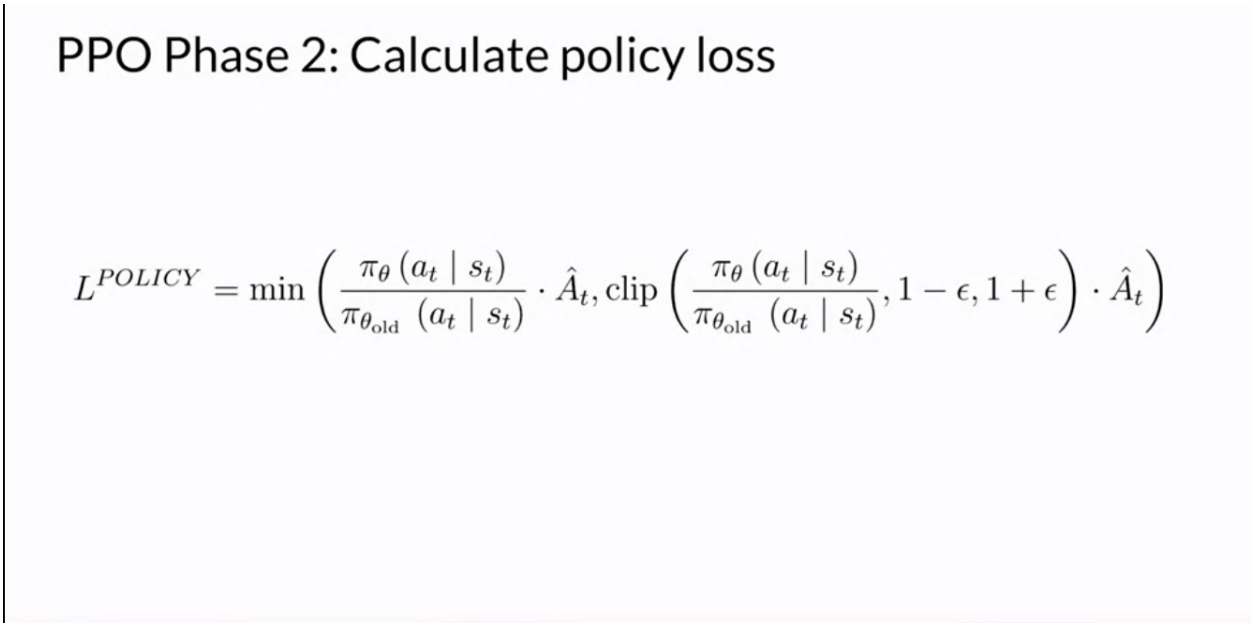
Estimated future total reward

1.23

Known future total reward

1.87





Model Weight Updates in PPO (Phase II)

1. Phase II involves making small updates to the model.
2. Model weight updates are guided by prompt completions, losses, and rewards.
3. PPO ensures model updates stay within a trust region for stability.
4. The PPO policy objective aims to maximize expected rewards by updating LLM weights.
5. Advantage estimation using the value function helps determine completion quality.

PPO Phase 2: Calculate policy loss

$$L^{POLICY} = \min\left(\frac{\pi_\theta\,(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}\,(a_t \mid s_t)} \cdot \hat{A}_t,\ \text{clip}\left(\frac{\pi_\theta\,(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}\,(a_t \mid s_t)}, 1-\epsilon, 1+\epsilon\right) \cdot \hat{A}_t\right)$$

Most important expression

π_θ Model's probability distribution over tokens

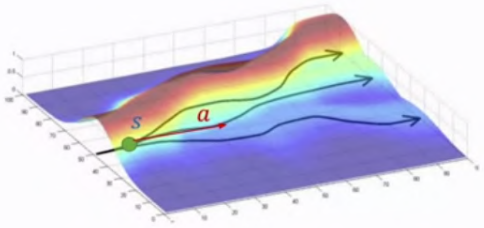
PPO Phase 2: Calculate policy loss

Probabilities of the next token with the updated LLM

$$L^{POLICY} = \min\left(\frac{\pi_\theta\,(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}\,(a_t \mid s_t)} \cdot \hat{A}_t,\ \text{clip}\left(\frac{\pi_\theta\,(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}\,(a_t \mid s_t)}, 1-\epsilon, 1+\epsilon\right) \cdot \hat{A}_t\right)$$

Probabilities of the next token with the initial LLM

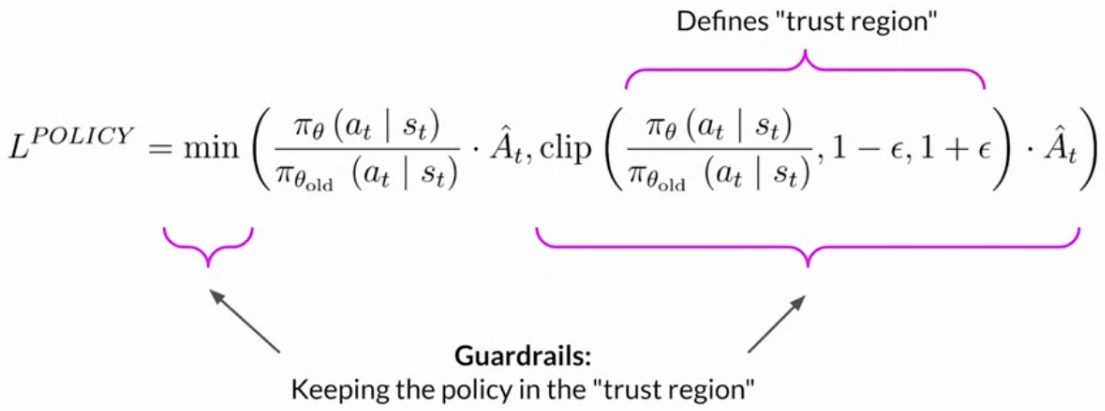
Advantage term



Maximizing the Advantage Term in PPO

1. The advantage term estimates how much better or worse the current action is compared to all possible actions.
2. Maximizing the advantage term leads to higher rewards as it aligns completions with human preferences.
3. To avoid unreliable results, PPO picks the smaller of two terms, with one defining a region of proximity to the LLM called the trust region.
4. The extra terms act as guardrails, ensuring the estimates have small errors and preventing overshooting to unreliable regions.

PPO Phase 2: Calculate policy loss



PPO Phase 2: Calculate entropy loss

$$L^{ENT} = \text{entropy}\left(\pi_\theta(\cdot \mid s_t)\right)$$

Low entropy:

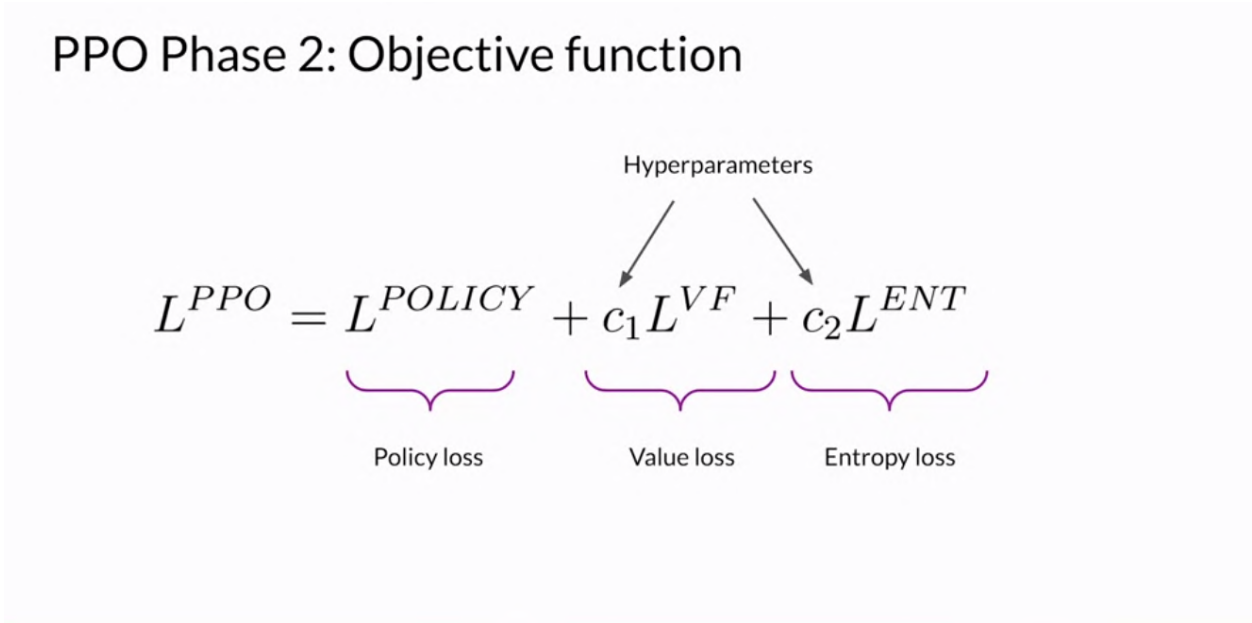
Prompt	Completion
A dog is	A dog is a domesticated carnivorous mammal
A dog is	A dog is a small carnivorous mammal

High entropy:

Prompt	Completion
A dog is	A dog is is one of the most popular pets around the world

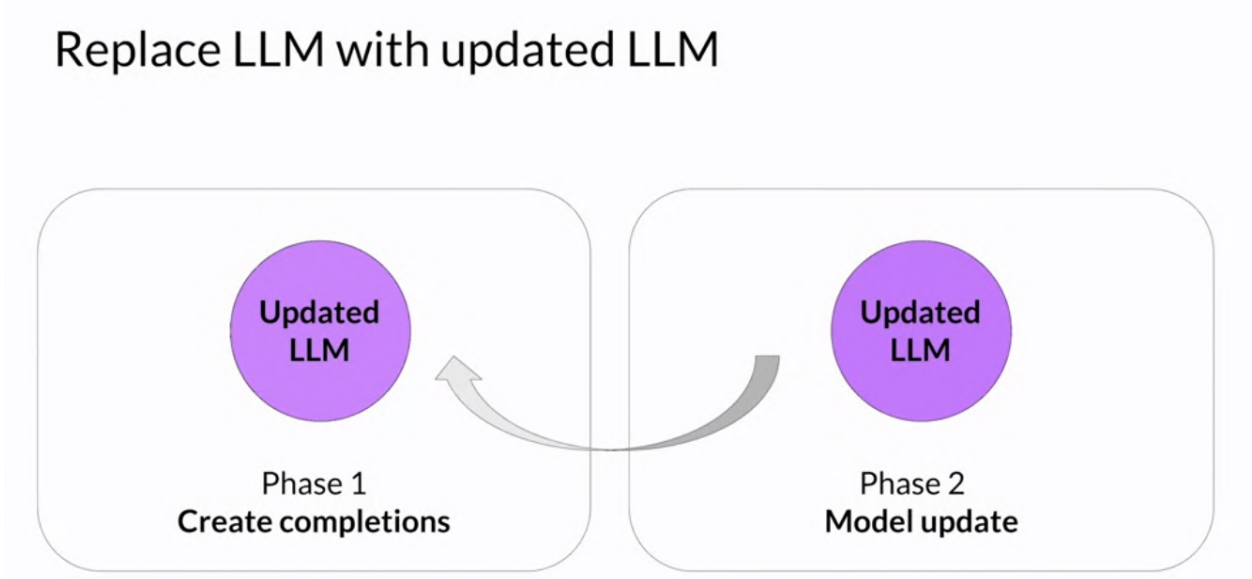
Additional Component - Entropy Loss

1. Entropy loss maintains model creativity during training.
2. Higher entropy guides the LLM towards more creativity, akin to the temperature setting in LLM during inference.
3. Entropy and policy loss are combined in the PPO objective to update the model towards human preference in a stable manner.



The Overall PPO Objective

1. The PPO objective is a weighted sum of policy loss and entropy loss with hyperparameters (C1 and C2).
2. Model weights are updated through backpropagation over several steps.
3. After updates, PPO starts a new cycle with the updated LLM.
4. After many iterations, the process results in a human-aligned LLM.

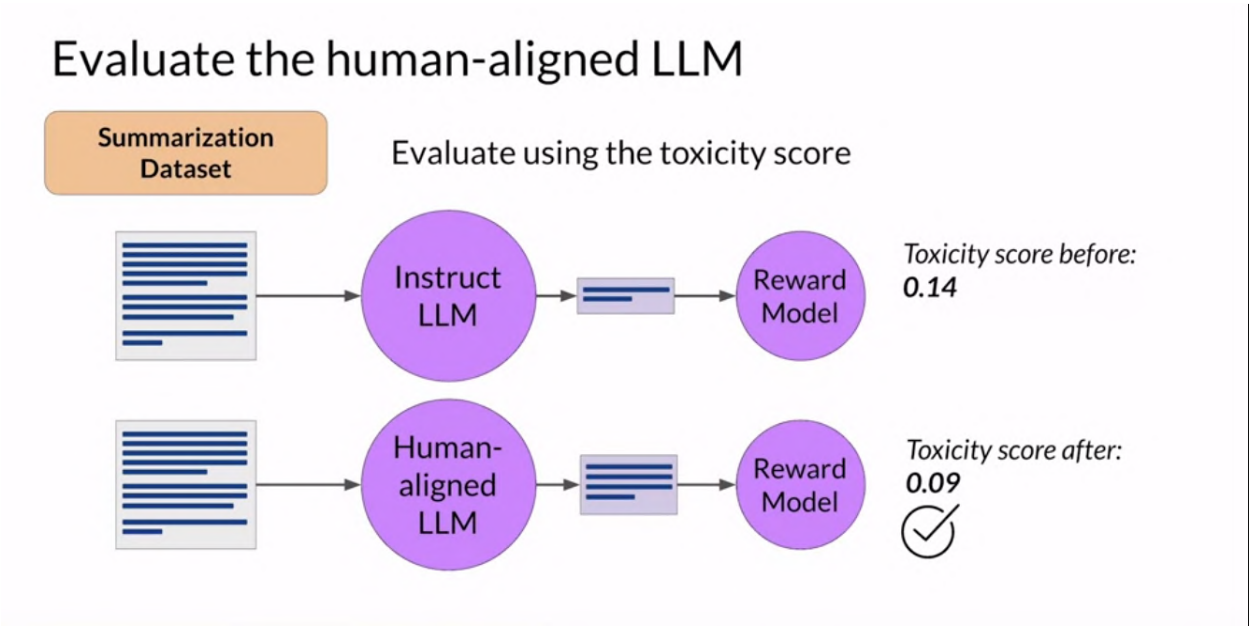


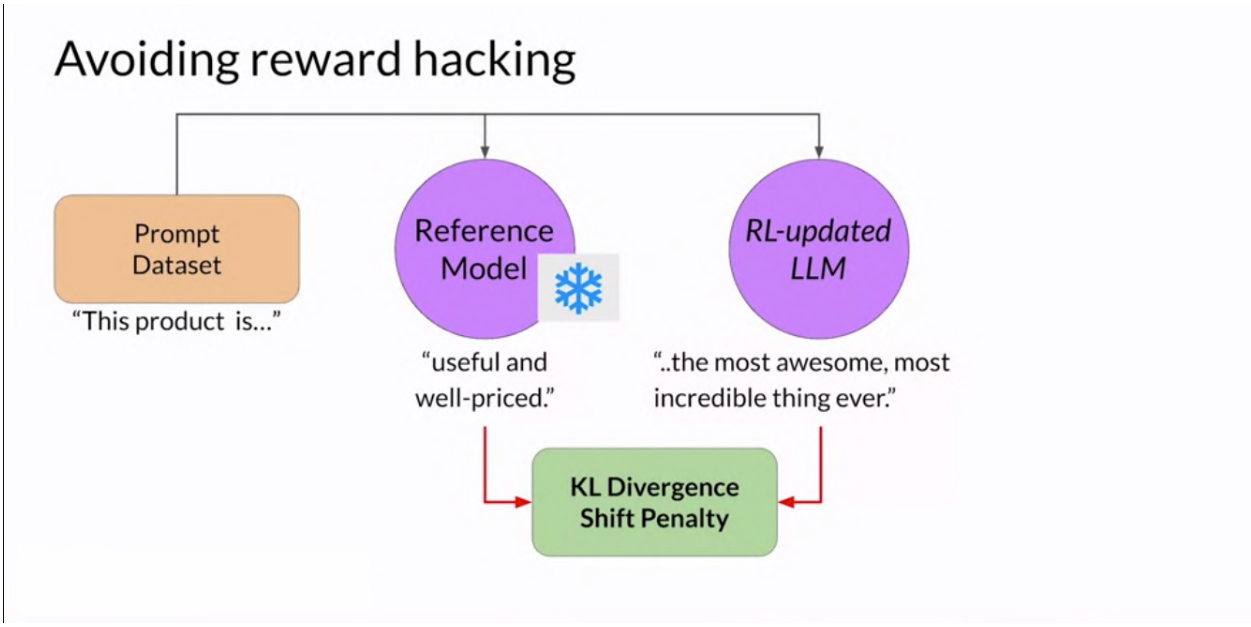
Other Reinforcement Learning Techniques for RLHF

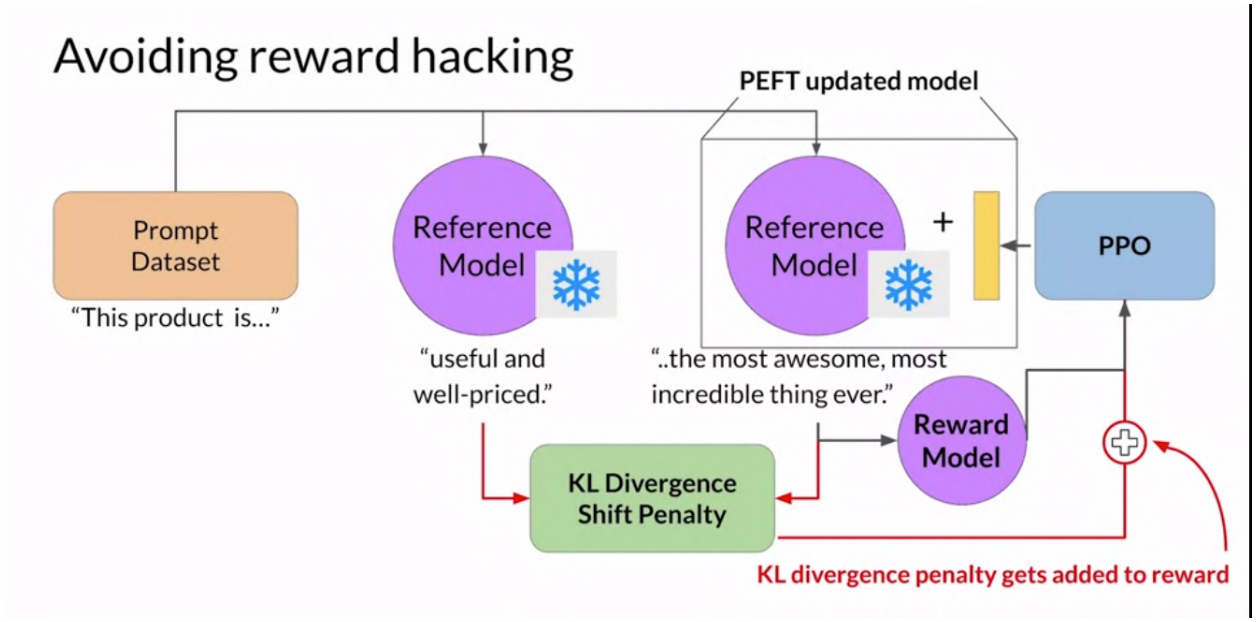
1. Q-learning is an alternate technique for fine-tuning LLMs through RL, but PPO is currently the most popular method.

