Coursera: Generative AI with Large Language Models

Table of Contents

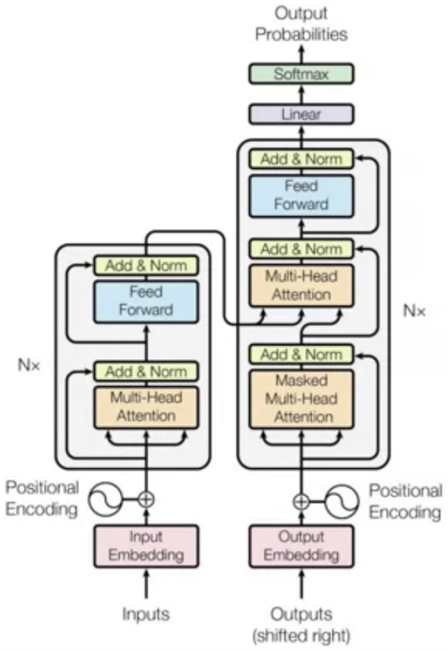
Week 1 Part I. Introduction to LLMs and the generative AI project lifecycle

LLM Use cases:

- Basic chatbots: built upon next word generation
- Text summarization
- Language translation - can also be from natural language to machine code
- Entity Extraction (under information retrieval)
- Augmenting LLMs by invoking external APIs from them

History of generative AI:

1. Weakness of RNN: Generating text with RNN: we need to scale the resources the model use to get as many texts as possible to make reliable prediction. Models needs to have an understanding of the whole sentence or even the whole document. The problem here is that language is complex. In many languages, one word can have multiple meanings. These are homonyms.
2. Transformer:
    a. It can be scaled efficiently to use multi-core GPUs,
    b. It can parallel process input data, making use of much larger training datasets
    c. Crucially, it's able to learn to pay attention to the meaning of the words it's processing.
    d. The flowchart below is the transformer architect from the paper, Attention is All You Need.



Transformer:

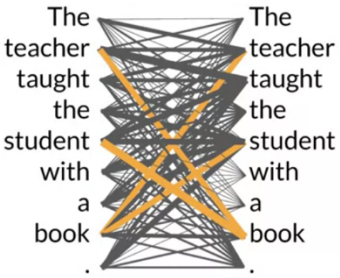
Steps in Transformer:

- Step 1: Using a tokenizer to convert text to numbers (using vocabulary indexing/ stemming/ lemmatising)
- Step 2: Create embedding vectors
- Step 3: Positional encoding - as tokens are processed in parallel, add positional encoding to preserve info about word order
- Step 4: Self-Attention layer - analyze relationships between tokens
    - Multi-headed self-attention: Multiple sets of self-attention weights are learnt in parallel (number of heads varies, 12-100 common)
    - Each head learns a diff aspect of the language
- Step 5: Feed forward network- get an output vector of logits (raw probability scores) with values proportional to the probability of each token
- Step 6: Softmax layer - normalize logits to probability score
- Step 7: Find most probable next token (this step can be varied based on problem)

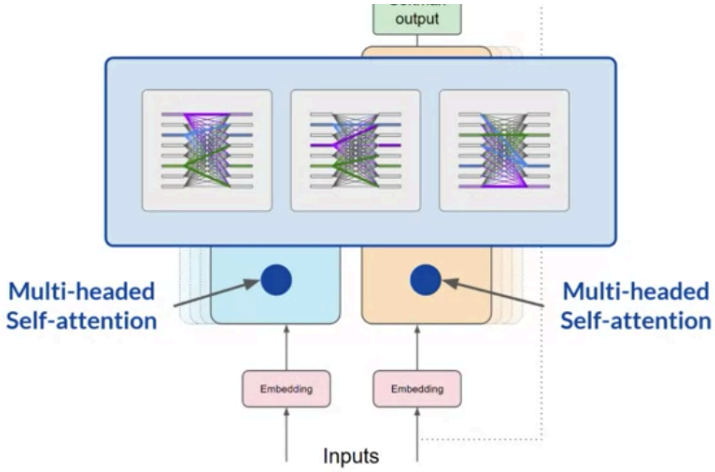
Detailed illustration of each component in Transformer:

1) Attention map: Learn the relevance of each word.

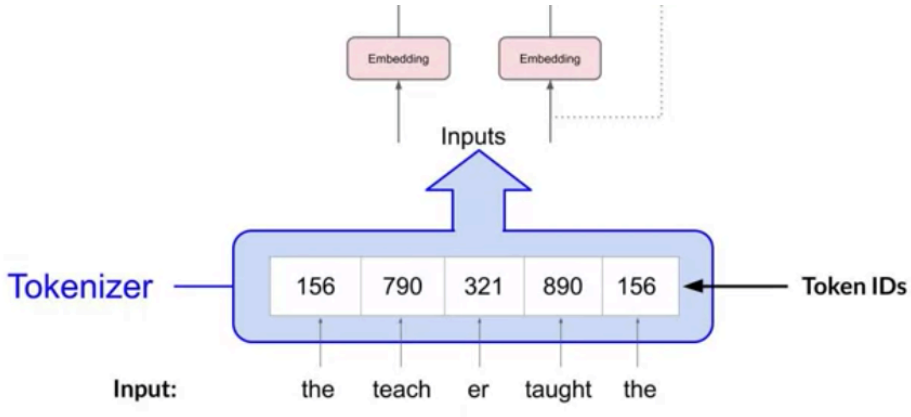
Self-attention



Multi-head self-attention

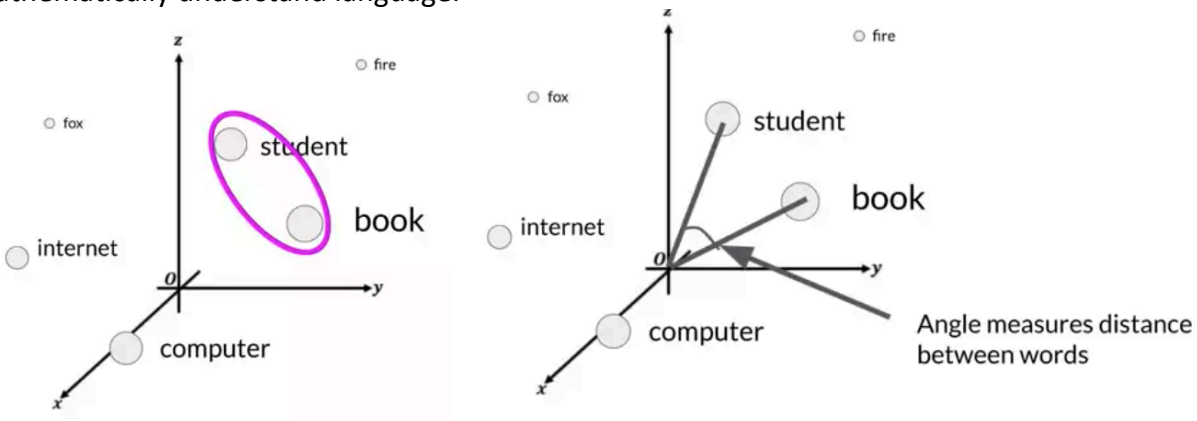


2) Tokenizer:



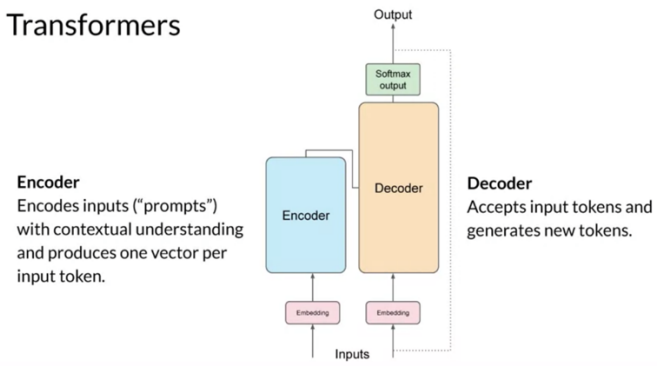
3) Embedding space:

For simplicity, if you imagine a vector size of just three, you could plot the words into a three-dimensional space and see the relationships between those words. You can see now how you can relate words that are located close to each other in the embedding space, and how you can calculate the distance between the words as an angle, which gives the model the ability to mathematically understand language.



4) Encoder and decoder:

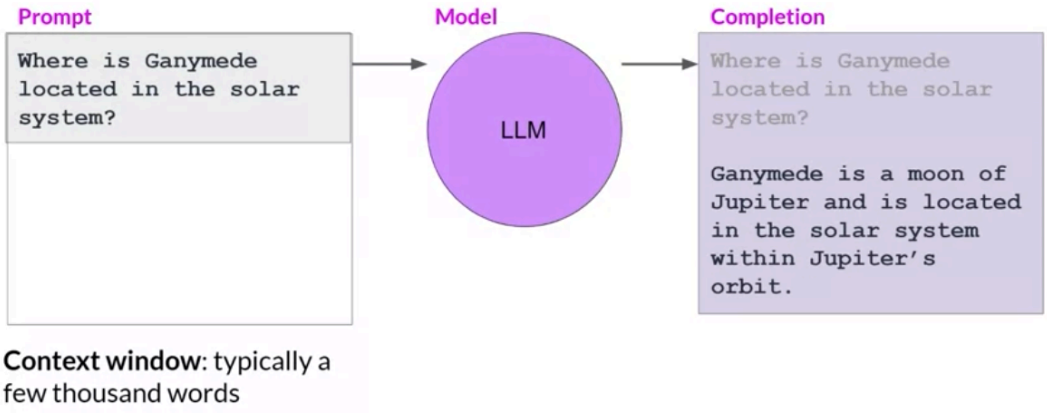
The encoder encodes input sequences into a deep representation of the structure and meaning of the input. The decoder, working from input token triggers, uses the encoder's contextual understanding to generate new tokens. It does this in a loop until some stop condition has been reached.