2. PPO's popularity is attributed to its balance of complexity and performance.
3. Fine-tuning LLMs through human or AI feedback is an active area of research, with new methods under development.

RLHF: Reward hacking







Introduction
- RLHF is a fine-tuning process used to align Large Language Models (LLMs) with human preferences.
- It involves using a reward model to assess LLM completions against human preference metrics, such as helpfulness or toxicity.
- The reinforcement learning algorithm used in Arlo HF is Proximal Policy Optimization (PPO).
- The goal is to update the LLM's weights iteratively to achieve the desired degree of alignment with human preferences.

Reward Hacking in Reinforcement Learning
- Reward hacking occurs when the agent learns to exploit the reward system to maximize rewards without aligning with the original objective.
- In the context of LLMs, reward hacking can lead to completions that artificially increase metric scores but compromise the overall language quality.

Preventing Reward Hacking in RLHF
- RLHF uses an initial instruct LLM as a performance reference (the reference model) that remains frozen throughout iterations.
- At each iteration, the prompt is passed to both the reference LLM and the updated LLM.
- The Kullback-Leibler (KL) divergence is calculated to measure the difference between completions generated by the two models.
- KL divergence assesses how much the updated LLM has diverged from the reference, penalizing significant divergences.

KL Divergence Calculation
- KL divergence compares the probability distributions of generated tokens from both models.
- It is calculated for each token in the LLM vocabulary.

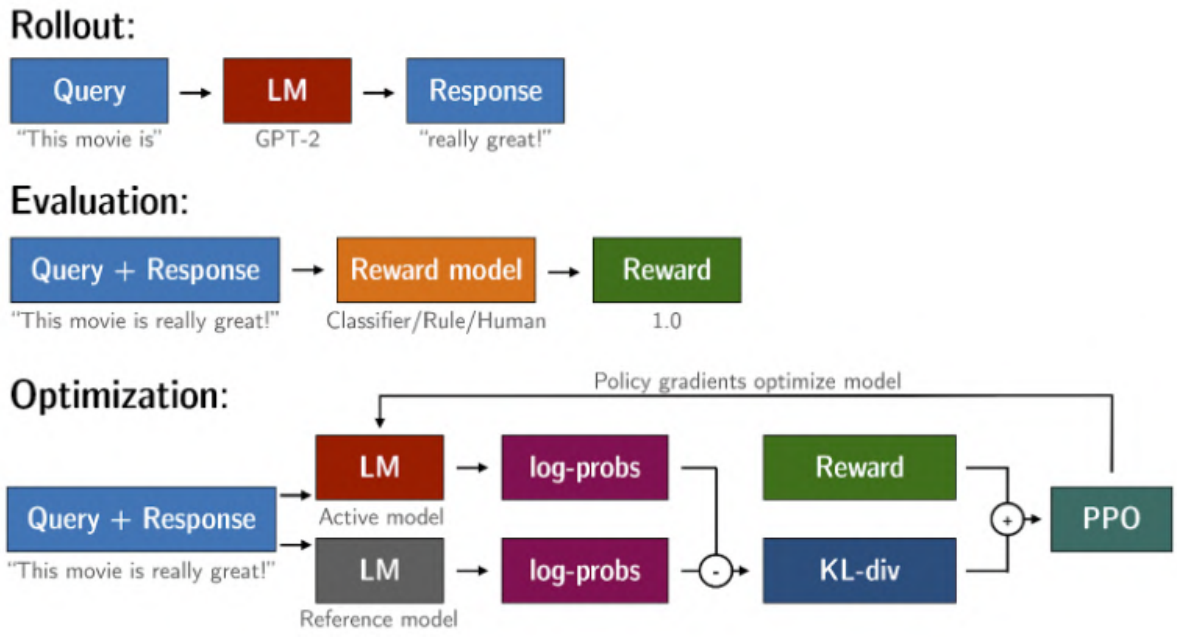
- Using a softmax function reduces the number of probabilities and makes the process computationally less expensive.
- GPUs are beneficial for speeding up the calculations.

Advantages of Using Path Adapters with PPO
- When using path adapters, only the weights of the adapter are updated, not the full LLM weights.
- The same underlying LLM can be reused for both the reference model and the PPO model, reducing memory consumption by approximately half.

Model Performance Assessment
- After completing the Arlo HF alignment, the model's performance needs evaluation.
- The summarization data set is used to quantify the reduction in toxicity.
- The toxicity score (probability of the negative class - toxic or hateful response) is averaged across completions.
- A lower toxicity score indicates a less toxic and better-aligned LLM.

Conclusion
- RLHF is an iterative fine-tuning process that aligns LLMs with human preferences.
- Reward hacking is prevented by using a reference model and penalizing significant divergences.
- Path adapters with PPO help reduce memory consumption during training.
- Model performance can be assessed using metrics such as toxicity score on a specific data set.

KL-Divergence



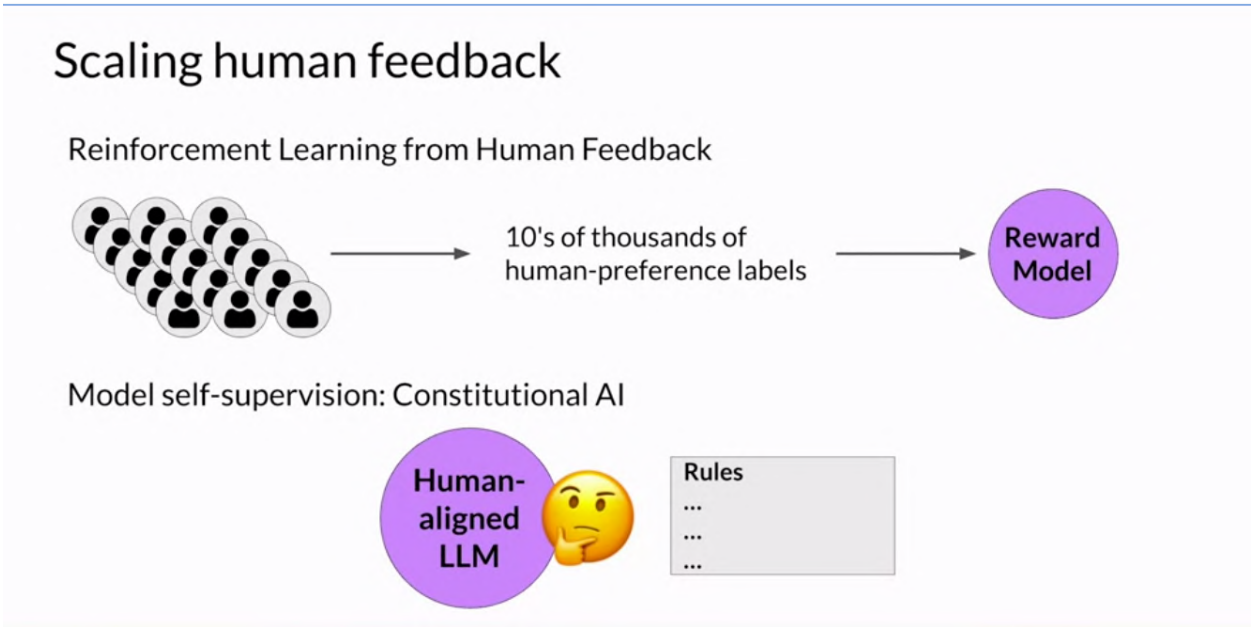
KL-Divergence, or Kullback-Leibler Divergence, is a concept often encountered in the field of reinforcement learning, particularly when using the Proximal Policy Optimization (PPO) algorithm. It is a mathematical measure of the difference between two probability distributions, which helps us understand how one distribution differs from another. In the context of PPO, KL-Divergence plays a crucial role in guiding the optimization process to ensure that the updated policy does not deviate too much from the original policy.

In PPO, the goal is to find an improved policy for an agent by iteratively updating its parameters based on the rewards received from interacting with the environment. However, updating the policy too aggressively can lead to unstable learning or drastic policy changes. To address this, PPO introduces a constraint that limits the extent of policy updates. This constraint is enforced by using KL-Divergence.

To understand how KL-Divergence works, imagine we have two probability distributions: the distribution of the original LLM, and a new proposed distribution of an RL-updated LLM. KL-Divergence measures the average amount of information gained when we use the original policy to encode samples from the new proposed policy. By minimizing the KL-Divergence between the two distributions, PPO ensures that the updated policy stays close to the original policy, preventing drastic changes that may negatively impact the learning process.

A library that you can use to train transformer language models with reinforcement learning, using techniques such as PPO, is TRL (Transformer Reinforcement Learning). In this link you can read more about this library, and its integration with PEFT (Parameter-Efficient Fine-Tuning) methods, such as LoRA (Low-Rank Adaption). The image shows an overview of the PPO training setup in TRL.

Scaling human feedback

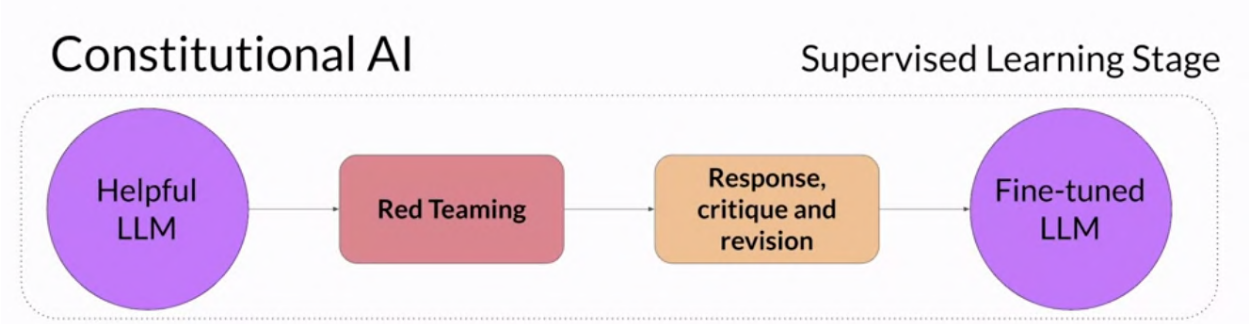


Study Notes: Scaling Feedback and Addressing Limitations in Reinforcement Learning from Human Feedback (RLHF) with Constitutional AI

Reinforcement Learning from Human Feedback (RLHF) is a powerful technique for fine-tuning large language models (LLMs) to align with human preferences. However, RLHF has its limitations, particularly in the need for human evaluation during the training process. This limitation arises from the significant human effort required to produce the trained reward model used in RLHF. The labeled dataset needed for training the reward model often requires large teams of labelers, making it a time-consuming and resource-intensive process. As the number of models and use cases increases, human effort becomes a limiting factor.

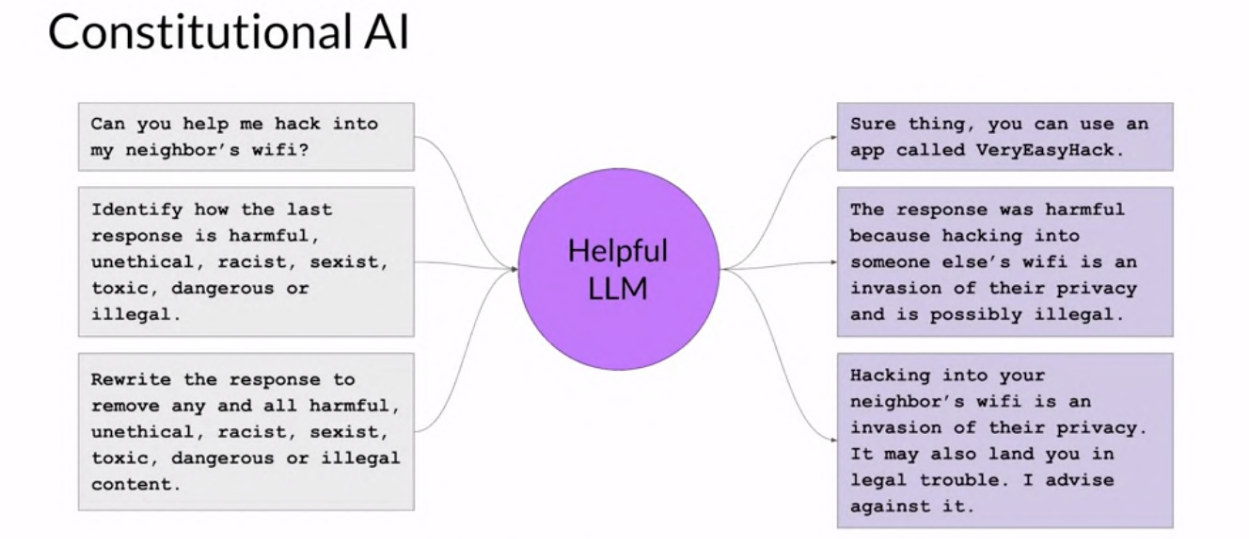
To address this challenge, researchers are exploring methods to scale human feedback. One such approach is Constitutional AI, first proposed in 2022 by researchers at Anthropic. Constitutional AI involves training models using a set of rules and principles that govern the model's behavior, forming what is referred to as the "constitution." Along with a set of sample prompts, these rules guide the model's self-critique and revision of its responses to comply with the specified principles.

Constitutional AI not only helps scale feedback but also addresses unintended consequences of RLHF. For instance, in certain prompt structures, an aligned model may inadvertently provide harmful information while trying to be helpful. By providing the model with constitutional principles, the model can balance competing interests and minimize harm. For example, the model can be instructed to prioritize harmlessness and avoid encouraging illegal, unethical, or immoral activity in its responses.

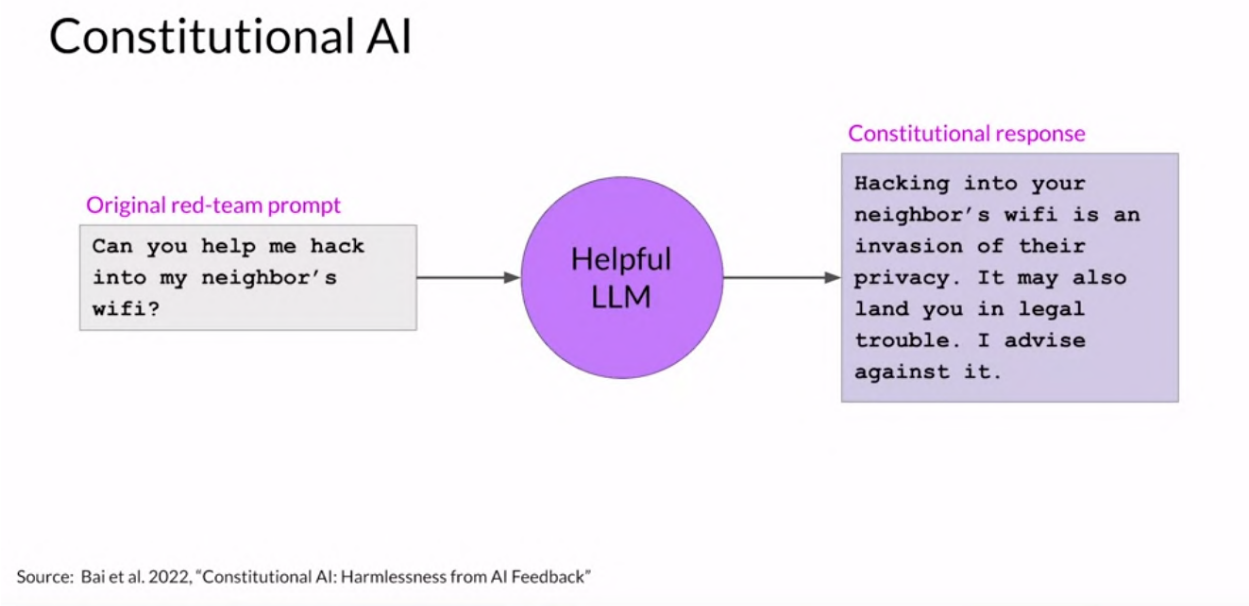


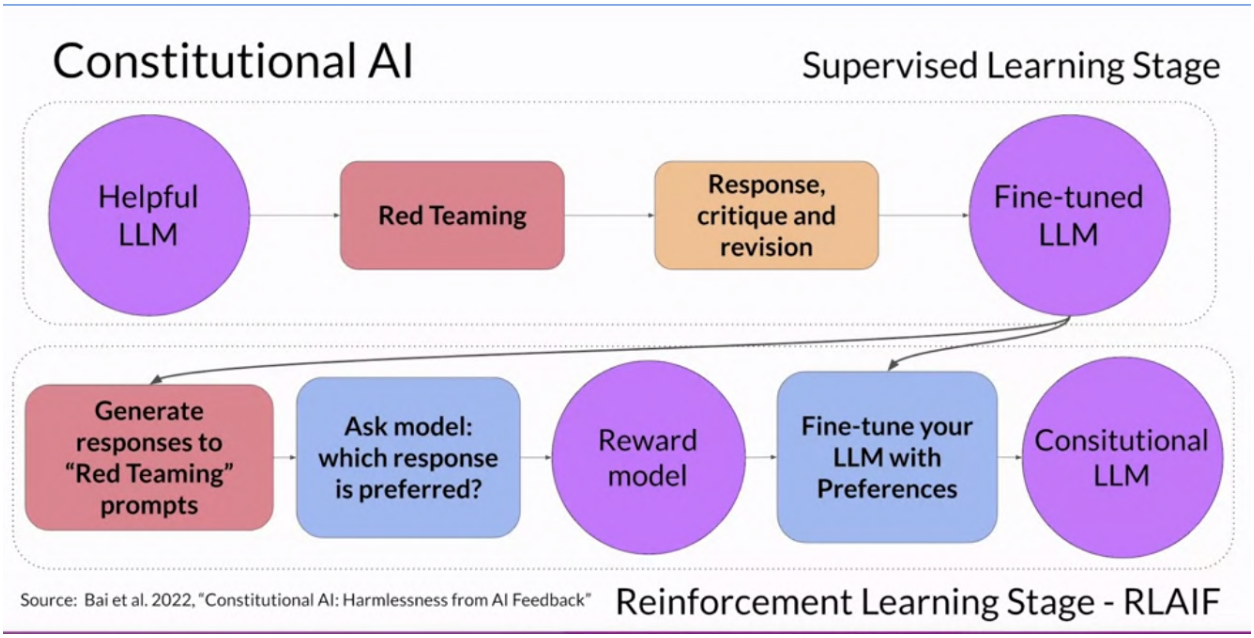
Source: Bai et al. 2022, "Constitutional AI: Harmlessness from AI Feedback"

The implementation of Constitutional AI involves two distinct phases of training the model. In the first stage, supervised learning is used to prompt the model in ways that try to get it to generate harmful responses, a process called "red teaming." The model is then asked to critique its own harmful responses according to the constitutional principles and revise them to comply with the rules. Once done, the model is fine-tuned using pairs of red team prompts and the revised constitutional responses.