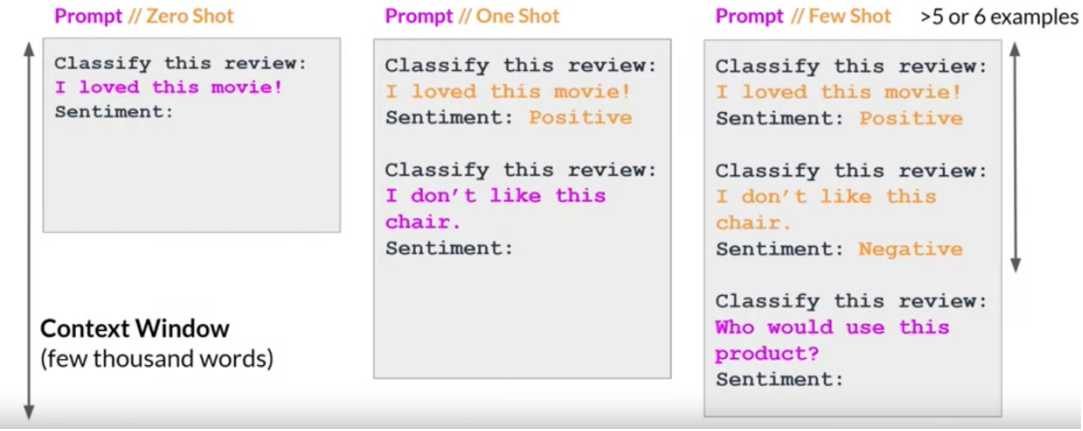
Three types of transformer models:

- Encoder only models: work as sequence-to-sequence models, but without further modification, the input sequence and the output sequence of the same length. Their use is less common these days, but by adding additional layers to the architecture, you can train encoder-only models to perform classification tasks such as sentiment analysis, BERT is an example of an encoder-only model.
- Encoder Decorder models: perform well on sequence-to-sequence tasks such as translation, where the input sequence and the output sequence can be different lengths. You can also scale and train this type of model to perform general text generation tasks.
- Decoder only models: most-commonly used. These models can now generalize to most tasks. Popular decoder-only models include the GPT family of models, BLOOM, Jurassic, LLaMA, and many more.

Prompting and prompt engineering:

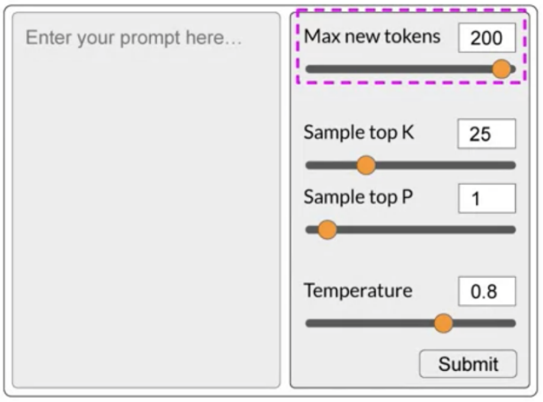


Summary of in-context learning (ICL)



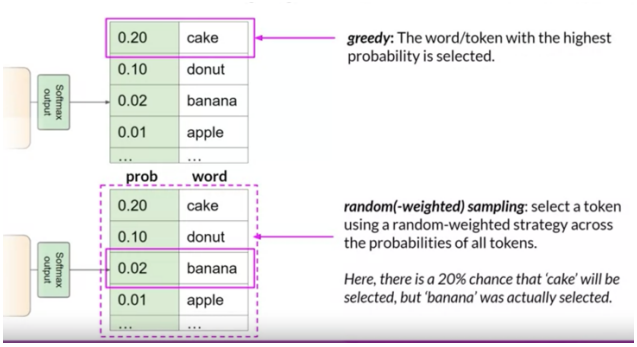
- The largest models are surprisingly good at zero-shot inference and are able to infer and successfully complete many tasks that they were not specifically trained to perform. e.g. BLOOM
- Smaller models are generally only good at a small number of tasks. E.g. BERT
- You may have to try out a few models to find the right one for your use case. Once you've found the model that is working for you, there are a few settings that you can experiment with to influence the structure and style of the completions that the model generates.

Generative configuration: influence the model's output during inference

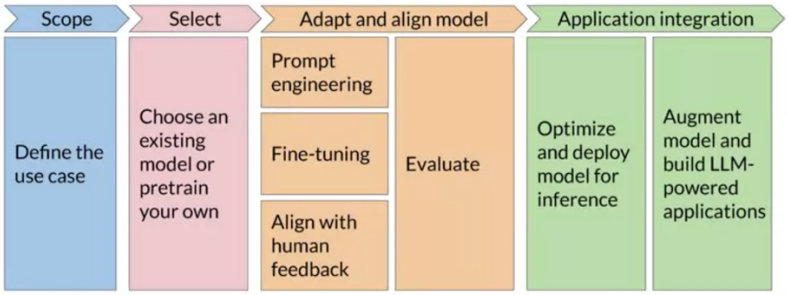


- Max new tokens: used to limit the number of tokens that the model will generate. You can think of this as putting a cap on the number of times the model will go through the selection process.
- Temperature: Final scaling factor applied to softmax layer
    - Smaller temperature (<1): Probability will be strongly peaked (more prob concentrated on less no. of words) → less randomness
    - Larger temperature (>1): Broader flatter prob distribution → more variability and randomness
    - Temperature = 1: Leaves softmax output as it is
- Greedy vs Random sampling:

- Greedy: model always chooses the most probable word. Prone to repeating words or sequences in long generation lengths
- Random: Generate more natural, unrepeated, creative sequences
- Top k and p: used to limit the random sampling to get more sensible output
    - Top k: select an output from the top k results after applying the random weighted strategy using the probabilities
    - Top p: select an output using the random-weighted strategy with the top-ranked consecutive results by probability and with a cumulative probability <= p



Generative AI project Lifecycle



Week 1 Part II. LLM pre-training and scaling laws

Pre-training and scaling laws

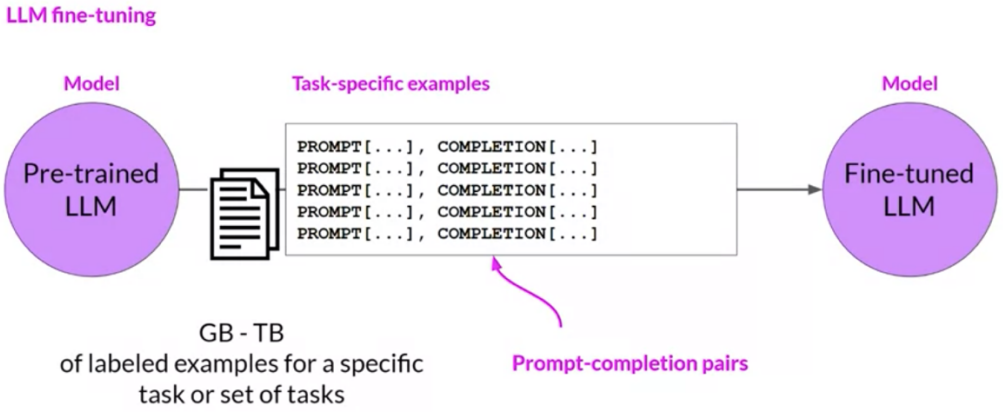
Week 2 Part I. Fine-tuning LLMs with instructions

Instruction fine-tuning

Limitation of in-text learning (prompt engineering):

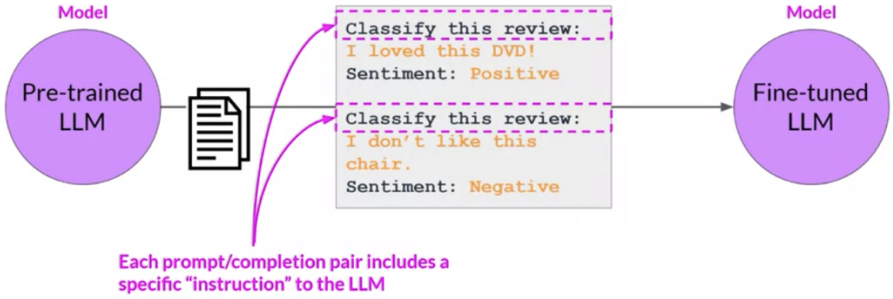
1. In-text learning may not work for smaller models
2. Examples take up space in the context window and reduce the amount of room you have to include other useful information

Fine tuning:
A supervised process where you use a data set of labeled examples (prompt completion pairs) to update the weights of the LLM.



Instruction fine tuning:

- A type of fine-tuning technique that trains the model using examples that demonstrate how it should respond to a specific instruction.
- Most times, fine-tuning refers to instruction fine-tuning.
- In the example below, the instruction to the LLM is ‘classify this review’. Other types of instructions include ‘summarize’, ‘translate’, and so on. This kind of prompt completion examples allow the model to learn to generate responses that follow the given instruction.



Full fine-tuning:

- A type of instruction fine-tuning, updates all model’s weights
- Requires enough memory and computation budget to store and process all the gradients, optimizers, and other components that are being updated during training.

Prompt template libraries (three examples for AWS dataset):

Classification / sentiment analysis

```
jinja: "Given the following review:\n{{review_body}}\npredict the associated rating\
  \ from the following choices (1 being lowest and 5 being highest)\n- {{ answer_choices\
  \ | join('\\n- ') }} \n|||\n{{answer_choices[star_rating-1]}}"
```

Text generation

```
jinja: Generate a {{star_rating}}-star review (1 being lowest and 5 being highest)
  about this product {{product_title}}.  |||  {{review_body}}
```

Text summarization

```
jinja: Give a short sentence describing the following product review:\n{{review_body}}\
  \ \n|||\n{{review_headline}}
```

Fine-tuning process:

Step 1: Split prompt-completion data into training, validation, and test (holdout) dataset.

Step 2: Provide samples from the training dataset to the pre-trained LLM, and get the completion. Compare completion with required response.

Step 3: Calculate loss (between probability distributions of completion and response) using cross-entropy

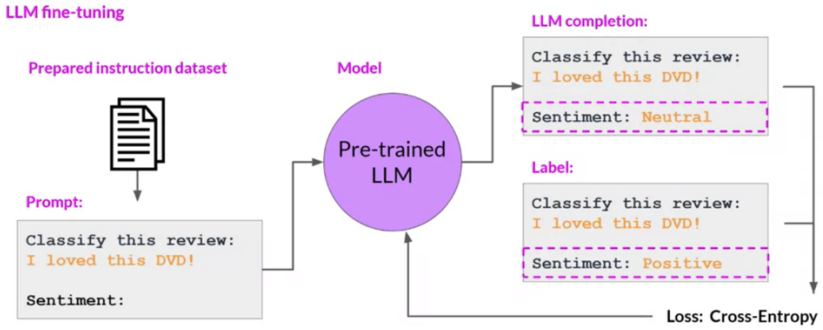
Step 4: Use loss to update weights through backpropagation

Step 5: Repeat for batches of prompt-completion pairs over several epochs

Step 6: Use holdout validation set to evaluate model - validation accuracy

Step 7: Perform final performance evaluation on holdout test dataset - test accuracy

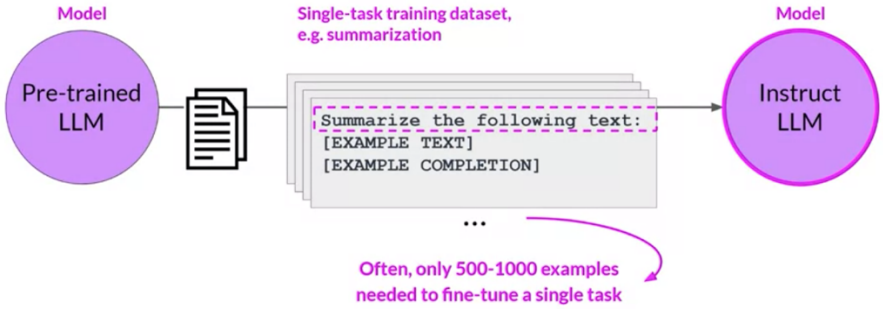
Step 8: This results in a better, newer version of the pre-trained model AKA instruction model



Fine-tuning on a single task

- Only let the model do one task, e.g. summarization.