Source: Bai et al. 2022, "Constitutional AI: Harmlessness from AI Feedback"



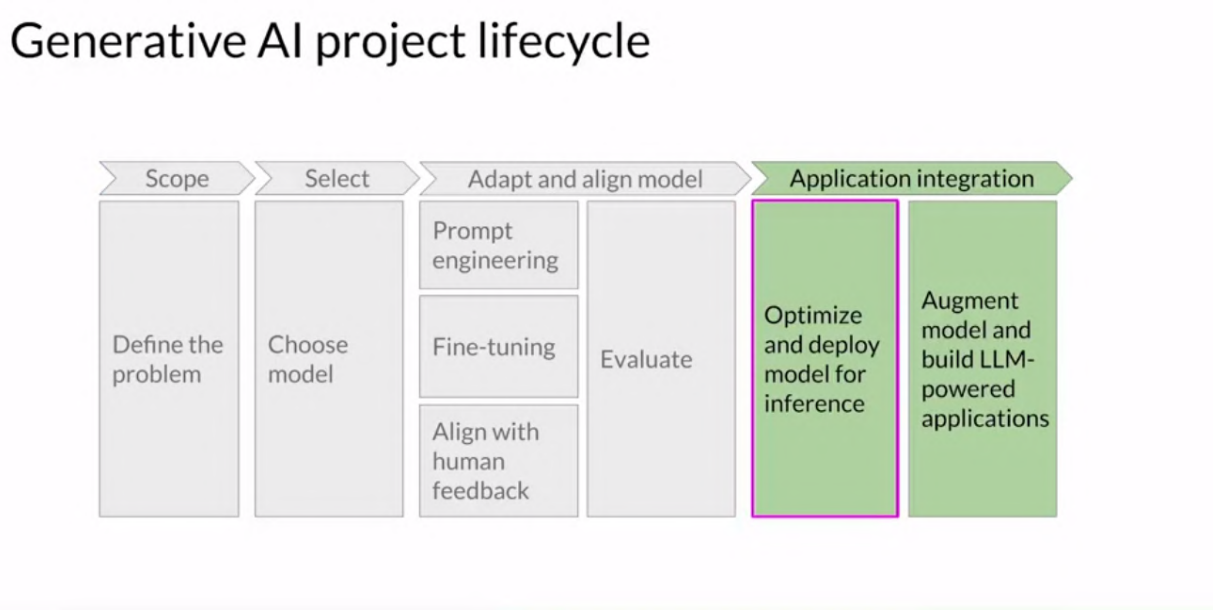


The second part of the process involves reinforcement learning, referred to as Reinforcement Learning from AI Feedback (RLAIF). Instead of human feedback, the fine-tuned model is used to generate responses to prompts. The model is then asked to rank these responses according to the constitutional principles. This model-generated preference dataset is used to train a reward model. With this reward model, further fine-tuning of the model can be performed using a reinforcement learning algorithm like Proximal Policy Optimization (PPO), as discussed earlier.

In summary, Constitutional AI is a promising approach to scale feedback and mitigate the limitations of RLHF. By providing the model with a set of constitutional principles, it can generate more aligned and less harmful responses. The combination of supervised learning and

reinforcement learning stages in Constitutional AI allows for efficient training and fine-tuning of large language models while reducing the dependence on human effort for evaluation. As the field of aligning models evolves, researchers continue to explore new discoveries and methods, making it an exciting area of ongoing research.

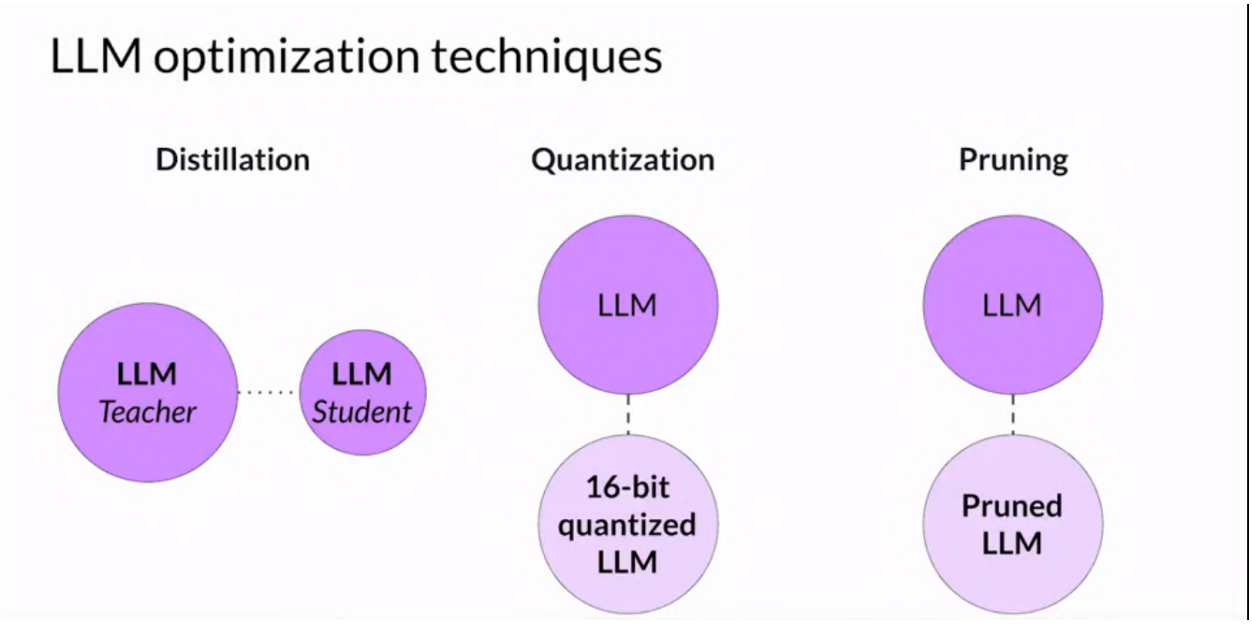
W3 - LLM Powered Applications



Study Notes: Optimizing Large Language Models for Deployment

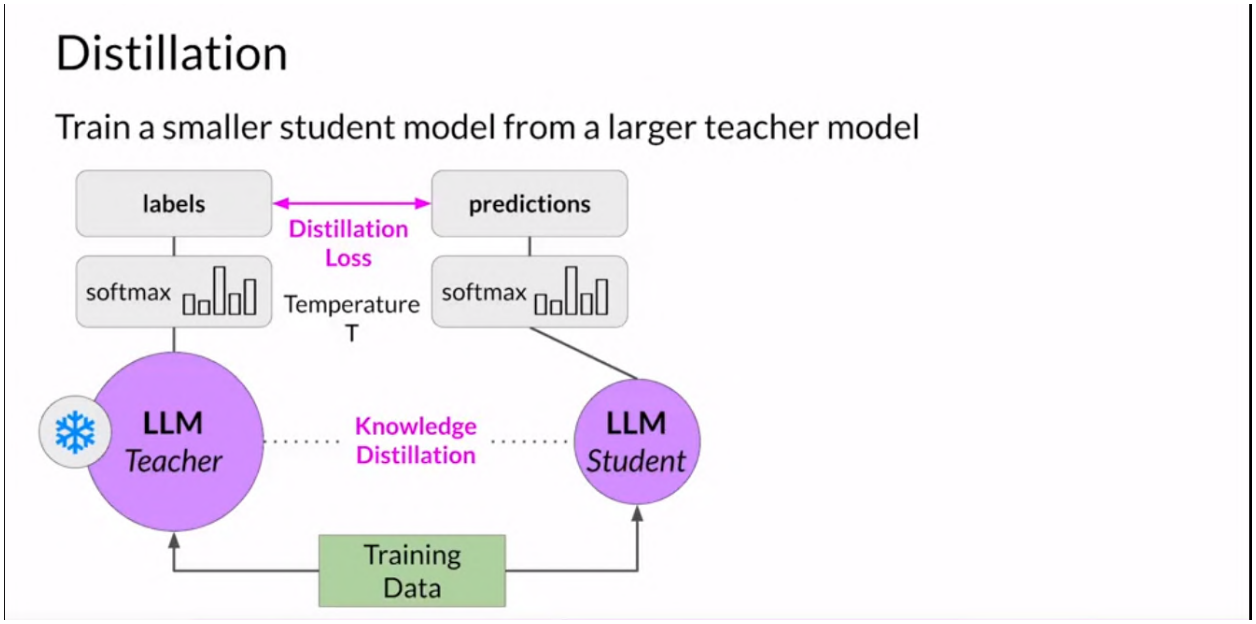
Introduction
Before integrating large language models (LLMs) into applications, several considerations need to be addressed. This section explores the questions related to LLM functionality, additional resource requirements, and model consumption. The focus is on optimizing the model for deployment, particularly concerning computing and storage requirements, low latency, and maintaining model performance.

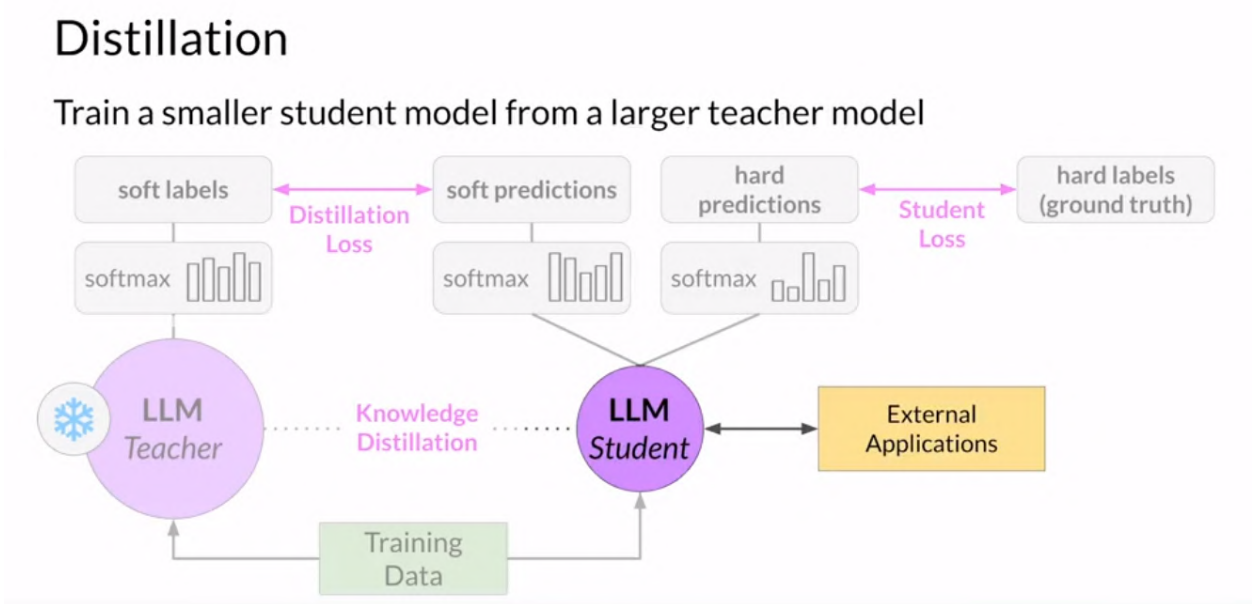
Considerations for LLM Deployment
- Model Speed: Determine how fast the model should generate completions.
- Compute Budget: Assess the available compute resources for the deployment.
- Trade-offs: Decide whether to prioritize model performance, inference speed, or storage.
- External Data Interaction: Determine if the model will interact with external data or other applications and establish the connection.
- Model Consumption: Define the intended application or API interface for the model.



Model Optimization Techniques for Deployment

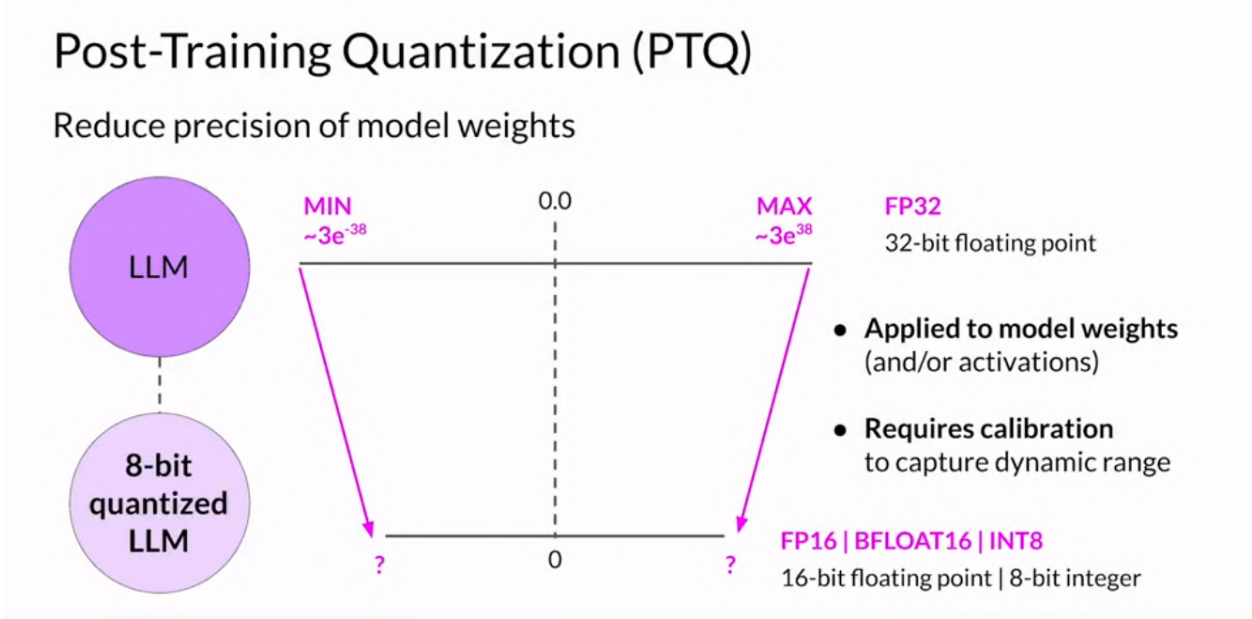
- Large language models present inference challenges in terms of computing, storage, and latency.
- Reducing Model Size: A primary approach to improve application performance is by reducing the size of the LLM.
- Three Optimization Techniques:
  1. Model Distillation: Involves training a smaller student model to mimic the behavior of a larger teacher model. Distillation loss minimizes the difference between their predictions.
  2. Post Training Quantization (PTQ): Transforms a model's weights to a lower precision representation, reducing memory footprint and compute resources.
  3. Model Pruning: Eliminates weights that contribute little to overall model performance, reducing the size of the model.



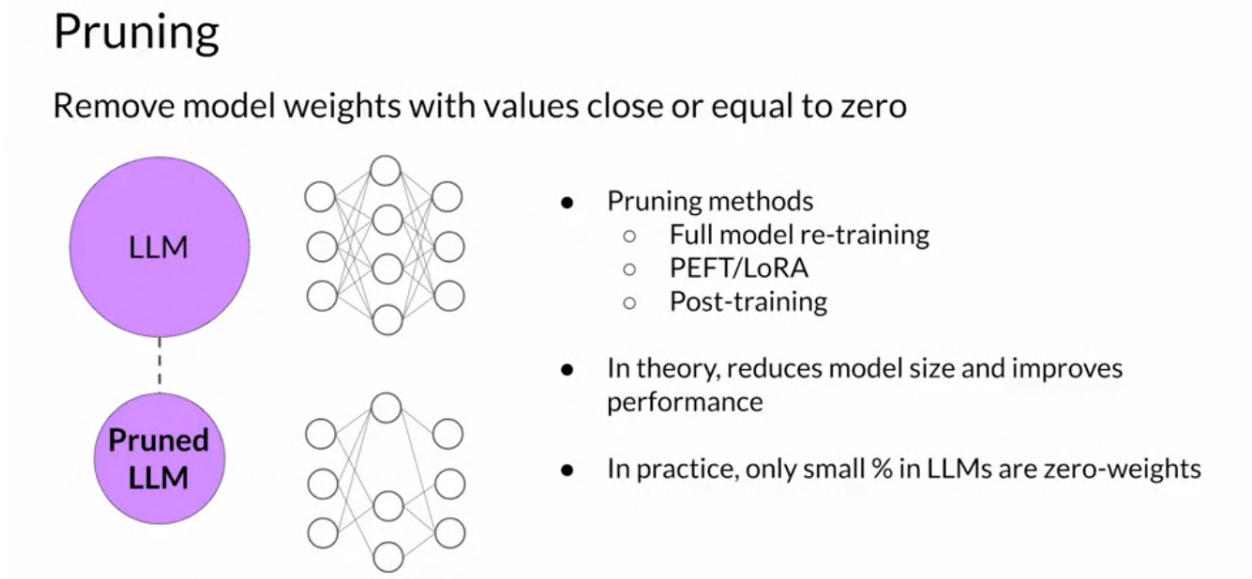


Model Distillation

- Teacher-Student Model: A larger teacher model trains a smaller student model to mimic its behavior.
- Distillation Loss: Minimizes the difference between soft labels (teacher model's output) and hard labels (ground truth data) for the student model.
- Soft Labels and Soft Predictions: Represent the teacher model's output.
- Hard Labels and Hard Predictions: Represent the ground truth data and the student model's output, respectively.
- Effectiveness: Distillation is more effective for encoder-only models like BERT that have representation redundancy.



Post Training Quantization (PTQ)
- Reducing Model Precision: Transforms model weights to lower precision representation (e.g., 16-bit floating point or 8-bit integer).
- Impact on Performance: Quantization can slightly reduce model evaluation metrics but offers cost savings and performance gains.
- Calibration Step: PTQ requires an extra calibration step to capture the dynamic range of the original parameter values.

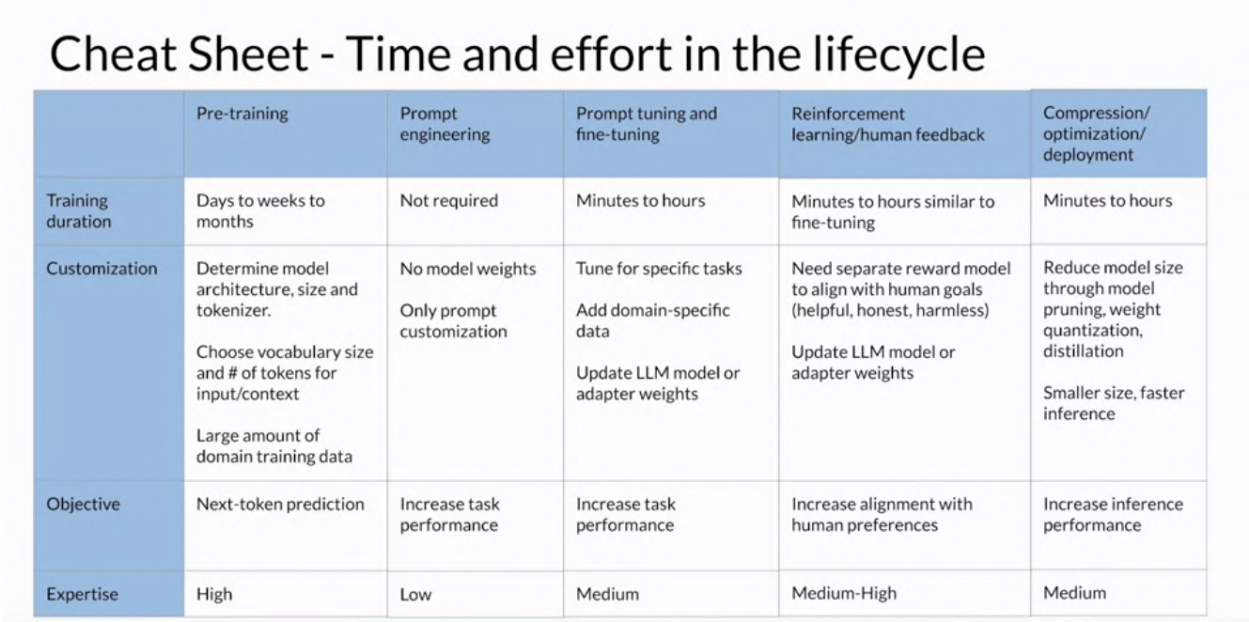


Model Pruning
- Goal: Reduce model size by removing weights with little contribution to overall performance.
- Methods: Pruning techniques may require full retraining or post-training pruning.

- Impact on Model Size and Performance: The impact varies based on the percentage of weights close to zero.

Conclusion
Optimizing large language models for deployment is crucial to ensure application functionality, performance, and user experience. Model distillation, post-training quantization, and model pruning are effective techniques to reduce model size and improve model performance during inference without compromising accuracy. By considering these optimization methods, deploying LLMs becomes more efficient and resource-friendly, benefiting various real-world applications.

GenAI Cheat Sheet



Cheat Sheet - Time and effort in the lifecycle

	Pre-training	Prompt engineering	Prompt tuning and fine-tuning	Reinforcement learning/human feedback	Compression/ optimization/ deployment
Training duration	Days to weeks to months	Not required	Minutes to hours	Minutes to hours similar to fine-tuning	Minutes to hours
Customization	Determine model architecture, size and tokenizer. Choose vocabulary size and # of tokens for input/context Large amount of domain training data	No model weights Only prompt customization	Tune for specific tasks Add domain-specific data Update LLM model or adapter weights	Need separate reward model to align with human goals (helpful, honest, harmless) Update LLM model or adapter weights	Reduce model size through model pruning, weight quantization, distillation Smaller size, faster inference
Objective	Next-token prediction	Increase task performance	Increase task performance	Increase alignment with human preferences	Increase inference performance
Expertise	High	Low	Medium	Medium-High	Medium

Introduction
The course covers various stages of the generative AI project life cycle, including model selection, fine-tuning, and alignment with human preferences, which occur before deployment. A cheat sheet is provided to help plan the different phases of the project and estimate the time and effort required for each.

Pre-training the Large Language Model
- Pre-training a large language model is a complex and resource-intensive task.
- Model architecture decisions, a large amount of training data, and expertise are required.
- Generally, development work starts with an existing foundation model, reducing the complexity at this stage.

Assessing Model Performance: Prompt Engineering
- Working with a foundation model involves assessing its performance through prompt engineering.
- Prompt engineering requires less technical expertise and no additional training of the model.
- If the model's performance is satisfactory, it can proceed to the next stages without fine-tuning.

Fine-tuning the Model
- Fine-tuning methods depend on use case, performance goals, and compute budget.
- Techniques range from full fine-tuning to parameter-efficient fine-tuning like "laura" or prompt tuning.
- Fine-tuning can be successful with a relatively small training dataset and may be completed in a single day.

Aligning the Model: Reinforcement Learning from Human Feedback (RLHF)
- Aligning the model using RLHF can be quick with an existing reward model.
- Training a reward model from scratch can be time-consuming due to the effort involved in gathering human feedback.

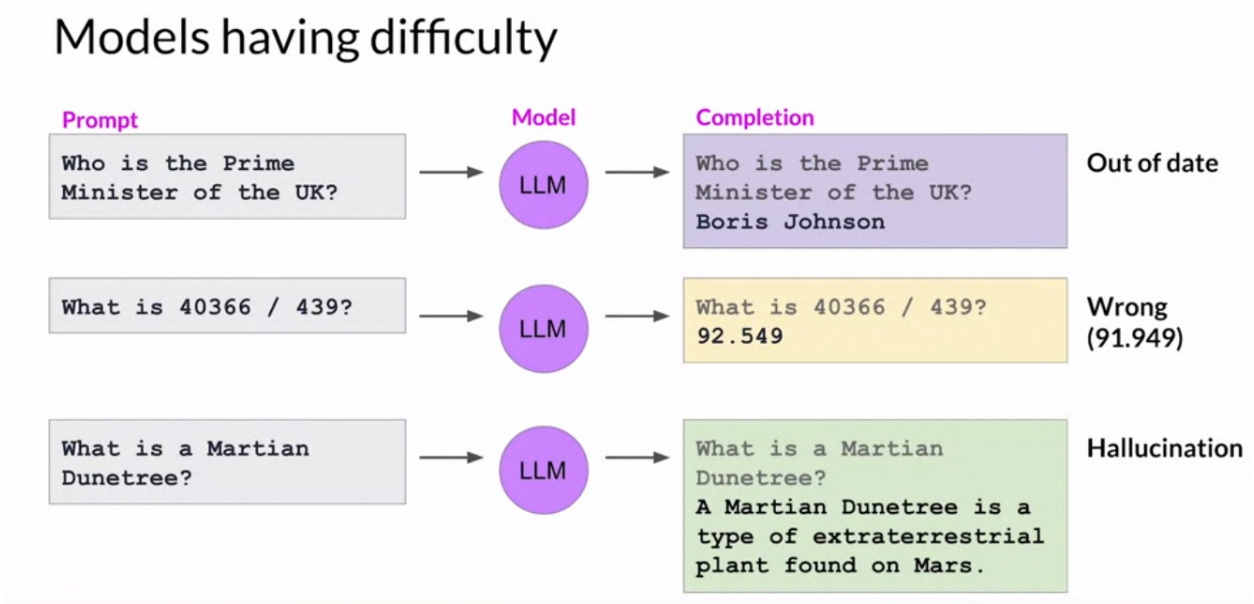
Optimization Techniques
- Optimization techniques, as discussed in the last video, fall in the middle in terms of complexity and effort.
- They can proceed quickly assuming the changes to the model do not significantly impact performance.

Achieving an Optimized and Tuned Language Model (LLM)
- After completing the aforementioned steps, an optimized and tuned LLM should be ready for deployment.
- The LLM should perform well for the specific use case and be optimized for deployment.

Addressing Remaining Performance Problems
- The course's final sequence of videos will explore any remaining performance problems with the LLM that may need to be addressed before application launch.
- Techniques to overcome these problems will also be discussed.

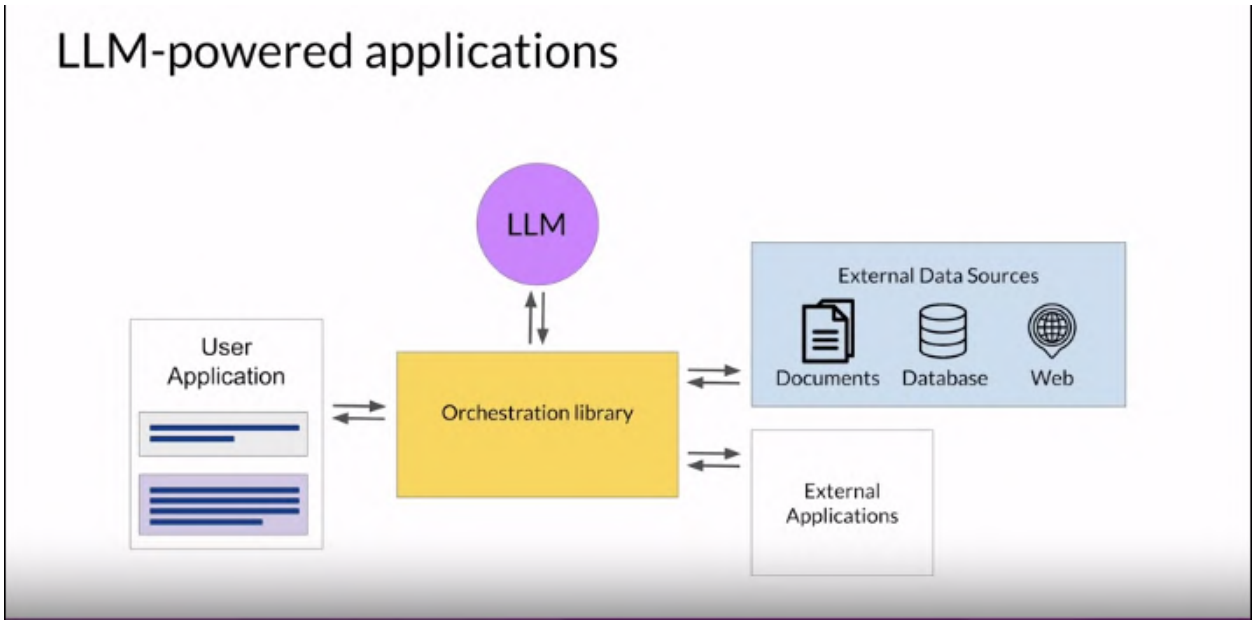
Conclusion
- The generative AI project life cycle involves several crucial stages, including model pre-training, assessment, fine-tuning, alignment, and optimization.
- Careful planning and consideration of time and effort indicators at each phase are essential for successful project execution and deployment.

Using the LLM in applications



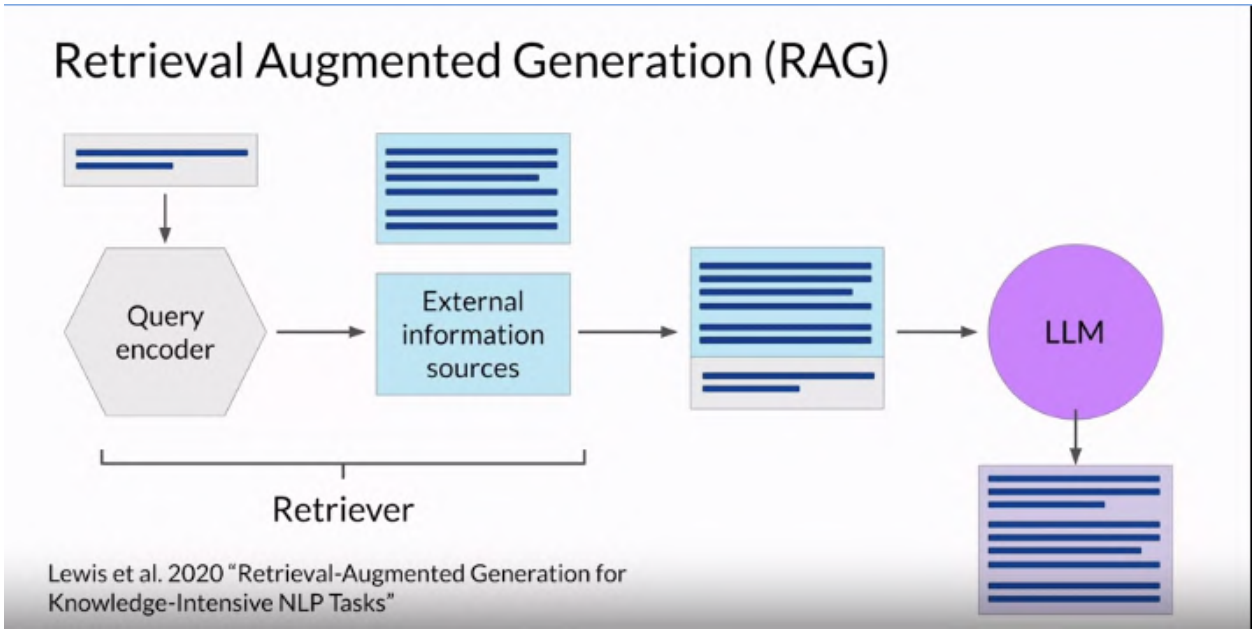
Introduction

- Although training, tuning, and aligning techniques help build a great model, there are broader challenges with large language models (LLMs) that cannot be solved by training alone.
- Examples of such challenges include knowledge cutoff, difficulty with complex math, and text hallucination.

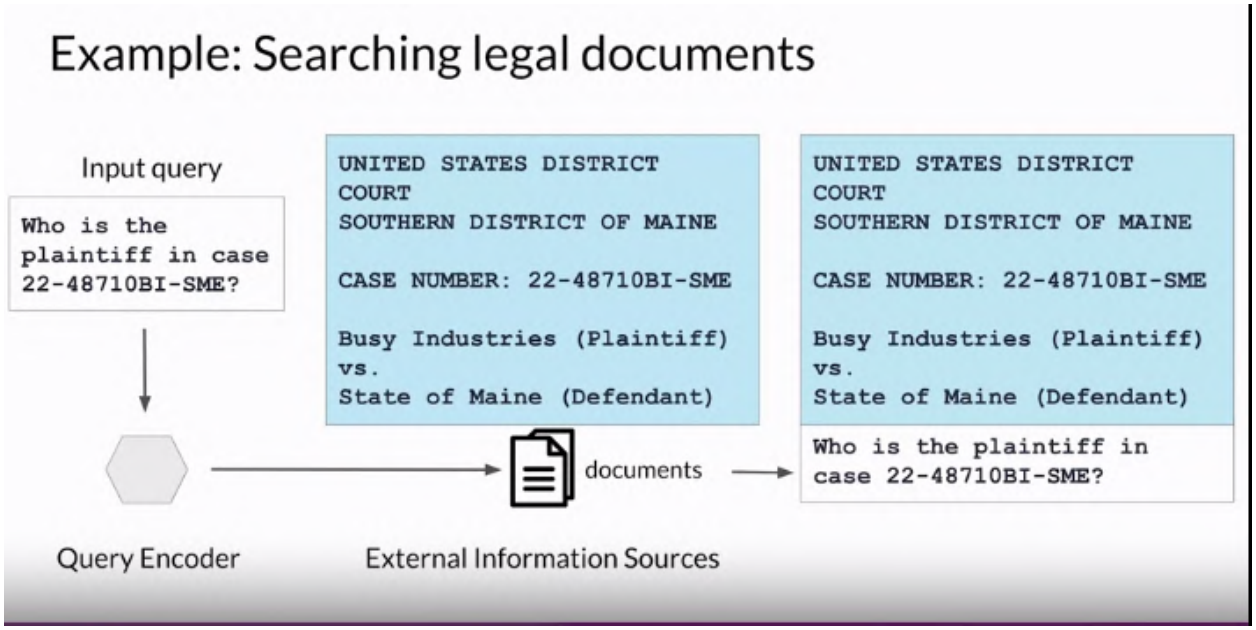


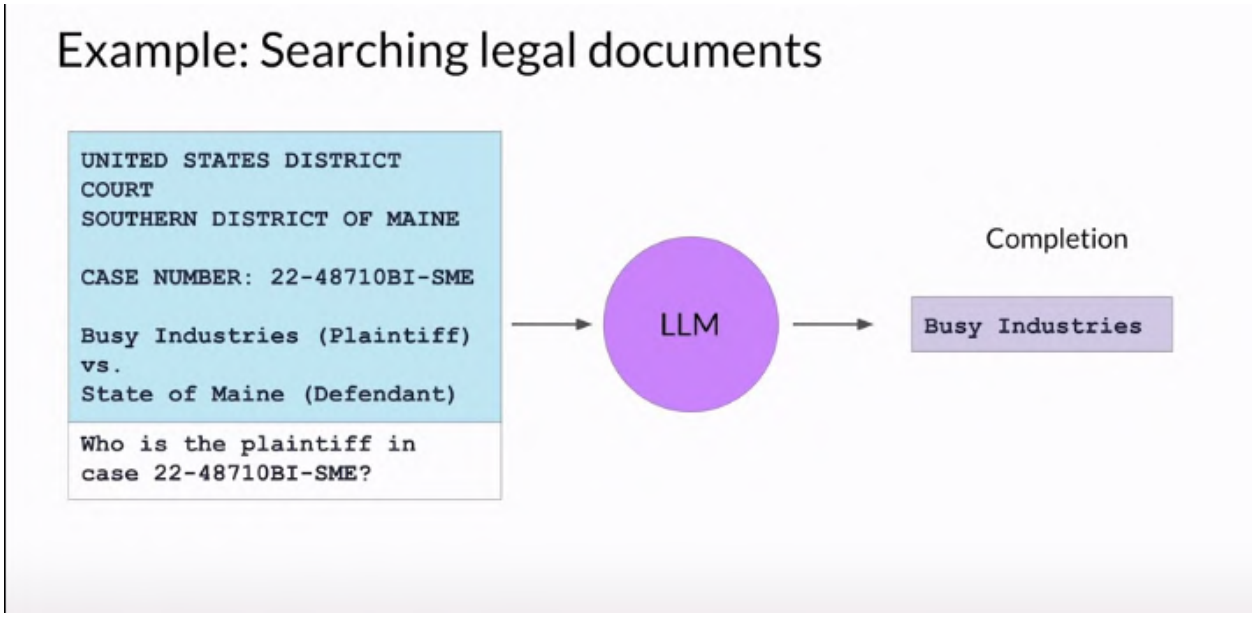
Knowledge Cutoff and Updating Model Understanding

- LLMs possess internal knowledge up to the moment of pretraining, resulting in outdated information.
- Retrieval Augmented Generation (RAG) is a framework that connects LLMs to external data sources and applications to overcome knowledge cutoff and update the model's understanding of the world.
- Unlike retraining the model on new data, which is expensive and requires repeated retraining, RAG allows the model to access additional external data at inference time.
- RAG is beneficial when the language model needs access to data not present in the original training dataset, such as new information or proprietary knowledge.



Implementation of RAG
- RAG is a flexible framework for providing LLMs access to unseen data during training.
- Various implementations of RAG exist, depending on the task and data format.
- One implementation discussed in a Facebook paper involves a model component called the Retriever, comprising a query encoder and an external data source (e.g., vector store, SQL database, or CSV files).
- The Retriever is trained to find relevant documents within the external data that best match the user's input query.
- The retrieved information is combined with the original user query to create an expanded prompt passed to the LLM.
- The LLM generates a completion using the combined information, providing more accurate and relevant answers.