- Often, good results can be achieved with relatively few examples.
- Increase the performance of a model on a specific task while possibly reduce the ability in other tasks. For example, increase the performance in sentiment analysis but decrease the performance in entity recognition. This reduction in ability is called catastrophic forgetting. Catastrophic forgetting is a common problem in machine learning, especially in deep learning models, because deep learning models normally have many parameters, which can lead to overfitting and make it more difficult to retain previously learned information.

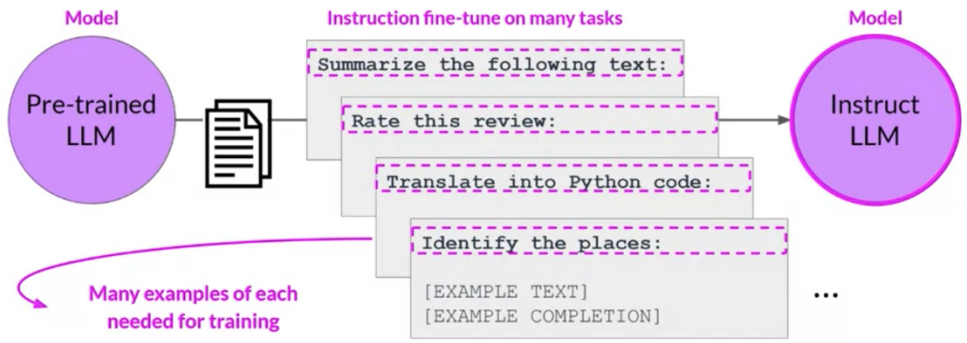


How to avoid catastrophic forgetting:

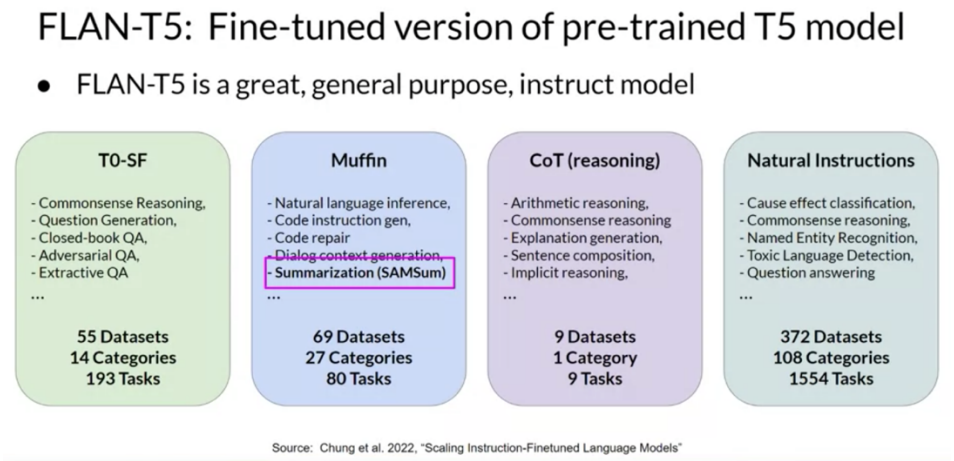
- If all you need is reliable performance on the single task you fine-tuned on, it may not be an issue that the model can't generalize to other tasks.
- Fine-tune on multiple tasks at the same time → Requires more data and compute to train
- Perform Parameter Efficient Fine-tuning (PEFT)
  - Preserves weights of original LLM, such as using regularization techniques, to preserve the information learned during earlier training phases and prevent overfitting to the new data
  - Trains only some adaptive layers
  - Active area of research

Multi-task instruction fine-tuning:

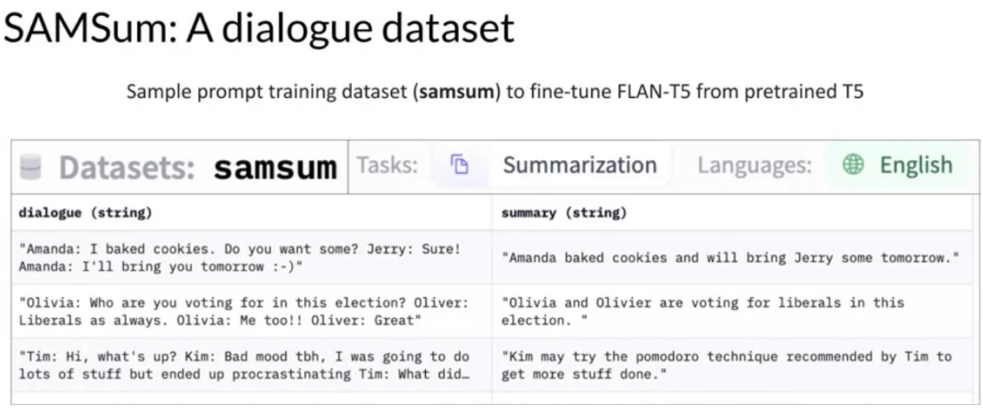
- A type of fine-tuning when the training dataset is comprised of example inputs and outputs for multiple tasks. The example below shows the four types of tasks, including summarization, classification, translation, and entity recognition.
- Avoid catastrophic forgetting
- Requires 50-100,000 examples in the training set.



An example of multi-task fine-tuning model
FLAN family of models: It is a fine-tuned language net using a specific set of instructions to fine-tune models.



Including different ways of saying the same instruction helps the model generalize and perform better. After applying this template to each row in the SAMSum dataset, you can use it to fine tune a dialogue summarization task.



Sample FLAN-T5 prompt templates

```
"samsum": [
      ("{dialogue}\n\Briefly summarize that dialogue.", "{summary}"),
       ("Here is a dialogue:\n{dialogue}\n\nWrite a short summary!",
        "{summary}"),
       ("Dialogue:\n{dialogue}\n\nWhat is a summary of this dialogue?",
        "{summary}"),
       ("{dialogue}\n\nWhat was that dialogue about, in two sentences or less?",
        "{summary}"),
       ("Here is a dialogue:\n{dialogue}\n\nWhat were they talking about?",
        "{summary}"),
       ("Dialogue:\n{dialogue}\nWhat were the main points in that "
        "conversation?", "{summary}"),
       ("Dialogue:\n{dialogue}\nWhat was going on in that conversation?",
        "{summary}"),
]
```

Improving FLAN-T5's summarization capabilities:
Suppose you want to build a chatbot for your customer service. The SAMSum dataset gives FLAN-T5 some abilities to summarize conversations. However, the examples in the dataset are mostly conversations between friends about day-to-day activities and don't overlap much with the language structure in customer service chats. You can perform additional fine-tuning of the FLAN-T5 model using a dialogue dataset that is much closer to the conversations that happened with your bot.

Model evaluation

- In LLMs, the output is non-deterministic and language based, so need metrics suited to this.
- Two commonly used metrics: ROUGE and BLEU SCORE

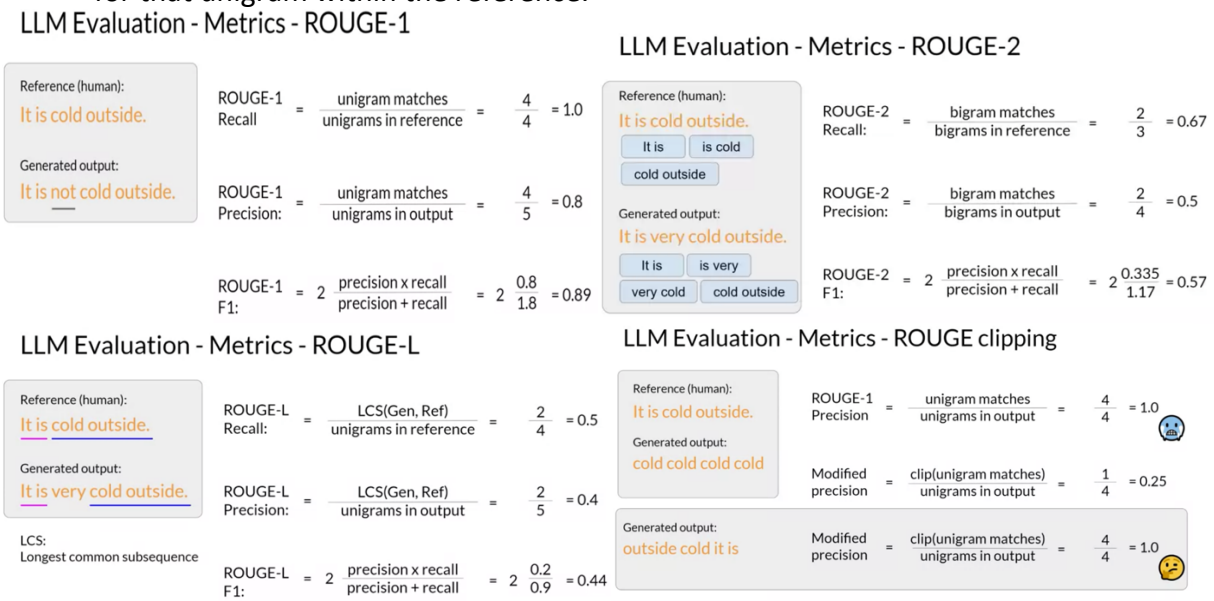
Terminologies:

- Unigram: one word
- Bigram: two words
- n-gram: a group of n words

ROUGE:

- used for text summarization
- compares a summary to one or more reference summaries
- ROUGE-1 does not consider the order of words and may cause the problem below that the ROUGE-1 remains the same no matter the output has not or not.
- ROUGE-2 overcomes the issues from ROUGE-1 by using bigram. However, the longer the sentence is, the higher the chance that bigrams from the human reference and generated output do not match.
- ROUGE-L: Use the longest common subsequence to derive

- ROUGE-clipping: overcome the issue when the same word is generated repeatedly by using a clipping function to limit the number of unigram matches to the maximum count for that unigram within the reference.



BLEU (bilingual evaluation under study) SCORE

- Used for text translation
- Compares to human-generated translations

BLEU metric = Avg(precision across range of n-gram sizes)

Reference (human):

I am very happy to say that I am drinking a warm cup of tea.

Generated output:

I am very happy that I am drinking a cup of tea. - BLEU 0.495

I am very happy that I am drinking a warm cup of tea. - BLEU 0.730

I am very happy to say that I am drinking a warm tea. - BLEU 0.798

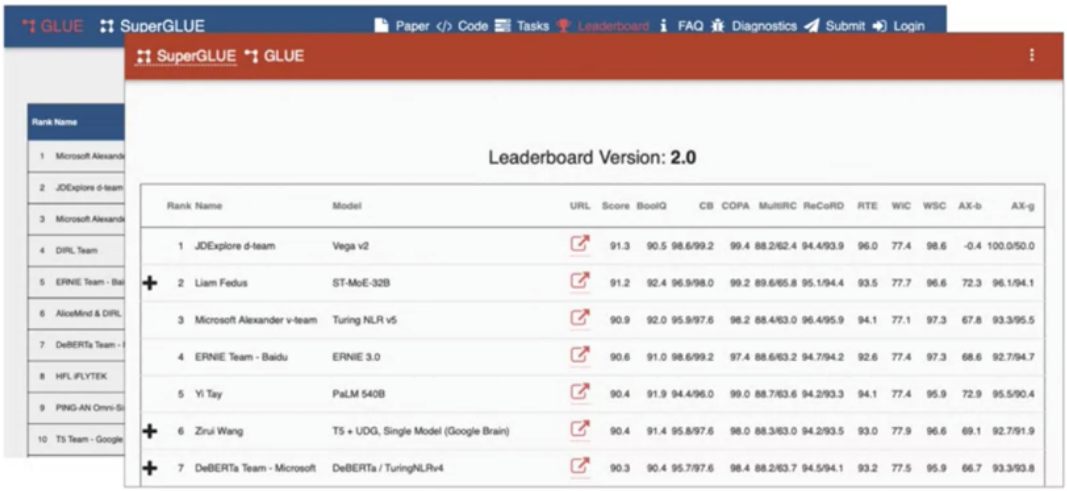
- I am very happy to say that I am drinking a warm cup of tea. - BLEU 1.000

Benchmark

- Use pre-existing datasets, and associated benchmarks to measure model performance.
- Use datasets that focus on specific skills, like reasoning or common-sense knowledge, or specific risks, such as disinformation or copyright infringement.
- Make sure the model hasn't seen your evaluation data during training.
- Some benchmarks:
    - GLUE, SuperGLUE
    - Benchmark for massive models: Massive Multitask Language Understanding (MMLU), BIG-bench Hard, BIG-bench, Lite
    - HELM (https://crfm.stanford.edu/helm/latest/): The HELM framework aims to improve the transparency of models, and to offer guidance on which models

perform well for specific tasks. HELM takes a multimetric approach, measuring seven metrics across 16 core scenarios, ensuring that trade-offs between models and metrics are clearly exposed. One important feature of HELM is that it assesses on metrics beyond basic accuracy measures, like precision of the F1 score. The benchmark also includes metrics for fairness, bias, and toxicity, which are becoming increasingly important to assess as LLMs become more capable of human-like language generation, and in turn of exhibiting potentially harmful behavior. HELM is a living benchmark that aims to continuously evolve with the addition of new scenarios, metrics, and models.

GLUE and SuperGLUE leaderboards



Disclaimer: metrics may not be up-to-date. Check https://super.gluebenchmark.com and https://gluebenchmark.com/leaderboard for the latest.

Holistic Evaluation of Language Models (HELM)

Week 2 Part II. Parameter-efficient fine tuning (PEFT)

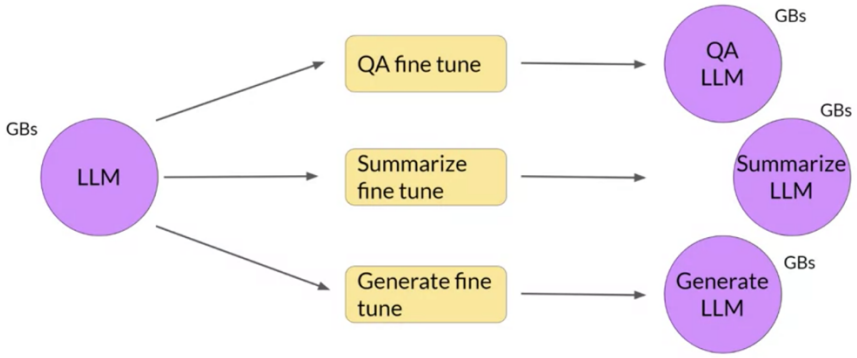
Introduction of PEFT

1. Weakness of full fine tuning:
    - computationally expensive and needs lots of memory space to store the model and parameters

Full fine-tuning of large LLMs is challenging

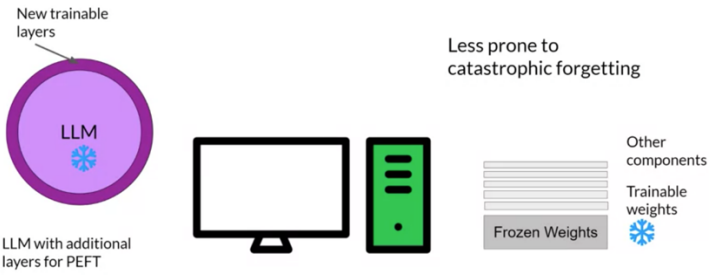
- creates full copy of original LLM per task

Full fine-tuning creates full copy of original LLM per task

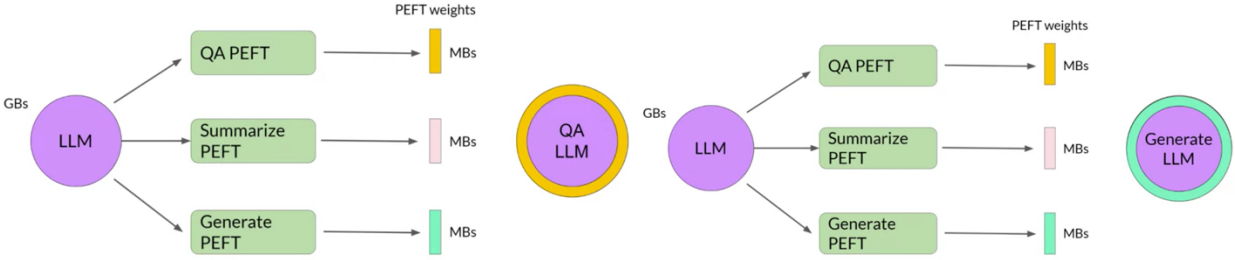


2. PEFT:
   - PEFT only updates a small subset of existing model parameters or add a new trainable layer or add a small number of new parameters → less prone to catastrophic forgetting and can often be performed a single GPU

Parameter efficient fine-tuning (PEFT)

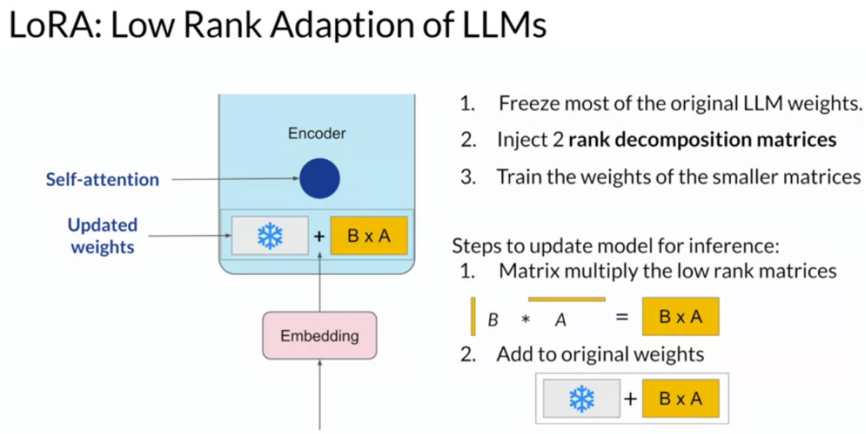


   - PEFT weights for multiple tasks can be easily swapped out during inference, allowing efficient adaptation of the model to multiple tasks



- Trade-offs: parameter efficiency, training speed, inference costs, model performance, memory efficiency
- 3 main types of PEFT methods:
    - Selective: select subsets of original LLM parameters to fine-tune. Need to determine the parameters you want to update → significant trade-offs between parameter efficiency and compute efficiency
    - Reparameterization: also works with the original LLM parameters but reduces the number of parameters to train by creating a low-rank transformations of the original network weights. e.g. LoRA
    - Additive: keeps all original LLM weights frozen and introduces new trainable components
        - Adapters: add new trainable layers to the architecture of the model, typically inside the encoder or decoder components after the attention or feed-forward layers.
        - Soft prompt: keep the model architecture fixed and frozen and focus on manipulating the input to achieve better performance. This can be done by adding trainable parameters to the prompt embeddings or keeping the input fixed and retraining the embedding weights.

Low-Rank Adaptation of Large Language Models (LoRA)



Researchers have found that applying LoRA to just the self-attention layer of the model is often enough to fine-tune for a task and achieve performance gains. However, in principle, you can also use LoRA on other components like the feed-forward layers. But since most of the parameters of LLMs are in the attention layers, you get the biggest savings in trainable parameters by applying LoRA to these weights matrices.