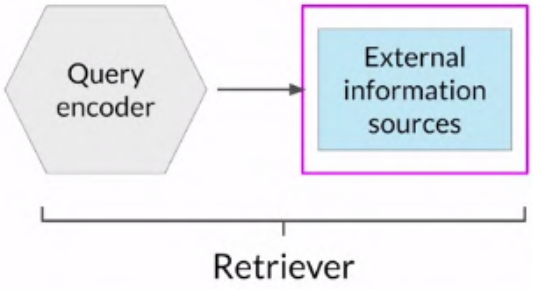




Specific Example: RAG for Legal Discovery

- A specific use case involves a lawyer using RAG to query a corpus of documents (e.g., previous court filings) for case-related information.
- The query encoder encodes the user's input prompt, enabling the Retriever to find relevant entries in the document corpus.
- The retrieved text is combined with the original prompt, and the expanded prompt is passed to the LLM for generating a completion.
- RAG enables complex tasks, such as generating summaries of filings or identifying specific entities within the full corpus of legal documents, increasing the model's utility for the legal use case.

RAG integrates with many types of data sources



External Information Sources

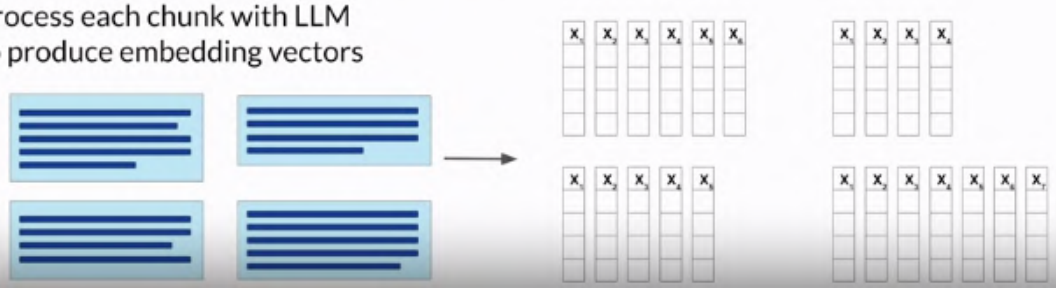
- Documents
- Wikis
- Expert Systems
- Web pages
- Databases
- Vector Store

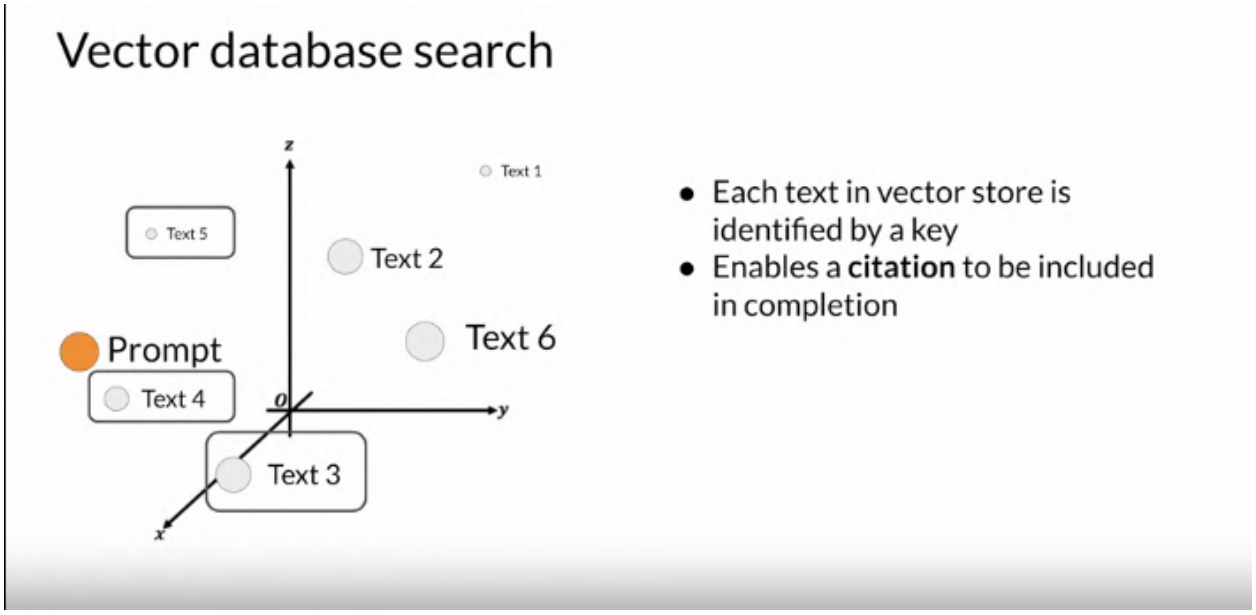
Data preparation for RAG

Two considerations for using external data in RAG:

1. Data must fit inside context window
2. Data must be in format that allows its relevance to be assessed at inference time: **Embedding vectors**

Process each chunk with LLM
to produce embedding vectors

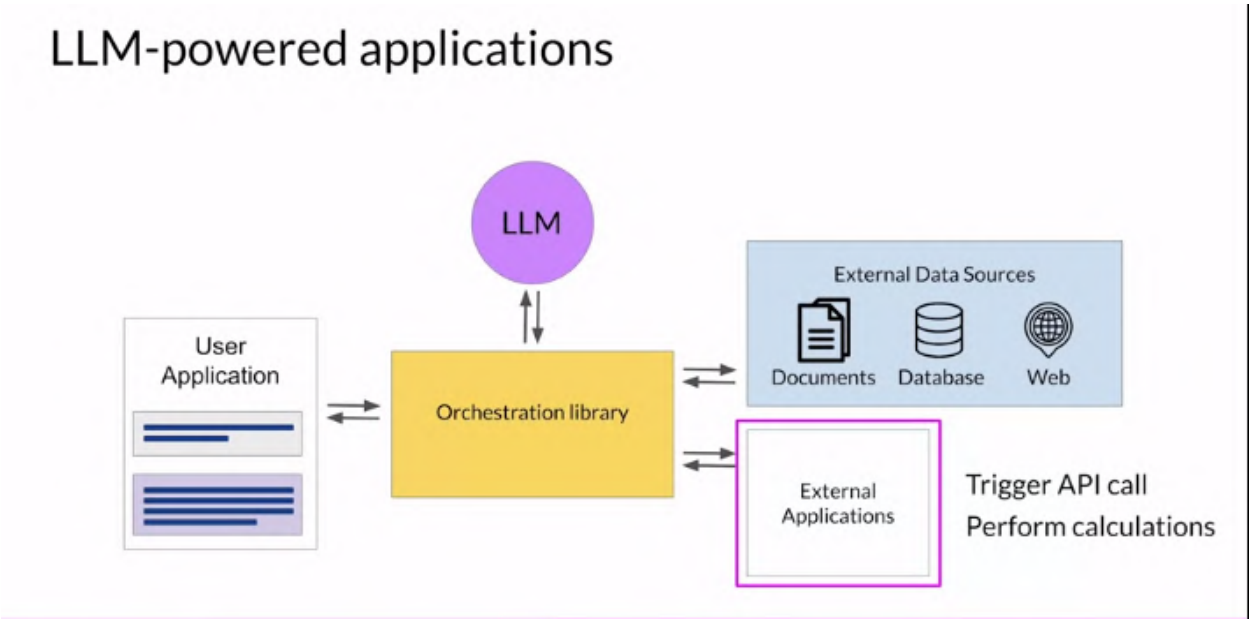




Avoiding Hallucinations and Access to External Information
- RAG helps address the issue of model hallucination by integrating multiple types of external information sources.
- External information sources can include local documents (e.g., private wikis), web pages (e.g., Wikipedia), databases (using SQL queries), and vector stores (containing vector representations of text).
- Vector stores are particularly useful since LLMs internally work with vector representations of language for text generation, allowing fast and efficient relevant searches based on similarity.
- Implementing RAG involves considering the context window size and data format for easy retrieval of relevant text.
- Langchain and other packages can handle the processing of external data chunks and the creation of vector stores.
- Vector databases, a specific implementation of a vector store, enable identification of semantically related text and citations for received documents.

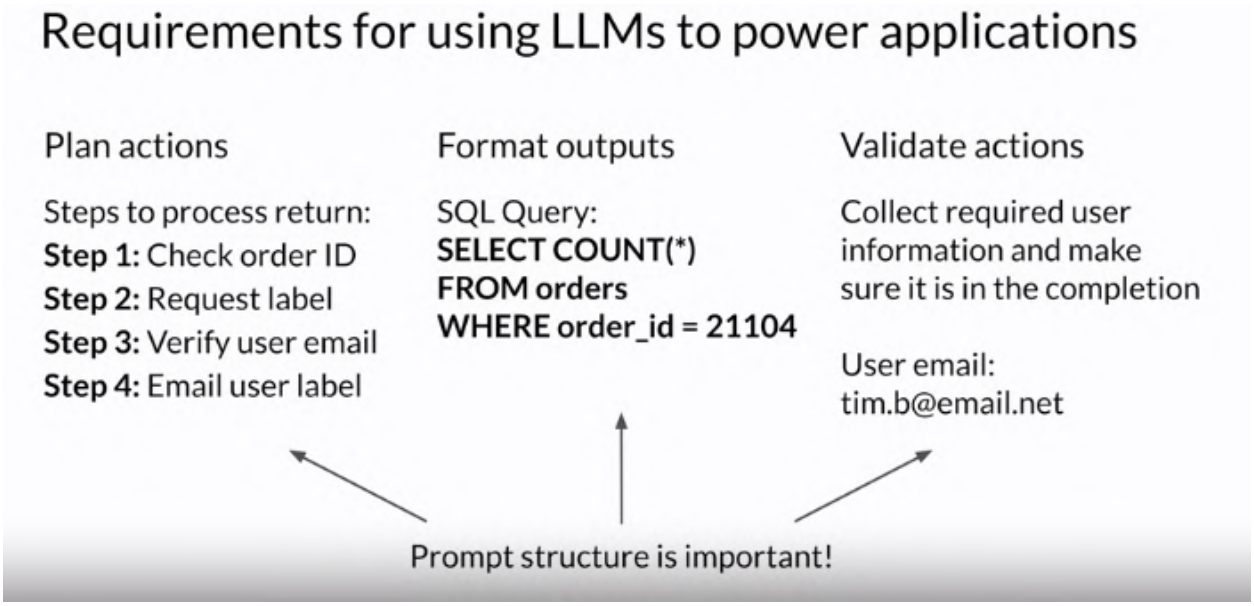
Enhancing Model Reasoning and Planning
- RAG enhances the LLM's capabilities by providing access to up-to-date and relevant information, improving user experience in applications powered by the LLM.
- Next, we'll explore a technique to improve an LLM's reasoning and planning abilities, essential steps for LLM-powered applications.

Interacting with external applications



Customer Service Bot Example
- Customer expresses a desire to return purchased items
- ShopBot asks for the order number and retrieves the information from the transaction database
- Next step: Confirm the items to be returned
- ShopBot initiates a request to the company's shipping partner for a return label using the shipper's Python API
- Customer provides email address for shipping label and confirmation
- Bartlett (ShopBot) informs the customer that the label has been sent, concluding the conversation

Extending LLM Utility Beyond Language Tasks
- Connecting LLMs to external applications allows interaction with the broader world
- LLMs can trigger actions when integrated with APIs and connect to other programming resources
- Prompts and completions play a crucial role in these workflows
- The LLM serves as the application's reasoning engine, determining actions based on completions



Important Information in Completions
- Completions must contain clear instructions for the application to know what actions to take
- Instructions should be understandable and aligned with allowed actions
- Formatting completions appropriately is vital for the broader application to interpret the information
- It could be as simple as a sentence structure or as complex as writing a Python script or generating a SQL command

Examples of Prompt Structures and Information Collection
- SQL query example to check if an order is present in the database
- Collection of information required for validation (e.g., customer's email address for order verification)
- Information collected in completions is crucial to be passed through to the application

Importance of Structuring Prompts
- Properly structuring prompts impacts the quality of generated plans and adherence to desired output format specification
- Correctly structured prompts enhance the model's ability to understand and perform actions effectively

Conclusion
LLMs' interaction with external applications enables their utilization in diverse use cases beyond language tasks. The customer service bot example illustrates how LLMs can trigger actions and communicate with APIs and other programming resources. The proper structuring of prompts and including relevant information in completions is crucial for effective interactions and achieving desired outcomes. The integration of LLMs with external applications expands their utility and paves the way for innovative applications in various domains.

Helping LLMs reason and plan with chain-of-thought

Study Notes: Improving Reasoning in Large Language Models (LLMs) with Chain of Thought Prompting

Challenges in Complex Reasoning for LLMs
- LLMs may struggle with complex reasoning, especially involving multiple steps or mathematics
- Even large models with good performance on many tasks can face difficulties

Demonstrating Model's Difficulty with Multi-Step Math Problem
- Example problem: Determine the number of apples a cafeteria has after using some for lunch and buying some more
- Model generates an incorrect completion (27 apples) despite the correct answer being nine apples

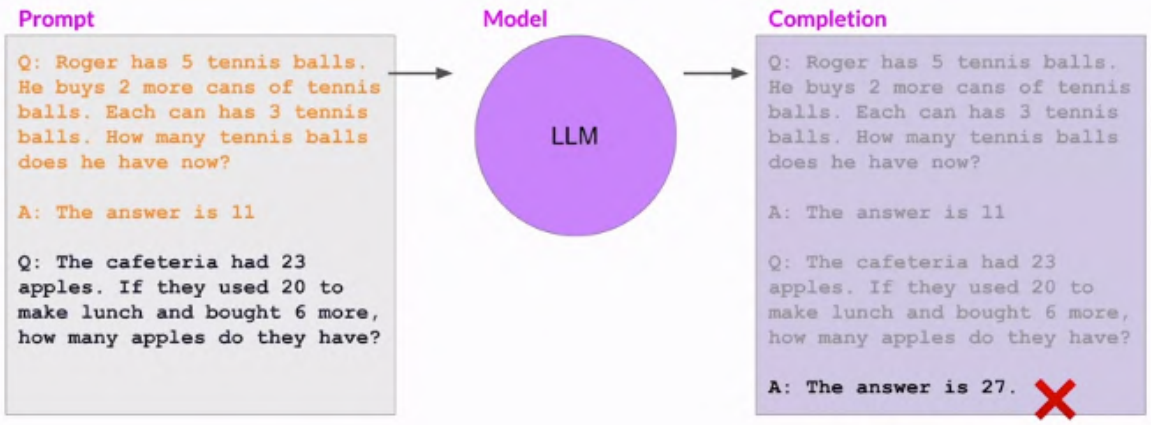
Strategy: Chain of Thought Prompting
- Prompting the model to think more like a human by breaking down the problem into steps
- Teaching the model to reason through the task by including intermediate reasoning steps in examples
- Mimicking the behavior of human reasoning

Chain of Thought Prompting in Action
- One-shot example: Calculating how many tennis balls Roger has after buying some new ones
- Human reasoning steps:
  1. Determine initial number of tennis balls
  2. Note that Roger buys two cans, each containing three balls (total of six new balls)
  3. Add the new balls to the original count (total of 11 balls)
  4. State the answer (11 balls)
- Chain of thought prompt includes reasoning steps in the solution text
- Model generates a more robust and transparent response, correctly determining nine apples left in the cafeteria

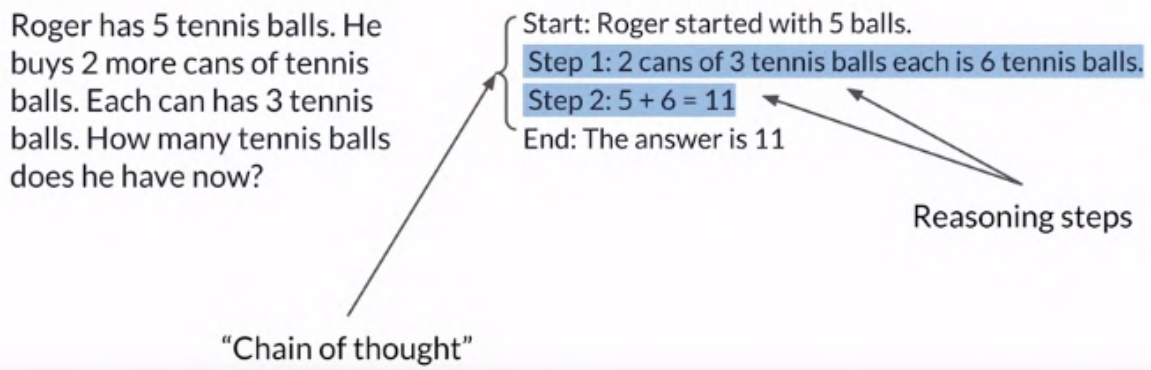
Expanding Chain of Thought Prompting to Other Problems
- Chain of thought prompting can be applied to various types of problems, not just arithmetic
- Example: Physics problem - determining if a gold ring would sink in a swimming pool
- Chain of thought prompt illustrates the reasoning that a pair would float due to its lower density than water
- Model generates a completion with similar structured reasoning, correctly identifying that gold is denser than water and the ring would sink

Limitations of LLM Math Skills
- While chain of thought prompting improves reasoning, LLMs may still struggle with accurate calculations in certain tasks (e.g., totaling sales, calculating tax)

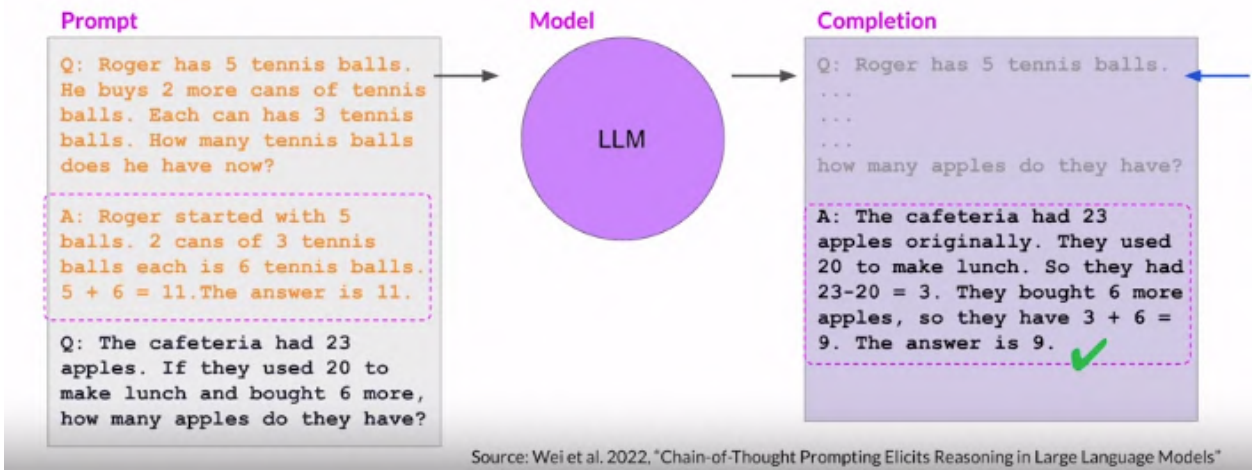
LLMs can struggle with complex reasoning problems