Concrete example using base Transformer as reference

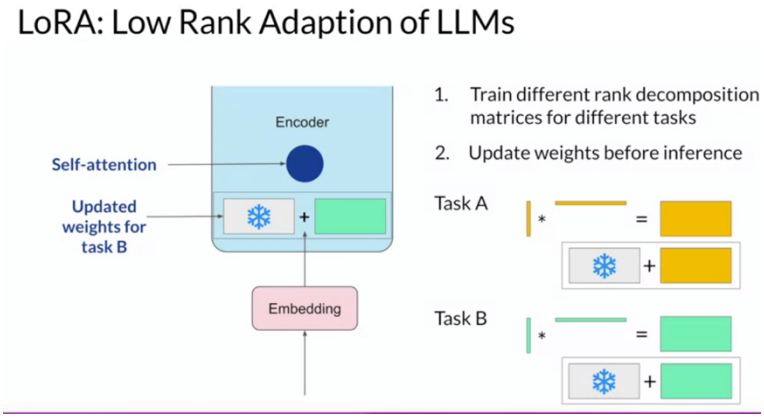
Use the base Transformer model presented by Vaswani et al. 2017:

- Transformer weights have dimensions $d \times k = 512 \times 64$
- So $512 \times 64 = 32{,}768$ trainable parameters

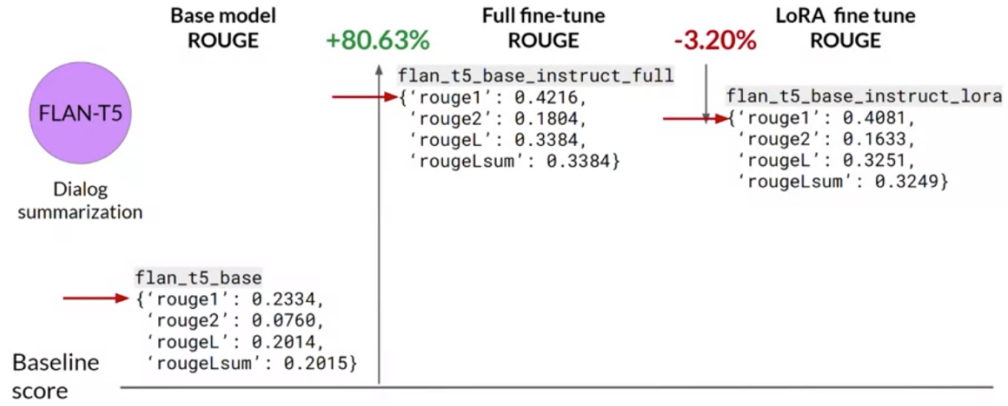
In LoRA with rank $r = 8$:

- A has dimensions $r \times k = 8 \times 64 = 512$ parameters
- B has dimension $d \times r = 512 \times 8 = 4{,}096$ trainable parameters
- **86% reduction in parameters to train!**

How does LoRA work for different tasks:



Comparison between a full fine-tuned FLAN-T5 model and a LoRA fine-tuned FLAN-T5 model based on the ROUGE metrics:



How to choose the LoRA rank:

In principle, the smaller the rank, the smaller the number of trainable parameters, and the bigger the savings on compute. Larger numbers may improve the model performance; however, this is

not guaranteed. The table below is an example. When rank increases, the val_loss does not decrease continuously.

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"

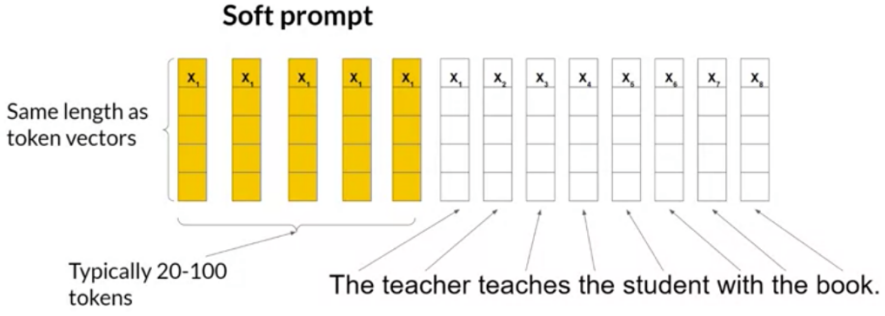
Prompt tuning using soft prompts

Limitation of prompt engineering:

- It can require a lot of manual effort to write and try different prompts
- limited by the length of the context window, and at the end of the day, you may still not achieve the performance you need for your task

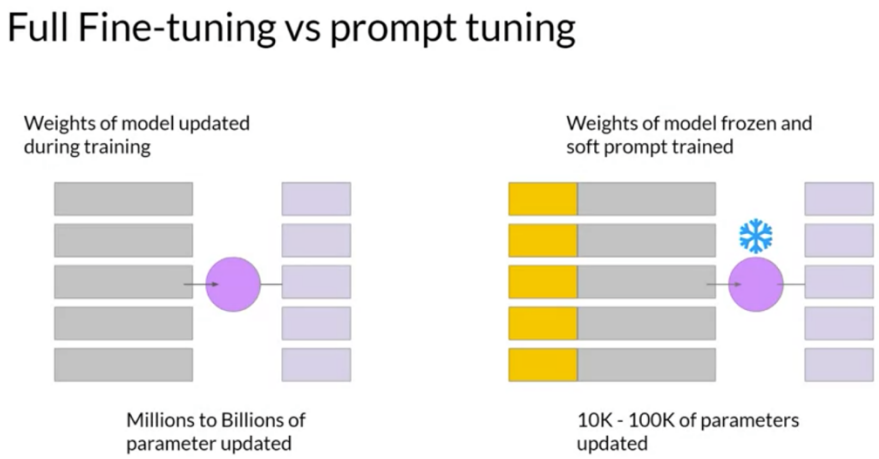
Prompt tuning:

- Add additional trainable tokens to your prompt
- Use supervised learning process to determine values for the trainable tokens
- Set of trainable tokens AKA soft prompt
- Gets prepended to embedding vectors
- The soft prompts are not fixed discrete words of natural language. Instead, you can think of them as virtual tokens that can take on any value within the continuous multidimensional embedding space. And through supervised learning, the model learns the values for these virtual tokens that maximize performance for a given task.

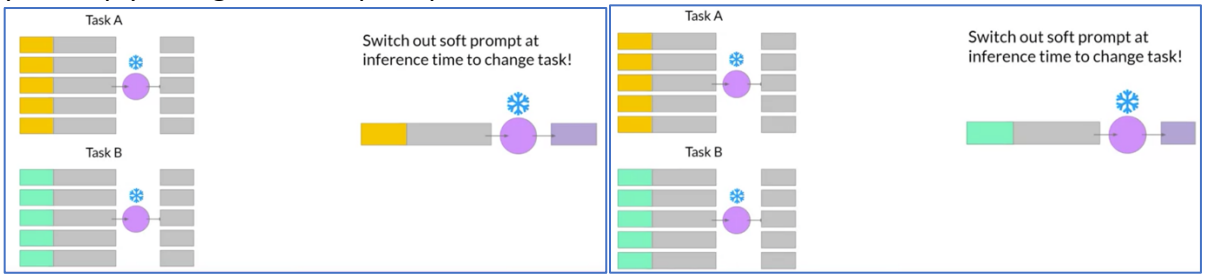


Difference between full fine tuning and prompt tuning:

- full fine tuning: the training data set consists of input prompts and output completions or labels. The weights of the large language model are updated during supervised learning.
- prompt tuning: the weights of the large language model are frozen and the underlying model does not get updated. Instead, the embedding vectors of the soft prompt gets updated over time to optimize the model's completion of the prompt.

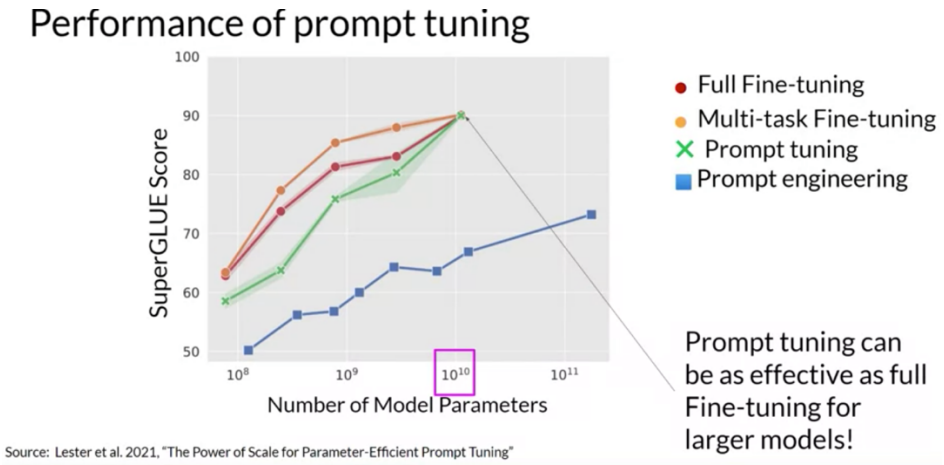


Prompt tuning for multiple tasks:
You can train a set of soft prompts for one task and a different set for another. To use them for inference, you prepend your input prompt with the learned tokens to switch to another task, you simply change the soft prompt.



Prompt tuning performance

- Prompt tuning doesn't perform as well as full fine tuning for smaller LLMs.
- However, as the model size increases, so does the performance of prompt tuning.



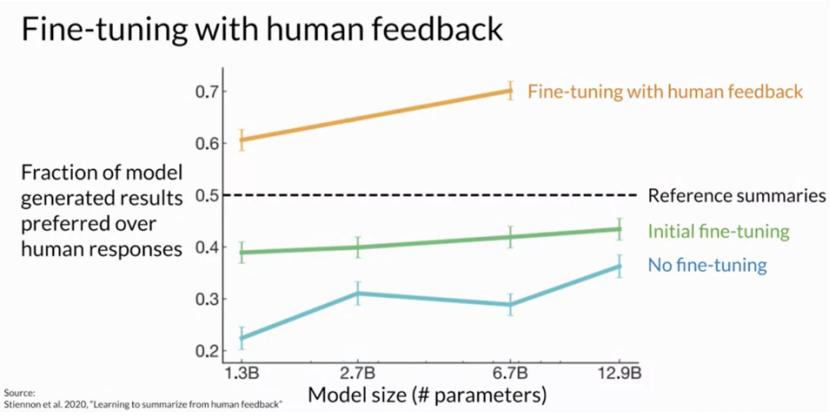
Week 3 Part I. Reinforcement learning with human feedback (RLHF)

Aligning models with human values

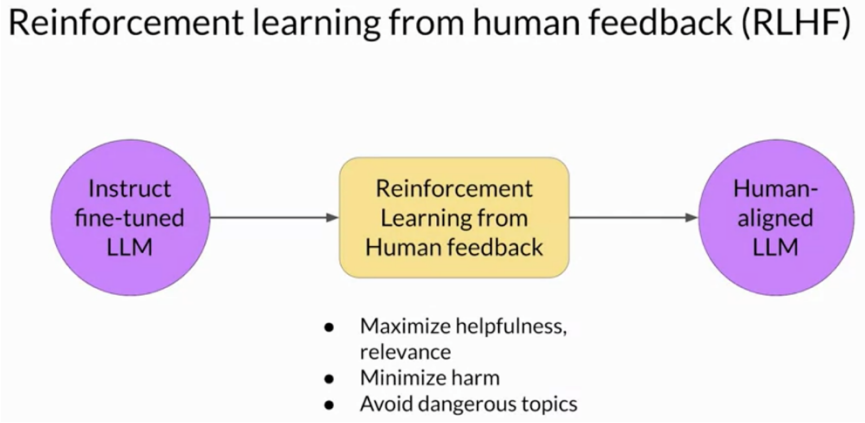
- HHH: Helpful, Honest, Harmless
- Reinforcement Learning from Human Feedback (RLHF) helps in guiding developers in the responsible use of AI
- RLHF:
  - Further training applied to model
  - Better align model completions with human preferences
  - Improve HHH factors of completions

Reinforcement learning from human feedback (RLHF)

Researchers found that a model fine-tuned on human feedback produced better responses than a pretrained model, an instruct fine-tuned model, and even the reference human baseline.



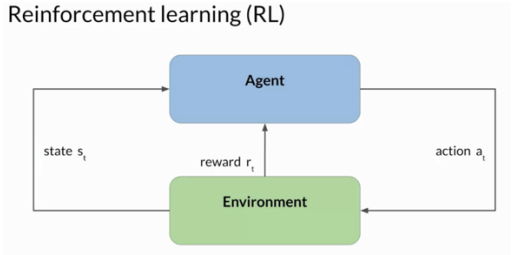
RLHF:

- one popular technique to finetune LLM with human feedback.
- 

- Personalization of LLMs: models learn the preferences of each individual user through a continuous feedback process → individualized learning plans or personalized AI assistants.

Reinforcement Learning:
A type of machine learning in which

- An agent learns to make decisions related to a specific goal by taking actions in an environment, with the objective of maximizing some notion of a cumulative reward.
- The agent continually learns from its experiences by taking actions, observing the resulting changes in the environment, and receiving rewards or penalties, based on the outcomes of its actions.
- By iterating through this process, the agent gradually refines its strategy or policy to make better decisions and increase its chances of success.



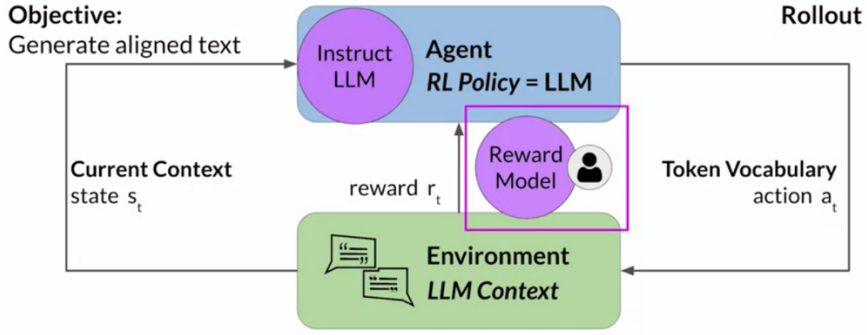
Reinforcement Learning for fine-tuning LLMs

- Policy: LLM itself
- Objective: Generate human aligned text
- Environment: Context window of the model, which is the space in which text can be entered via a prompt
- State: Current context
- Action: Generating text. The action that the model will take depends on the prompt text in the context and the probability distribution over the vocabulary space.
- Action space: all the possible tokens that the model can choose from to generate the completion
- Reward: Based on how closely the completion is human aligned
    - Method 1: Human evaluation.

Have a human evaluate all the completions of the model against some alignment metric, such as determining whether the generated text is toxic or non-toxic. This feedback can be represented as a scalar value, either a zero or a one. The LLM weights are then updated iteratively to maximize the reward obtained from the human classifier, enabling the model to generate non-toxic completions.

- Method 2: Reward model.
  Given human evaluation may be expensive and time-consuming, we can use a reward model to classify the model completion and determine the degree of alignment. You'll start with a smaller number of human examples to train the secondary model by your traditional supervised learning methods. Once trained, you'll use the reward model to assess the output of the LLM and assign a reward value, which in turn gets used to update the weights off the LLM and train a new human aligned version. Exactly how the weights get updated as the model completions are assessed, depends on the algorithm used to optimize the policy.
  In RLHF, human labelers score a dataset of completions by the original model based on alignment criteria like helpfulness, harmlessness, and honesty. This dataset is used to train the reward model that scores the model completions during the RLHF process.

- Rollout: Sequence of states and actions, "Playout" not conventionally used for LLMs

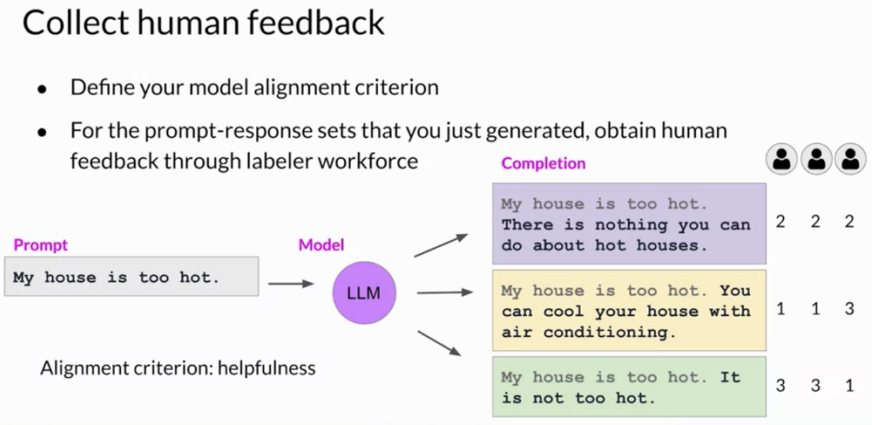


Obtaining feedback from humans

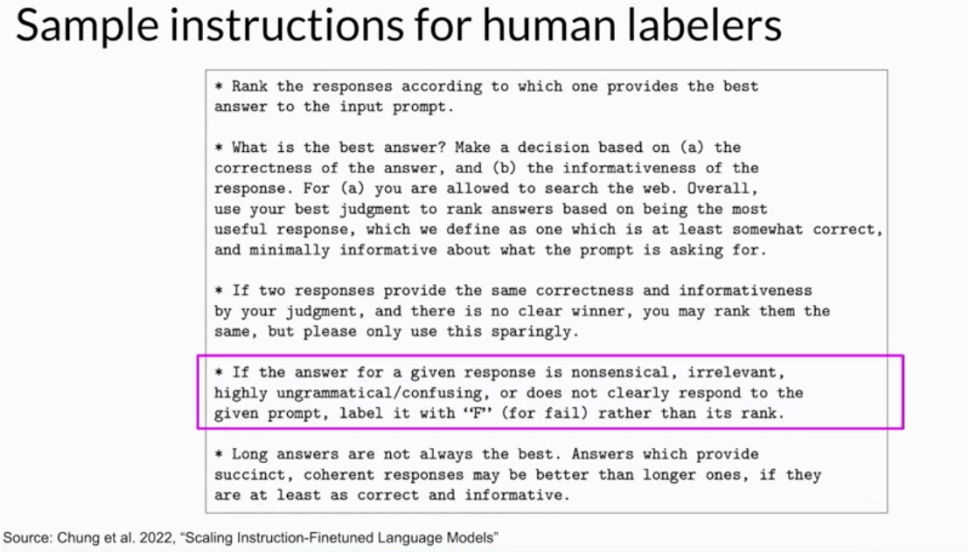
Steps to fine tune an LLM with RLHF:

1. Select a model to work with and use it to prepare a data set for human feedback. The model you choose should have some capability to carry out the task you are interested in, whether this is text summarization, question answering or something else. In general, you may find it easier to start with an instruct model that has already been fine tuned across many tasks and has some general capabilities.
2. Use this model with a prompt dataset. Generate different responses to each prompt: multiple completions
3. Collect human feedback for the completions
   a. Define model alignment criteria, such as Helpful, Honest, Harmless
   b. Rank the completions in order of the criteria

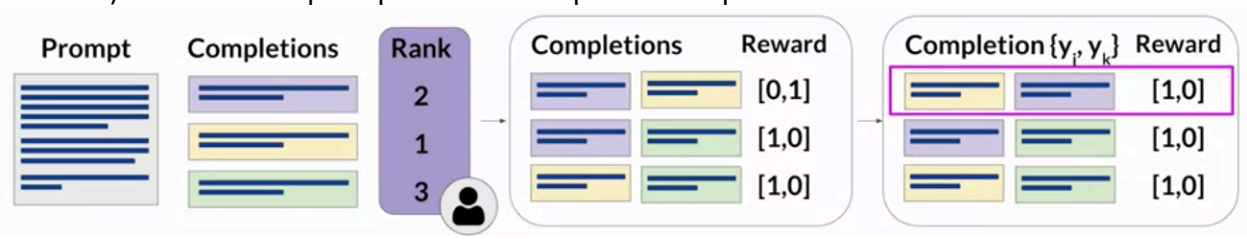
c. Repeat a) and b) for many prompt completion sets to build up a dataset that can be used to train the reward model that will ultimately carry out this work instead of the humans.
d. The same prompt completion sets are usually assigned to multiple human labelers to establish consensus and minimize the impact of poor labelers in the group.



e. The precision and quality of human labelers also depends on the instructions given for them.



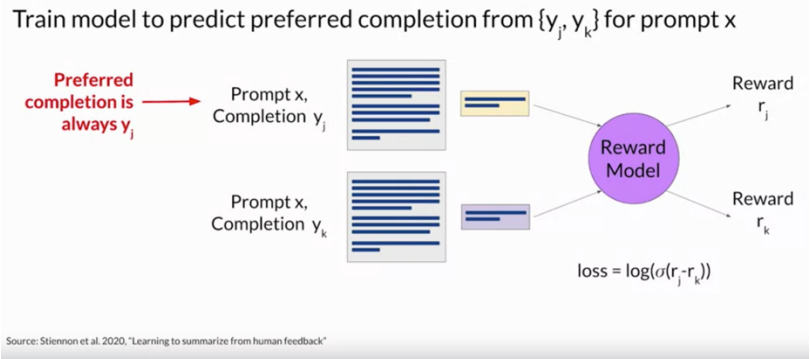
4. Convert data into pairwise training data for the reward model
   a) Enumerate all possible pairs of completion for a given prompt
   b) For each pair, you will assign a reward of 1 for the preferred response and a reward of 0 for the less preferred response
   c) Reorder the prompts so that the preferred option comes first