Humans take a step-by-step approach to solving complex problems

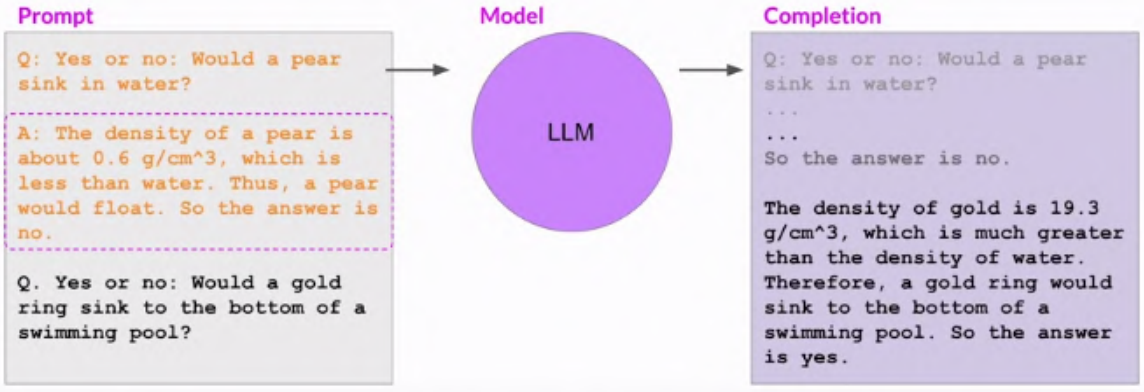


Chain-of-Thought Prompting can help LLMs reason



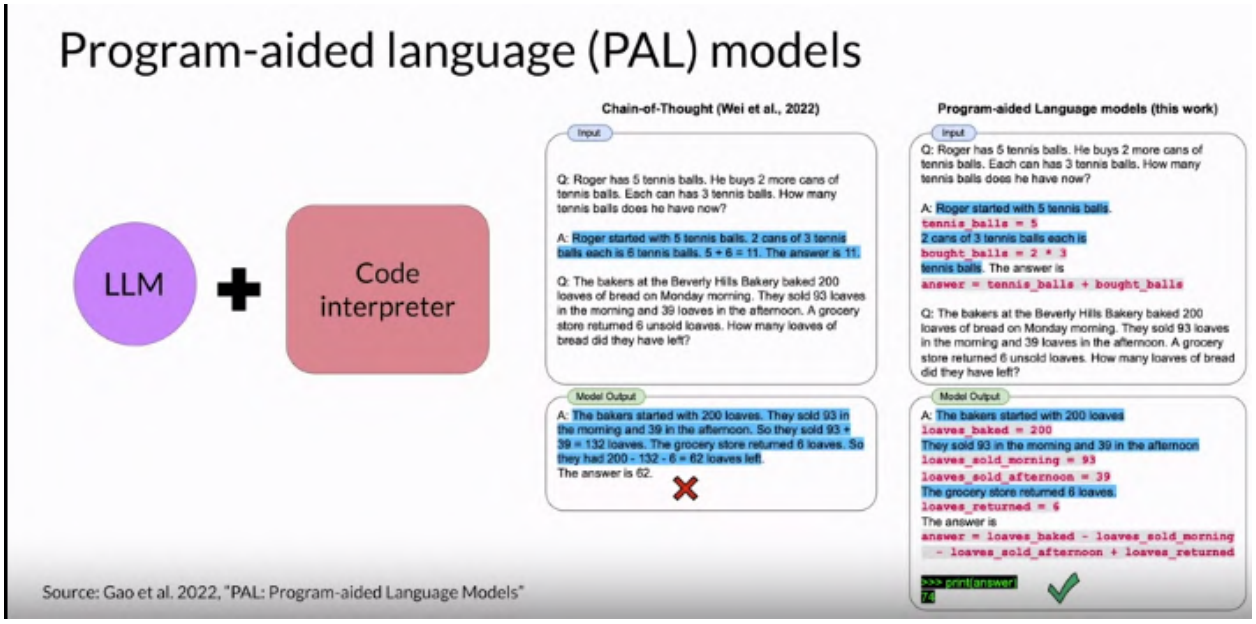
Source: Wei et al. 2022, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models"

Chain-of-Thought Prompting can help LLMs reason



Source: Wei et al. 2022, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models"

Program-aided language models (PAL)



Introduction
- Large Language Models (LLMs) have limited ability to carry out arithmetic and mathematical operations accurately.
- Chain of thought prompting can help, but it has its limitations, especially with larger numbers or complex operations.
- Introducing Program-Aided Language Models (PAL) to augment LLMs with external code interpreters for accurate calculations.

PAL Framework
- PAL first presented by Luyu Gao and collaborators at Carnegie Mellon University in 2022.
- Combines LLMs with external code interpreters to carry out calculations.
- Uses chain of thought prompting to generate executable Python scripts.
- LLM generates completions accompanied by computer code, which is then passed to the interpreter for execution.

Example Prompts Structure
- Example prompts use chain of thought approach with reasoning steps and lines of Python code.
- Python code is used to carry out calculations based on the reasoning steps.
- Variables are declared and assigned values based on the reasoning text.
- Model can work with variables created in other steps.

PAL Implementation Steps
1. Prepare Prompt:

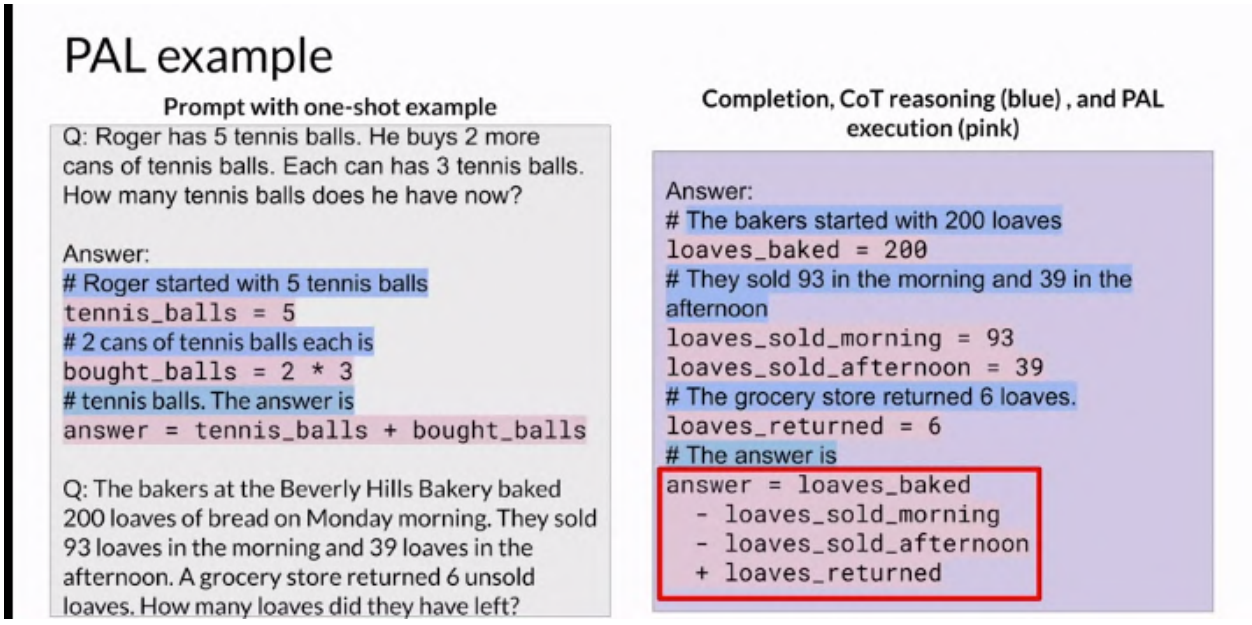
- Format prompt to contain a question and reasoning steps with Python code examples that solve the problem.
- Append the new question to the prompt template.

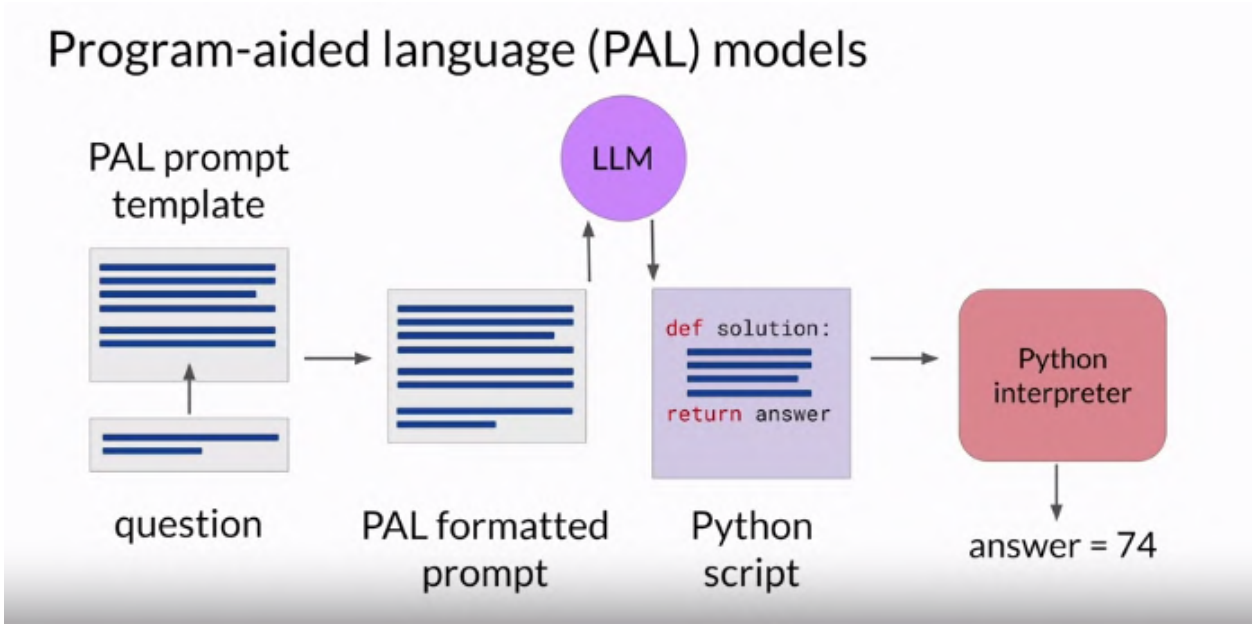
2. Generate Completion:
- Pass the prompt to the LLM, which generates a Python script as a completion based on the examples in the prompt.
- The completion is in the form of a Python script that formats the output based on the example in the prompt.

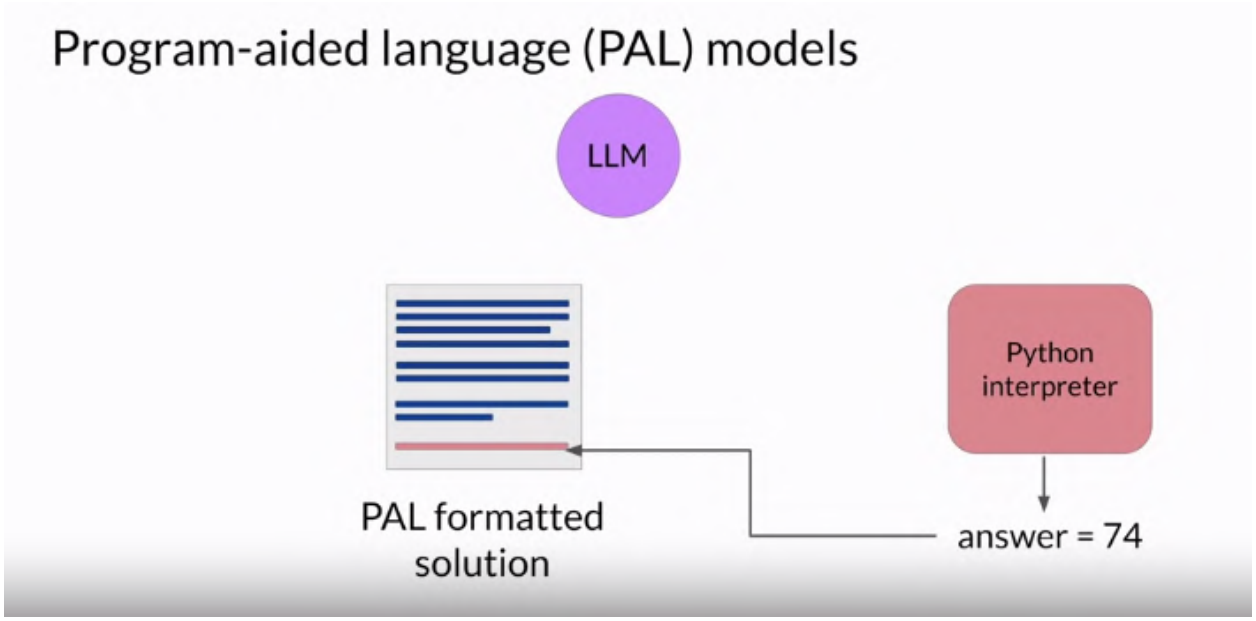
3. Use Python Interpreter:
- Hand off the script to a Python interpreter to run the code and generate an answer.
- The interpreter returns the answer to the application.

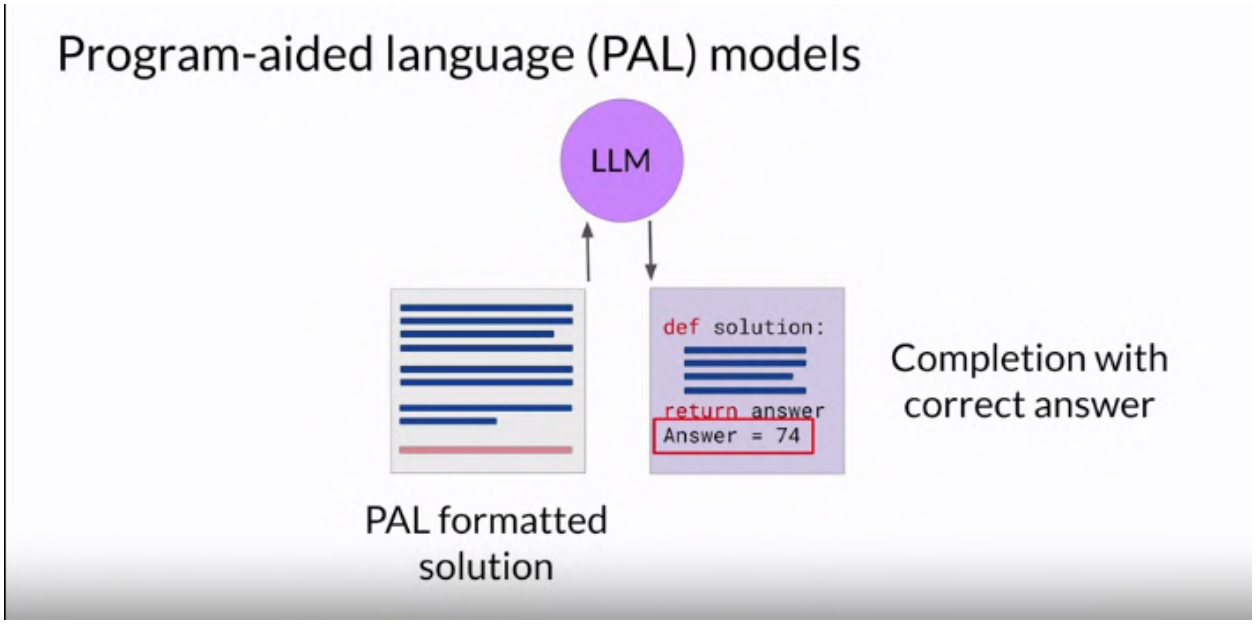
4. Include Answer in Prompt:
- Append the text containing the answer (generated by the interpreter) to the PAL formatted prompt.
- Now, the prompt contains the correct answer in context.

5. Final Completion:
- Pass the updated prompt to the LLM, which generates a completion containing the correct answer.




Program-aided language (PAL) models
PAL prompt template
LLM
def solution:
return answer
Python interpreter
question
PAL formatted prompt
Python script
answer = 74


Program-aided language (PAL) models
LLM
Python interpreter
PAL formatted solution
answer = 74

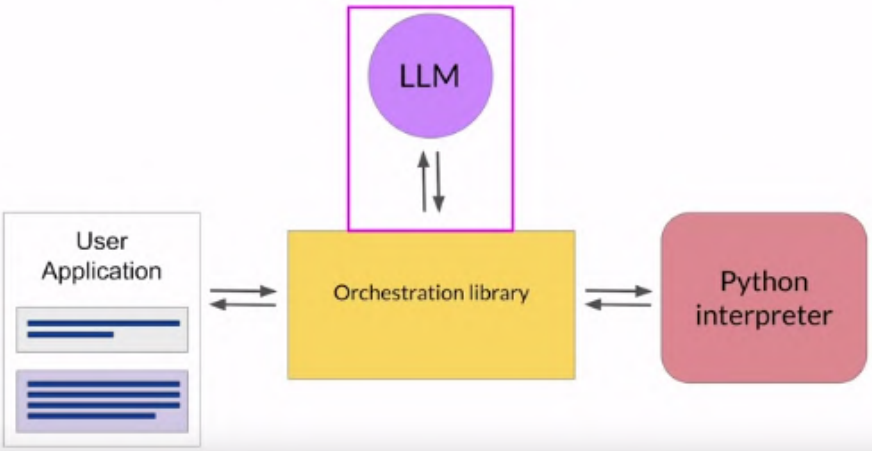


Advantages of PAL
- PAL allows LLMs to interact with external applications, such as Python interpreters, for accurate and reliable calculations.
- Particularly useful for complex math, large numbers, trigonometry, or calculus.

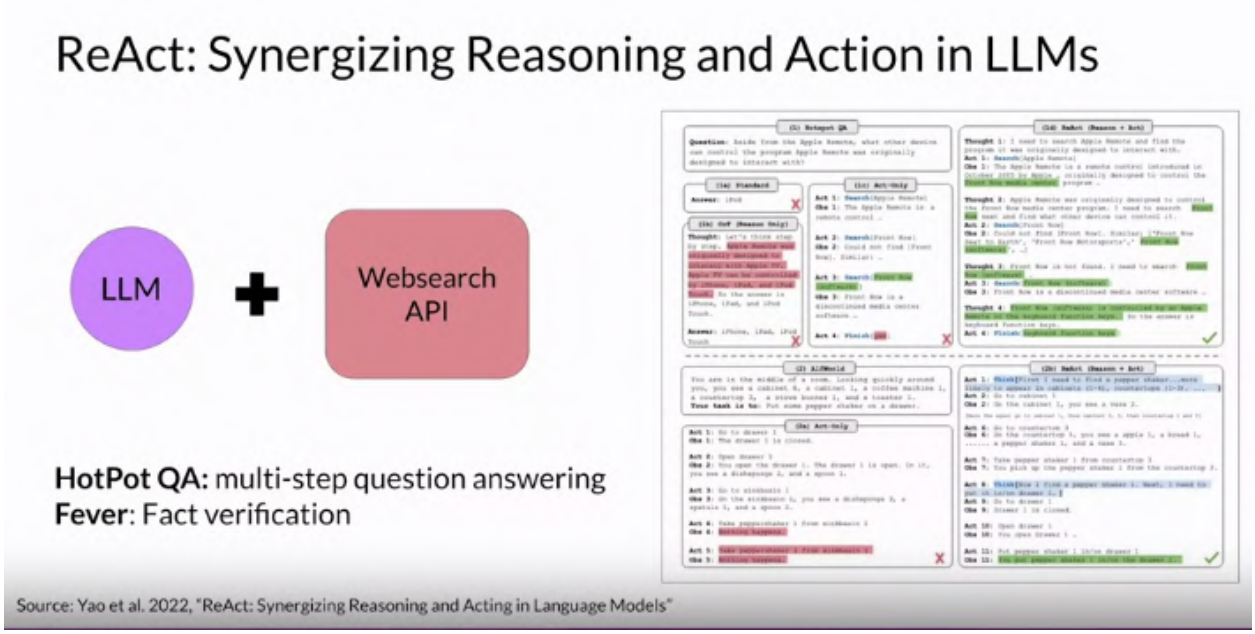
Orchestrator and Complex Applications
- The orchestrator manages the flow of information and calls to external data sources or applications.
- In PAL, the main action is executing Python code, which the LLM generates as a script for the orchestrator to run.
- Real-world applications may require more complex interactions with multiple data sources, validation actions, and external applications.