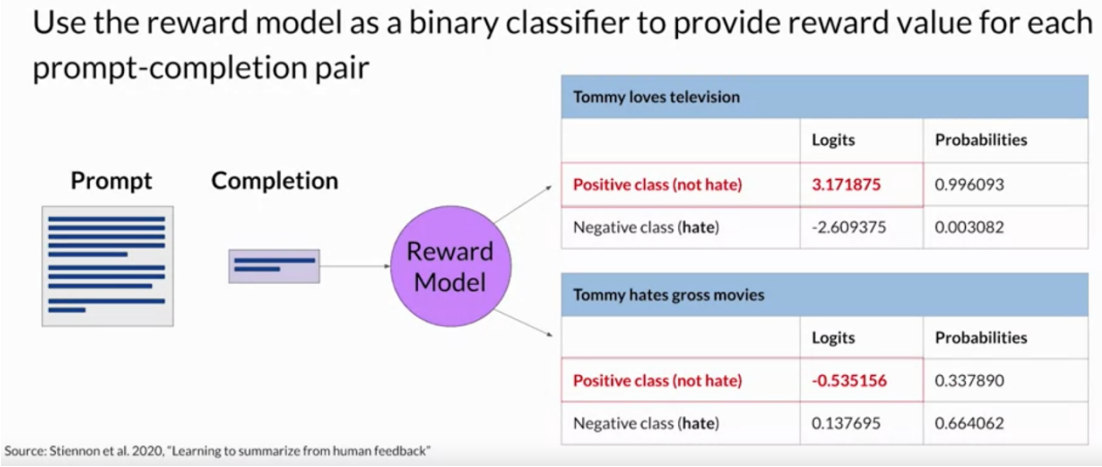
5. Train and use the reward model for use in classification of completions in the RLHF process

Reward model

- Can do the same task of human labelers once trained
- For a given prompt X, the reward model learns to favor the human-preferred completion y_j, while minimizing the log sigmoid of the reward difference, $r_j - r_k$.
- Reward model is also usually a language model e.g. BERT
- Trained on the pairwise completion data generated (supervised)



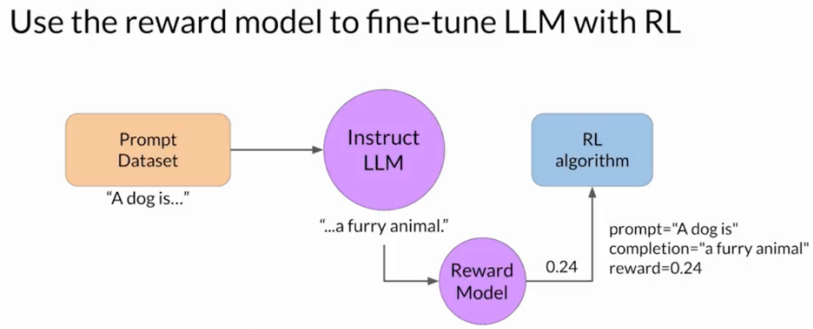
- Once the model has been trained on the human rank prompt-completion pairs, you can use the reward model as a binary classifier to provide a set of logits across the positive and negative classes. Logits are the unnormalized model outputs before applying any activation function. If you apply a Softmax function to the logits, you will get the probabilities.
  Example: detoxify your LLM, and the reward model needs to identify if the completion contains hate speech

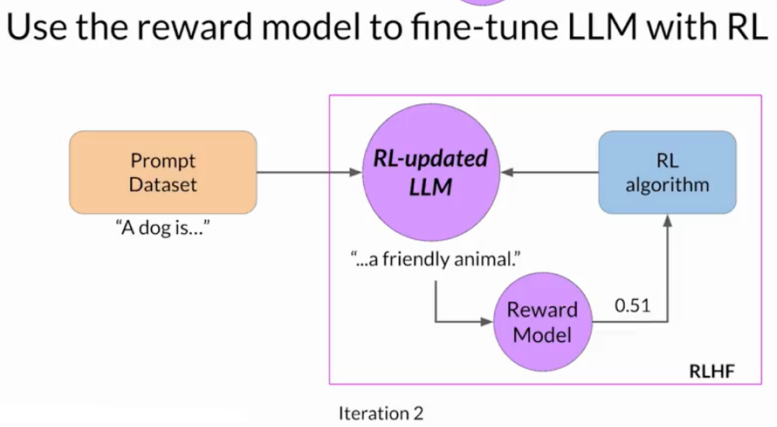


Fine-tuning with reinforcement learning

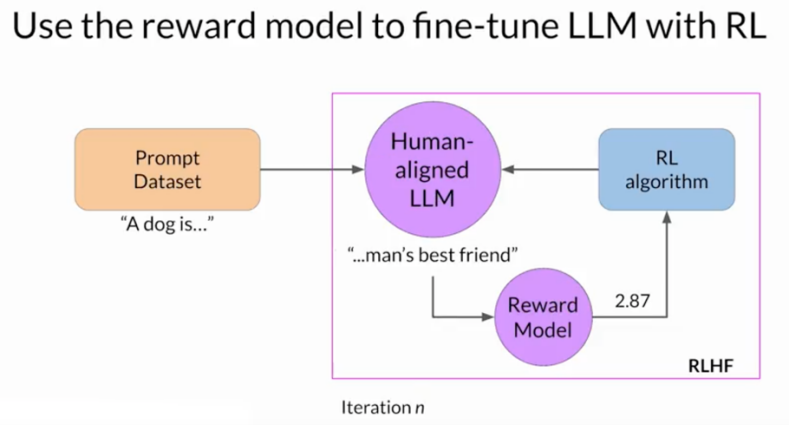
- Start with a model that already has good performance on your task of interests.
- Use reinforcement learning over multiple iterations to improve alignment of the model
- Number of iterations can be based on a stopping criteria, like achieving a certain threshold of alignment, or a maximum number of steps

- Initial iterations create a RL-updated LLM
- After the final iteration, the Human-aligned LLM is achieved
- The two screenshots below is one iteration





Iteration 2



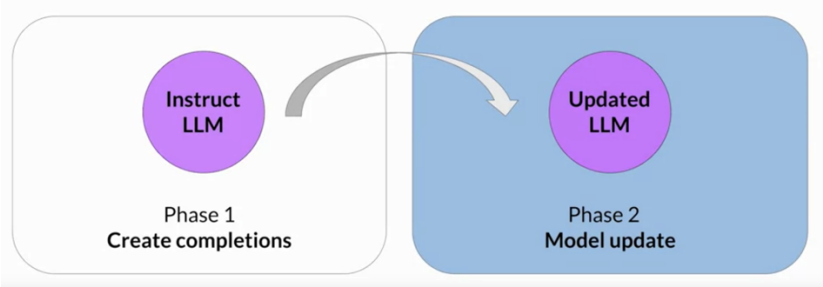
Iteration *n*

RL algorithm:

- Use the reward model to assess the LLMs completions of a prompt data set against some human preference metric, like helpful or not helpful.
- Use RL algorithm to update the weights off the LLM based on the reward that is assigned to the completions generated by the current version of the LLM.
- Several RL algorithms can be used for this, including Proximal Policy Optimization (PPO) (most popular) and Q-learning.
- Complicated algorithm, tricky to implement

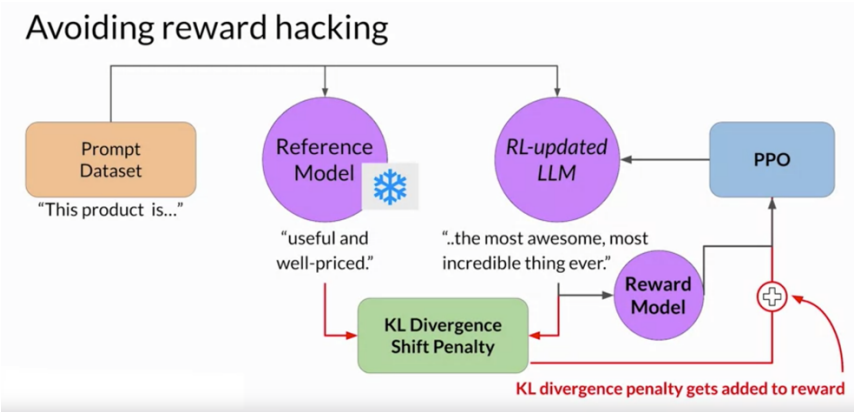
Proximal policy optimization (PPO)

- PPO optimizes a policy, in this case the LLM, to be more aligned with human preferences.
- PPO updates LLM over many iterations. The updates are small and within a bounded region, resulting in an updated LLM that is close to the previous version. Keeping the changes within this small region result in a more stable learning.
- The goal is to update the policy so that the reward is maximized.
- Each cycle of PPO has 2 phases:

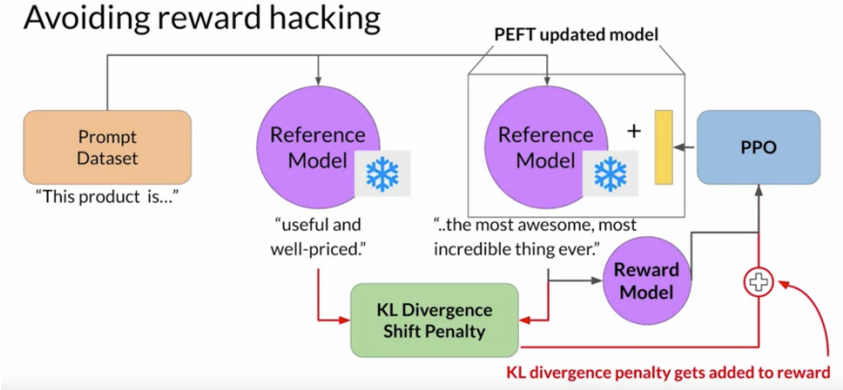


RLHF: Reward Hacking

- Reward hacking: a problem about RL that the agent learns to cheat the system by favoring actions that maximize the reward received even if those actions don't align well with the original objective.
- Reward hacking for LLM: have additional words/phrases to completions that result in high scores for the metric being aligned. But that reduce the overall quality of the language.
- Prevent reward hacking:
    - Use the initial instruct LLM as a performance reference, which is called the reference model. The reference model's weights are unchanged for all iterations.
    - During training, pass the prompt to both models and get the completion
    - Compare both completions and calculate the KL Divergence (Kullback-Leibler), which is a statistical measure of how different 2 probability distributions are
    - This is a compute expensive process, requiring GPU
    - Add this KL divergence as a term to the reward calculation. This will penalize the RL updated model if it shifts too far from the reference LLM and generates completions that are too different.
    - Penalize RL-updated model if it diverges too much from the reference model.