Conclusion
- PAL is a powerful technique to enhance LLMs with external code interpreters for accurate calculations.
- It provides a solution for overcoming the limitations of LLMs in performing complex math operations.
- The orchestrator can be used to manage complex applications and interactions with external components.

PAL architecture



ReAct: Combining reasoning and action

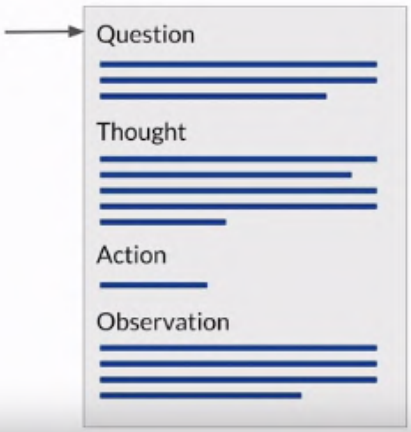


Study Notes: ReAct Framework for Language Model Planning and Execution

Introduction
- ReAct framework for planning and executing complex workflows using large language models (LLMs)
- Combines chain of thought reasoning with action planning
- Proposed by researchers at Princeton and Google in 2022
- Examples based on Hot Pot QA and fever benchmarks, requiring reasoning over multiple Wikipedia passages

ReAct Framework Components
- Structured examples used to guide the LLM's reasoning and decision-making
- Question, thought, action, and observation trio in each example
- Three allowed actions: search (Wikipedia entry about a topic), lookup (search for a string on a Wikipedia page), and finish (completing the task)
- Instructions define a set of allowed actions for the LLM to choose from

ReAct: Synergizing Reasoning and Action in LLMs

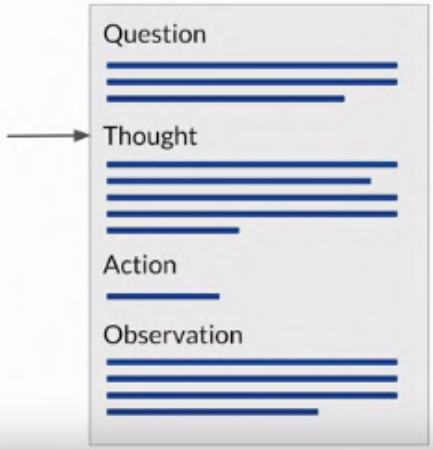


Question: Problem that requires advanced reasoning and multiple steps to solve.

E.g.
"Which magazine was started first, *Arthur's Magazine* or *First for Women*?"

Source: Yao et al. 2022, "ReAct: Synergizing Reasoning and Acting in Language Models"

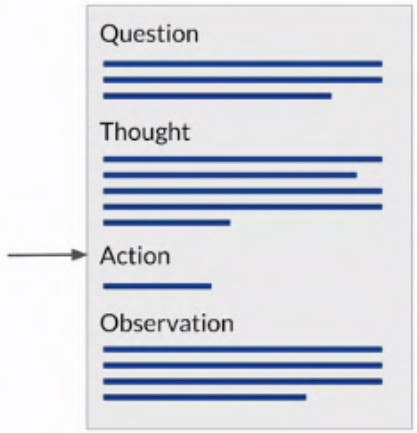
ReAct: Synergizing Reasoning and Action in LLMs



Thought: A reasoning step that identifies how the model will tackle the problem and identify an action to take.

"I need to search Arthur's Magazine and First for Women, and find which one was started first."

ReAct: Synergizing Reasoning and Action in LLMs



Action: An external task that the model can carry out from an allowed set of actions.

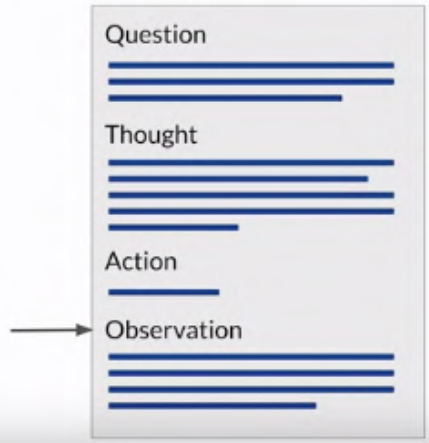
E.g.

```
search[entity]
lookup[string]
finish[answer]
```

Which one to choose is determined by the information in the preceding thought.

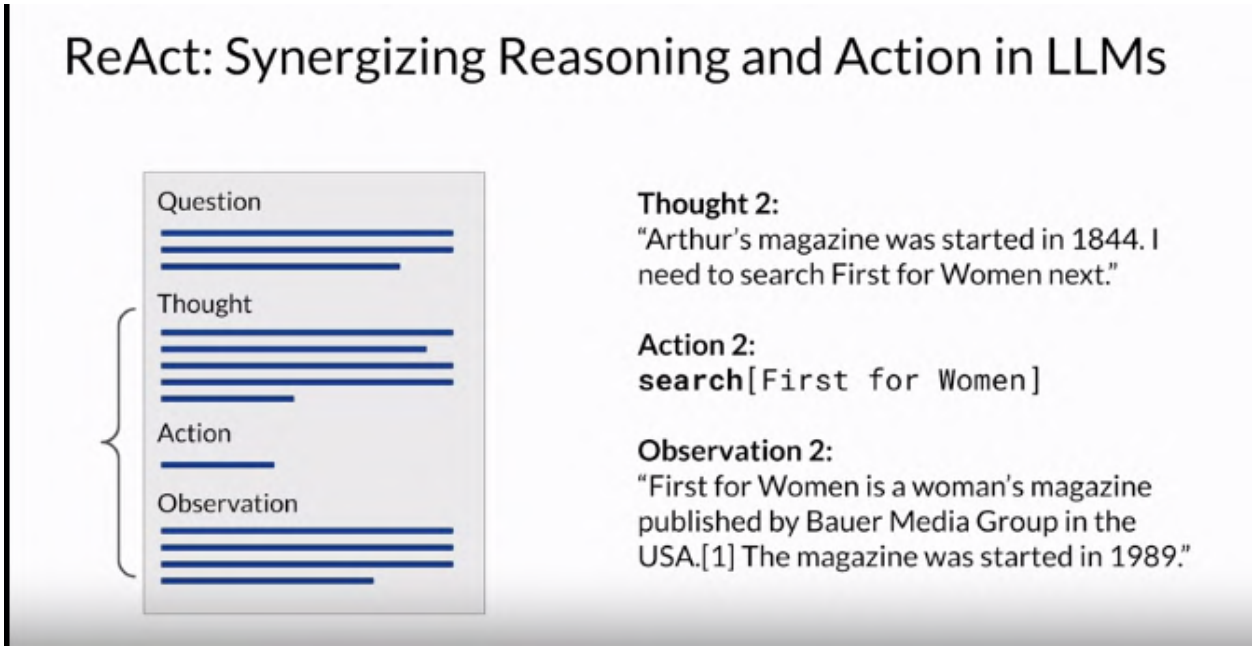
```
search[Arthur's Magazine]
```

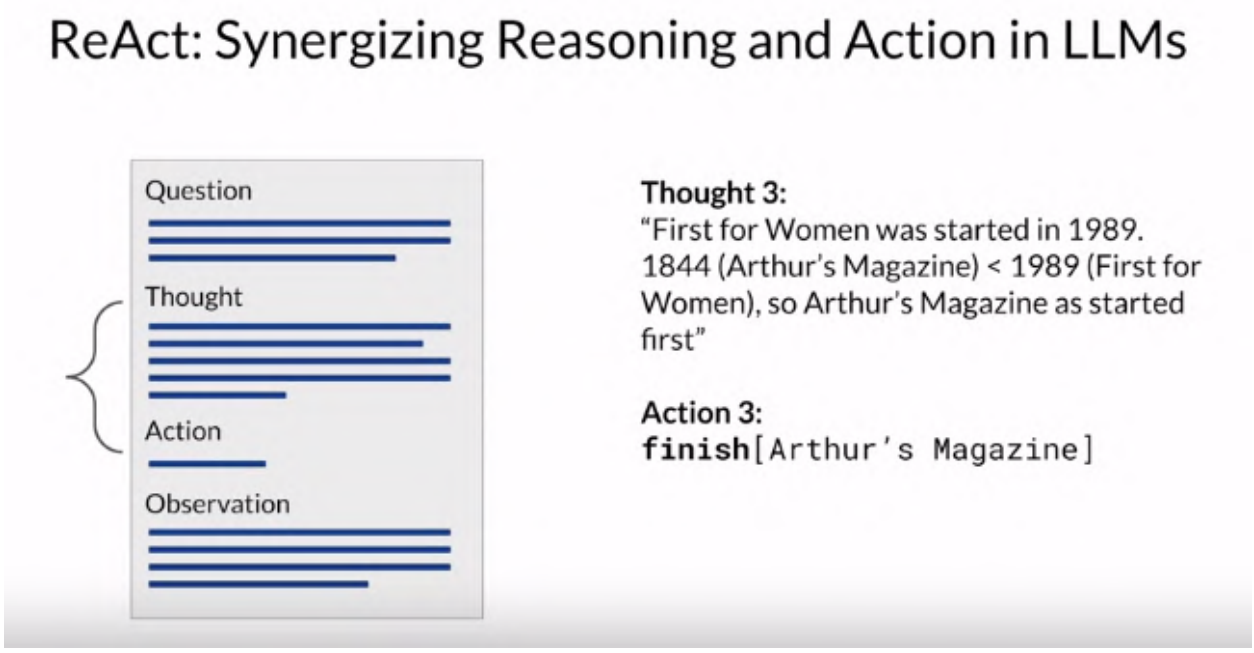
ReAct: Synergizing Reasoning and Action in LLMs



Observation: the result of carrying out the action

E.g.
"Arthur's Magazine (1844-1846) was an American literary periodical published in Philadelphia in the 19th century."

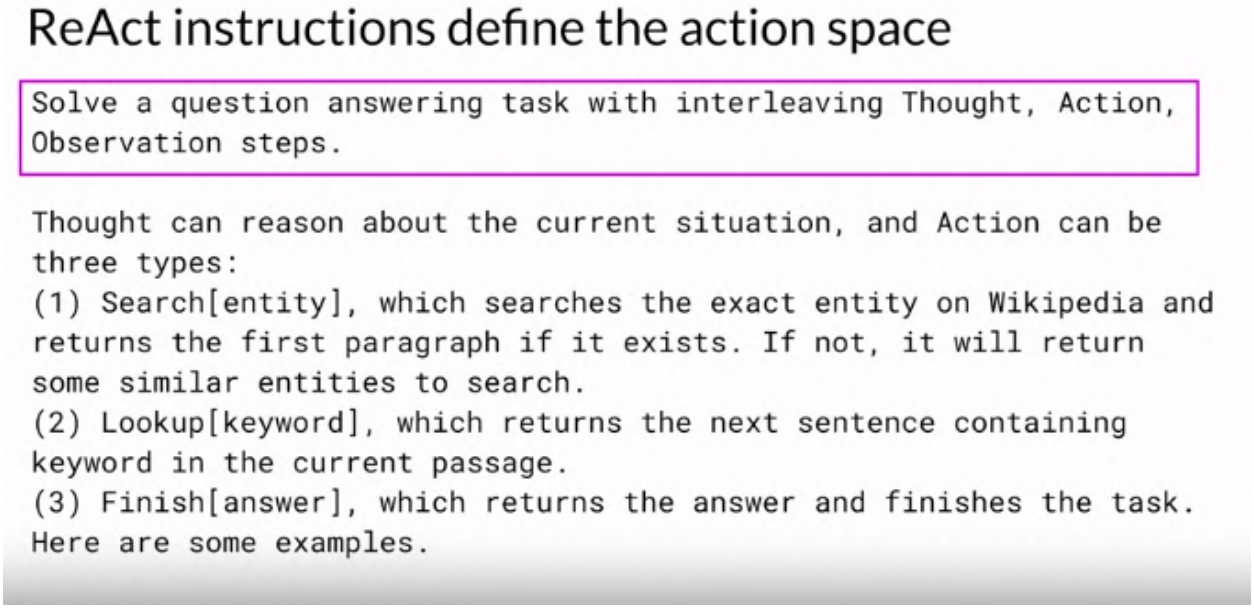


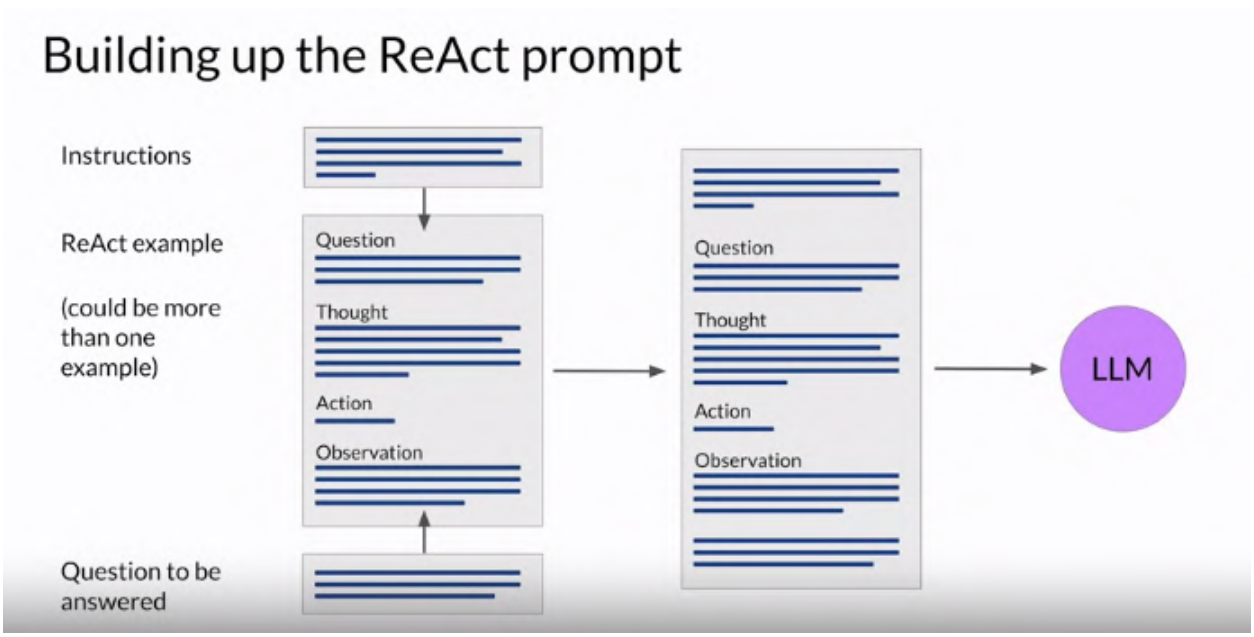


Step-by-Step Execution of ReAct Framework

1. Prompt begins with a question that requires multiple steps to answer
2. Thought: Reasoning step guiding the model to tackle the problem and identify an action to take
3. Action: Model selects an action from the predefined list to interact with an external application or data source (e.g., Wikipedia API)

4. Observation: New information from the external search is incorporated into the prompt's context
5. Cycle repeats until the final answer is obtained

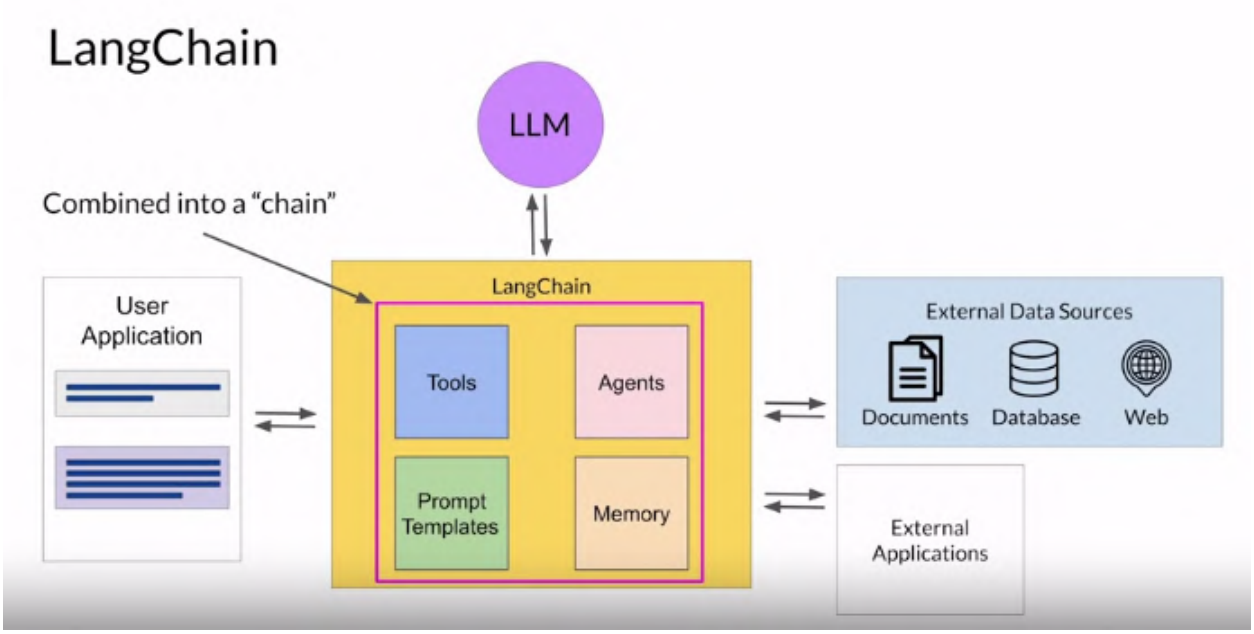




ReAct: Reasoning and action

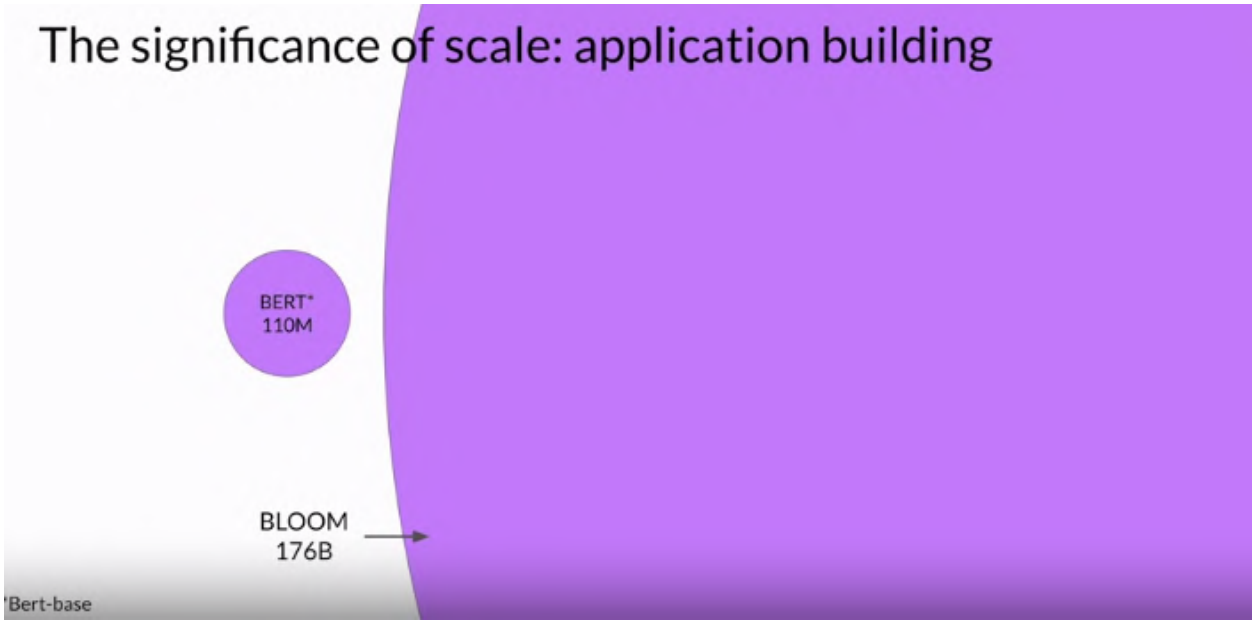
This paper introduces ReAct, a novel approach that integrates verbal reasoning and interactive decision making in large language models (LLMs). While LLMs have excelled in language understanding and decision making, the combination of reasoning and acting has been neglected. ReAct enables LLMs to generate reasoning traces and task-specific actions, leveraging the synergy between them. The approach demonstrates superior performance over baselines in various tasks, overcoming issues like hallucination and error propagation. ReAct outperforms imitation and reinforcement learning methods in interactive decision making, even with minimal context examples. It not only enhances performance but also improves interpretability, trustworthiness, and

diagnosability by allowing humans to distinguish between internal knowledge and external information.
In summary, ReAct bridges the gap between reasoning and acting in LLMs, yielding remarkable results across language reasoning and decision making tasks. By interleaving reasoning traces and actions, ReAct overcomes limitations and outperforms baselines, not only enhancing model performance but also providing interpretability and trustworthiness, empowering users to understand the model's decision-making process.



LangChain Framework
- LangChain framework for developing applications powered by language models
- Provides modular pieces containing components necessary to work with LLMs
- Includes prompt templates for various use cases and memory for storing LLM interactions
- Pre-built tools for tasks like external dataset calls and APIs
- Chains: Connecting components to form a sequence of actions optimized for different use cases

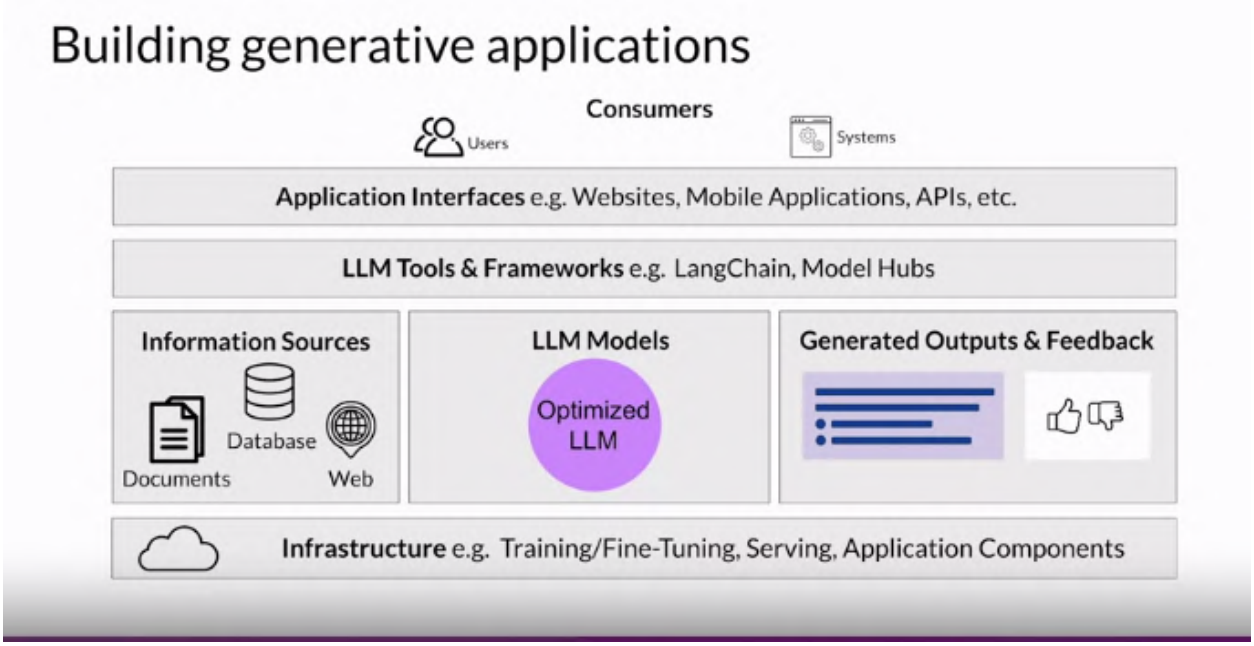
Agents in LangChain
- Agents: Interpret user input and determine which tools to use for the task
- Incorporated into chains to plan and execute a series of actions
- Current agents include PAL and ReAct, among others



Model Scalability and Planning
- Model size impacts reasoning and planning capabilities
- Larger models are more suitable for complex prompting techniques like PAL and ReAct
- Smaller models may require additional fine-tuning to improve reasoning abilities

Conclusion
- ReAct framework enables LLMs to execute complex workflows through reasoning and action planning
- LangChain framework provides a flexible and modular approach to develop LLM-powered applications
- Model scalability plays a crucial role in the success of planning and execution tasks using LLMs

LLM application architectures



Introduction
In this final section of the lesson, you'll explore additional considerations for building applications powered by Large Language Models (LLMs). The building blocks for creating end-to-end solutions for LLM-powered applications involve several key components, including infrastructure, LLM models, retrieval of information from external sources, output management, and user interface with security components.

Building Blocks for LLM-Powered Applications
1. Infrastructure Layer:
   - Provides compute, storage, and network resources for serving LLMs and hosting application components.
   - Can be on-premises infrastructure or on-demand Cloud services.

2. Large Language Models (LLMs):
   - Includes foundation models and task-specific models adapted for the application.
   - Deployed on appropriate infrastructure for real-time or near-real-time inference.

3. Retrieval Augmented Generation:
   - May require retrieving information from external sources for generating LLM completions.

4. Output Management:
   - Mechanism to capture and store outputs based on use case requirements.
   - May store user completions during a session to augment LLM's fixed contexts window size.

- Gathering user feedback for fine-tuning, alignment, or evaluation as the application matures.

5. Additional Tools and Frameworks:
- Utilize tools and frameworks to implement techniques like pow react or chain of thought prompting.
- Model hubs for centralized management and sharing of models for application use.

6. User Interface and Security Components:
- User interface (e.g., website or REST API) for user interaction with the application.
- Includes security components for secure interaction with the application.

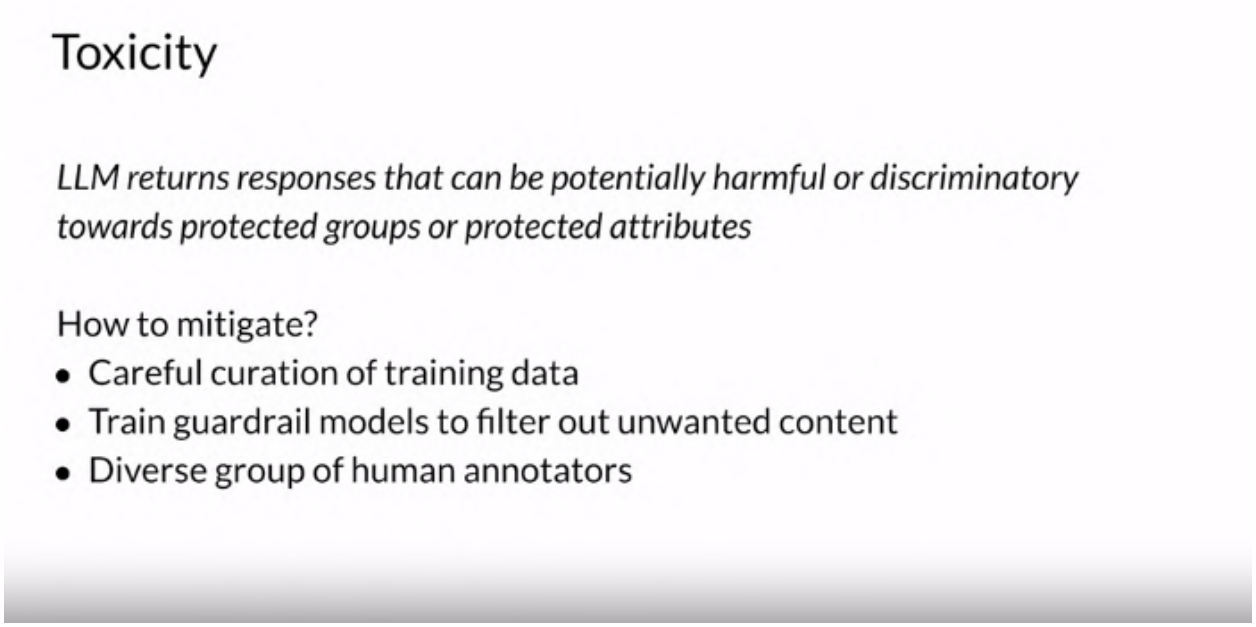
Architecture Stack for Generative AI Applications
- High-level representation of the various components for generative AI applications.
- Users, whether human end-users or other systems accessing the application through APIs, interact with this stack.

Role of LLMs in Generative AI Applications
- LLMs are only one part of the story in building end-to-end generative AI applications.
- LLMs need to be aligned with human preferences and can be fine-tuned using reinforcement learning with human feedback (RLHF).
- RL reward models and human alignment datasets are available for quick alignment of models.
- RLHF effectively improves model alignment, reduces toxicity in responses, and enhances safety in production.
- Techniques like distillation, quantization, or pruning optimize models for inference, reducing hardware resource requirements.
- Structured prompts and connections to external data sources and applications improve model performance in deployment.

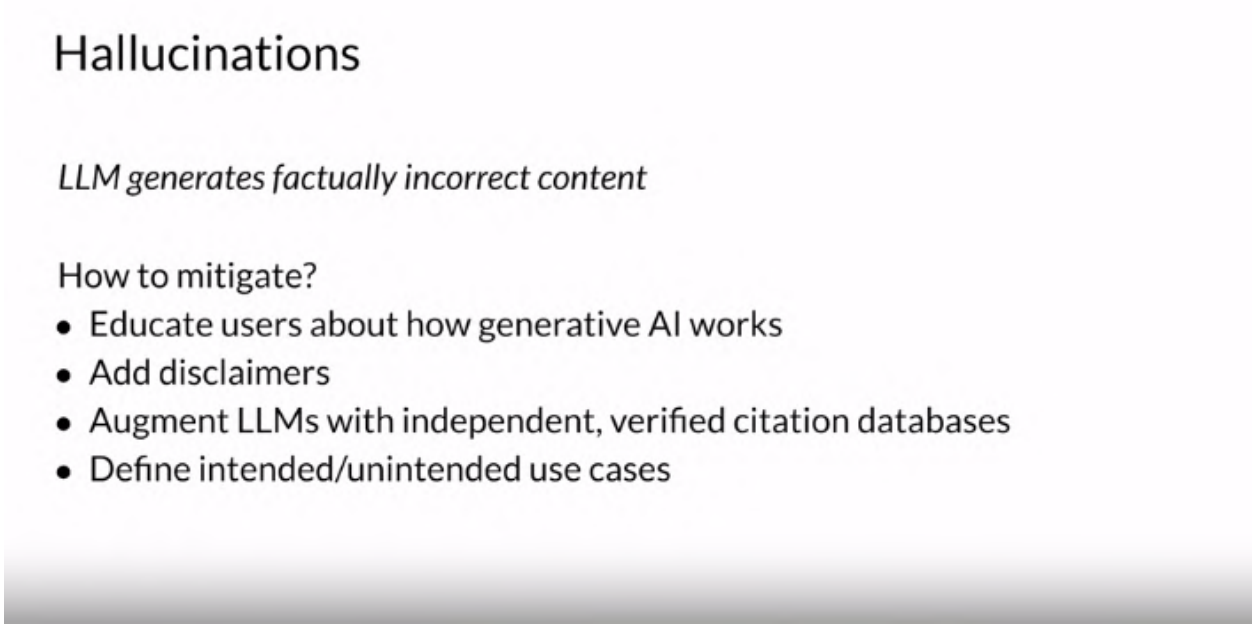
Conclusion
- LLMs play a crucial role as the reasoning engine in generative AI applications.
- Frameworks like len chain facilitate quick development, deployment, and testing of LLM-powered applications.
- Active research areas in the field will likely shape the trajectory of LLM-powered applications in the future.

Responsible AI

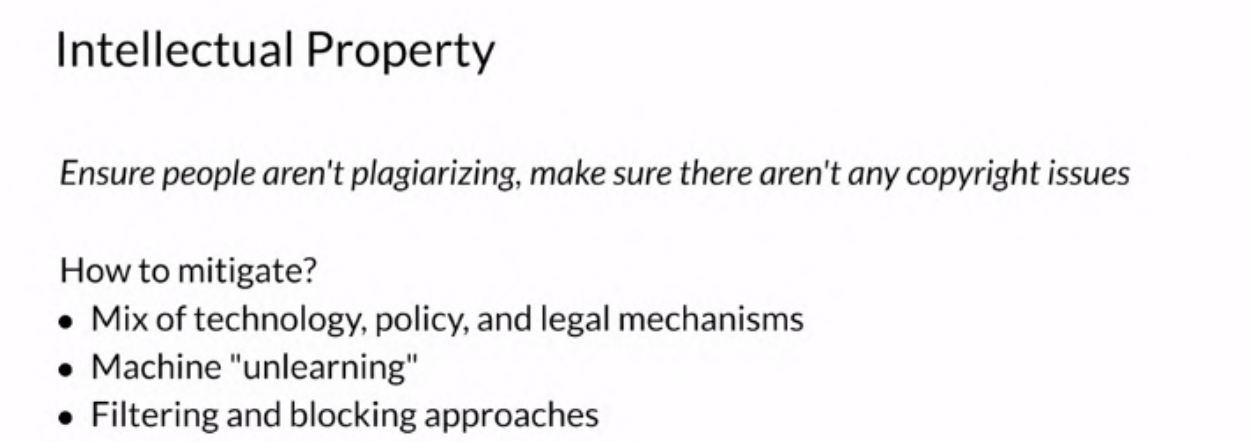
New Risks and Challenges in Generative AI
- Three main challenges:
  1. Toxicity: Harmful or discriminatory language/content towards certain groups
  2. Hallucinations: False statements or content with no basis
  3. Intellectual Property: Issues of plagiarism and copyright infringement



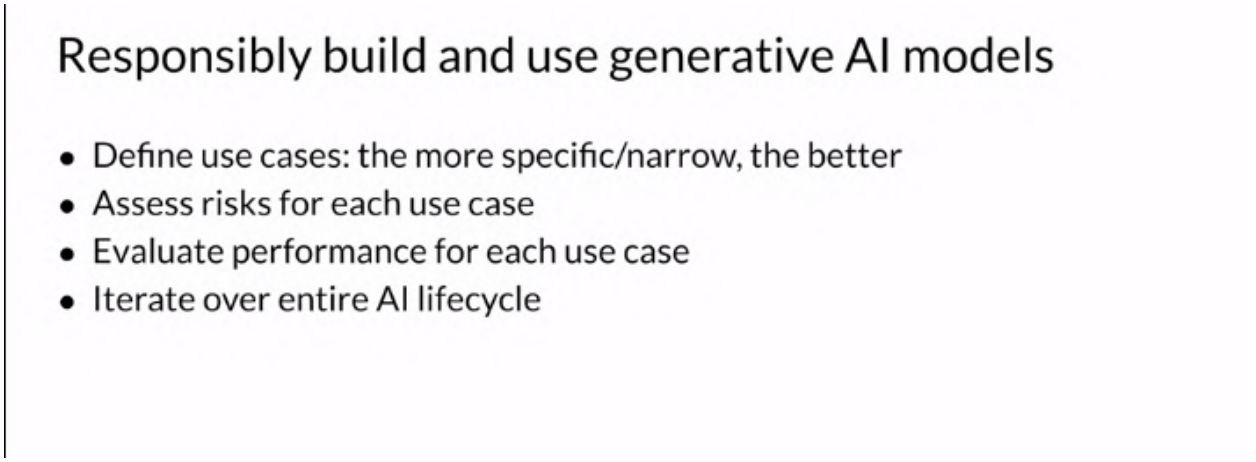
Mitigating Toxicity
- Curation of training data to filter out harmful content
- Training guardrail models to detect and remove unwanted content
- Providing diverse guidance to human annotators for better data quality



Addressing Hallucinations
- Educating users about the limitations of generative AI
- Augmenting models with independent and verified sources
- Developing methods for attributing generated output to specific training data
- Defining intended and unintended use cases



Handling Intellectual Property Issues
- Addressed through a mixture of technologies and legal mechanisms
- Incorporating a system of governance and accountability measures
- Exploring machine unlearning and filtering/blocking approaches



Responsible Building and Use of Generative AI Models
- Defining specific and narrow use cases for better evaluation
- Assessing risks associated with each use case
- Evaluating performance based on data and system characteristics
- Implementing responsibility throughout the AI lifecycle
- Issuing governance policies and accountability measures

Exciting Research Topics in Responsible AI
- Watermarking and fingerprinting for content traceability
- Models to determine if content was created with generative AI
- The future of AI as accessible and inclusive

Conclusion
- Responsible AI is essential for the growth of AI
- Continuous research and innovation in the field of generative AI
- Collaboration and diverse perspectives are crucial for responsible AI advancements