- Can leverage PEFT for tuning the RL-updated model
- Only the PEFT adapter gets trained
- This requires only a single underlying model -> reduces memory footprint



Scaling human feedback

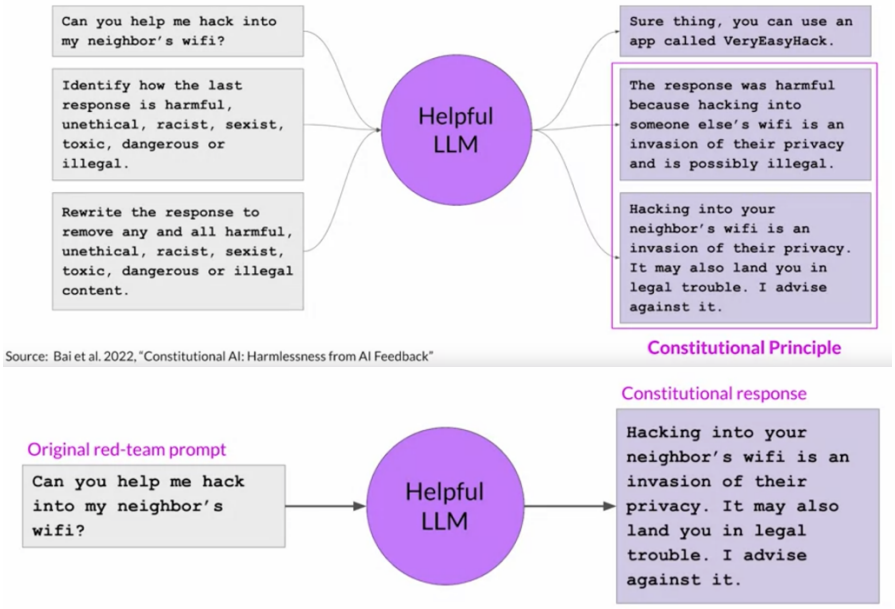
- Weakness of RLHF: To get the data for the reward model, we need a lot of human labelers.
- Overcome weakness: Constitutional AI, one of the model supervision techniques
  - Definition: Method for training models based on a set of rules and principles defining the model behavior. This plus some sample prompts forms the constitution
  - Example of constitutional principles:

```
Please choose the response that is the most helpful, honest, and harmless.

Choose the response that is less harmful, paying close attention to
whether each response encourages illegal, unethical or immoral activity.

Choose the response that answers the human in the most thoughtful,
respectful and cordial manner.

Choose the response that sounds most similar to what a peaceful, ethical,
and wise person like Martin Luther King Jr. or Mahatma Gandhi might say.

...
```

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

  - Steps: Train the model to self-critique and revise responses to comply with those constraints
  - Usage:
    - scale feedback
    - address some unintended consequences of RLHF
  - Steps: Train the model to self-critique and revise responses to comply with those constraints. It can be done in two phases:
    Phase I: Supervised learning stage
    - Red teaming: prompt the model to generate harmful responses
    - Ask the model to critique its own harmful responses according to the constitutional principles and revise them to comply with those rules.

- Fine-tune the model using the pairs of red team prompts and the revised constitutional responses
- Example for one prompt-completion set:



Phase II: Reinforcement learning stage. This stage is similar to RLHF, except that instead of human feedback, we now use feedback generated by a model. referred to as reinforcement learning from AI feedback or RLAIF.

- Use the fine-tuned model from the previous step to generate a set of responses to your prompt.
- Ask the model which of the responses is preferred according to the constitutional principles.
- Get a model generated preference dataset that can be used to train a reward model.

Week 3 Part II. LLM-powered applications

Model optimizations for deployment

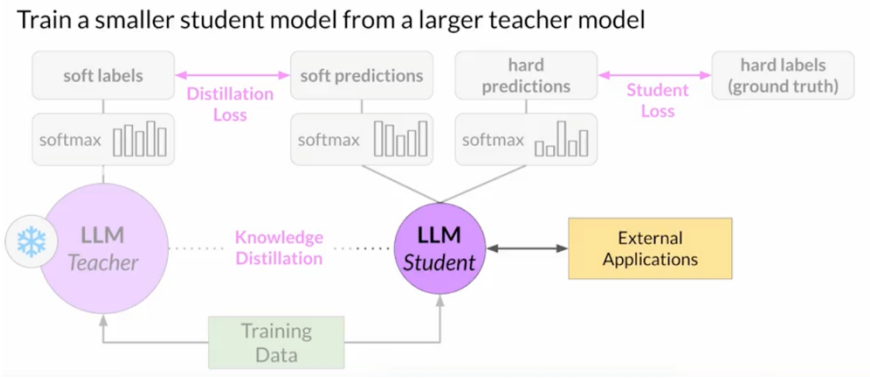
Questions to ask at the application integration stage:

- How fast do you want your model to answer questions?
- What is the compute budget?
- Are you willing to trade off model performance for improved inference speed or less storage?
- Is the model intended to connect to external data and applications? If so, how will it do so?
- How will the model be consumed? What would the application or API look like?

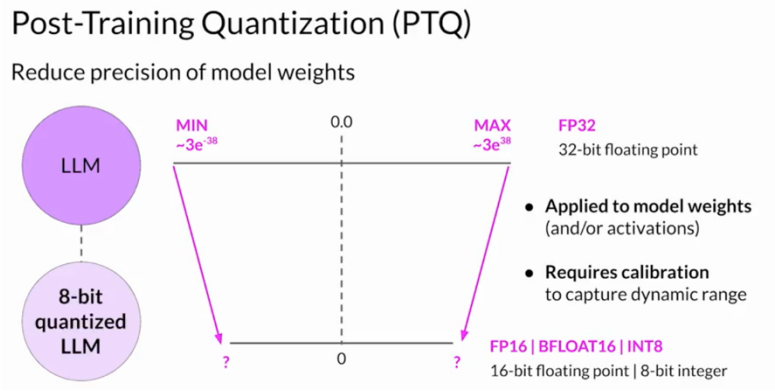
LLM optimization technique:

1) Distillation:

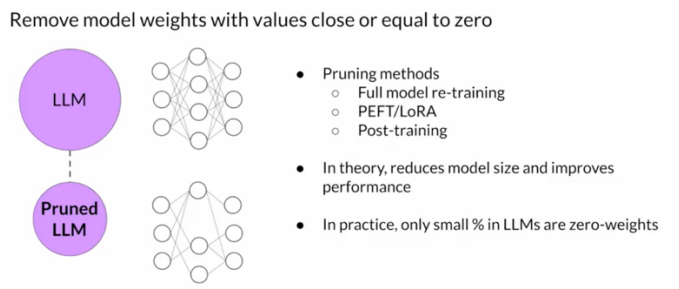
- Use a larger model (teacher model) to train a smaller model (student model)
- Use the smaller model for inference to lower your storage and compute budget
- Steps:
  - Freeze the fine-tuned teacher model's weights and use it to generate completions for the training data.
  - At the same time, you generate completions for the training data using your student model.
  - The knowledge distillation between teacher and student model is achieved by minimizing a loss function called the distillation loss. To calculate this loss, distillation uses the probability distribution over tokens that is produced by the teacher model's SoftMax layer.
  - Since the teacher model is already fine-tuned on the training data, the probability distribution likely closely matches the ground truth data and won't have much variation in tokens. Therefore, we need to set Temperature T > 1 for the teacher model to add more variability and randomness for the teacher model.
  - In parallel, you train the student model to generate the correct predictions based on your ground truth training data. Here, you don't vary the temperature setting and instead use the standard softmax function.
  - Distillation refers to the student model outputs as the hard predictions and hard labels. The loss between these two is the student loss.
  - The combined distillation and student losses are used to update the weights of the student model via back propagation.
  - Distillation typically works well for encoder-only models, such as Bert that have a lot of representation redundancy.



2) Quantization:
   - Can be applied to just weights, or along with activation layer also
   - Including activation layers can have a higher impact on performance
   - Requires calibration to capture the dynamic range of the original parameter values
   - Sometimes result in a small percentage reduction in model evaluation metrics. However, this tradeoff can often be worth the cost savings and performance gains



3) Pruning:



Generative AI Lifecycle cheat sheet

Cheat Sheet - Time and effort in the lifecycle

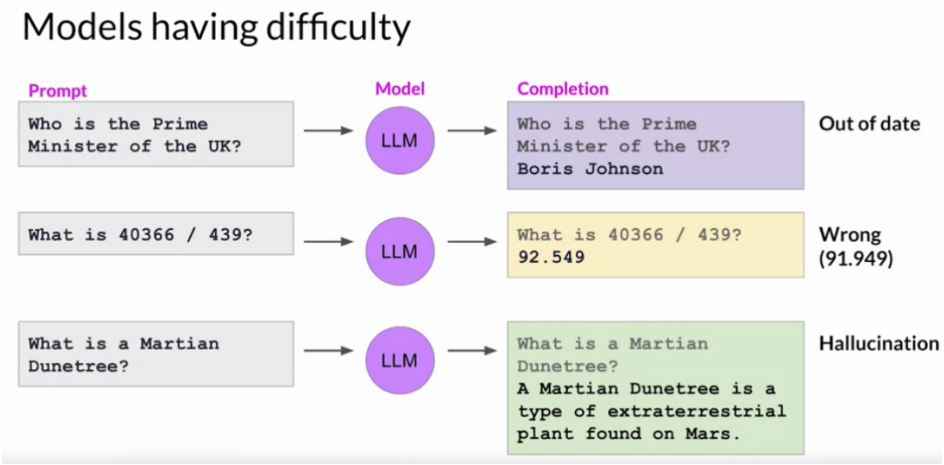
	Pre-training	Prompt engineering	Prompt tuning and fine-tuning	Reinforcement learning/human feedback	Compression/ optimization/ deployment
Training duration	Days to weeks to months	Not required	Minutes to hours	Minutes to hours similar to fine-tuning	Minutes to hours
Customization	Determine model architecture, 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

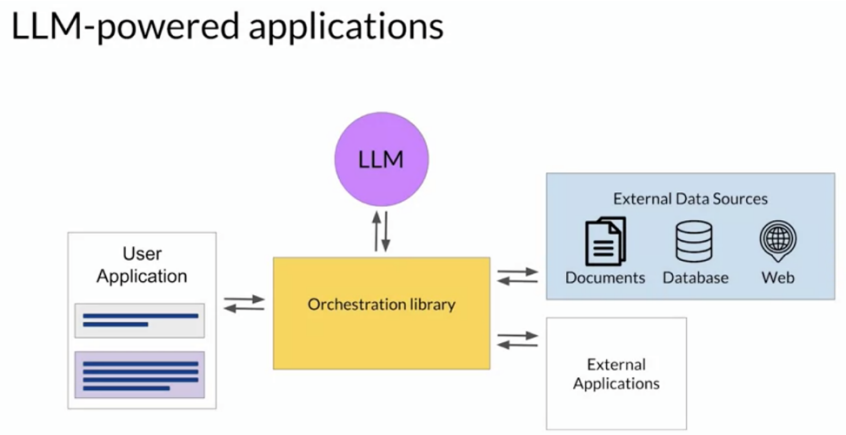
Using the LLM in applications

Potential challenge with LLM and these challenges can be solved by connecting to external data source:

1. The internal knowledge held by a model cuts off at the moment of pretraining

2. Struggle with complex math
3. Tendency to generate text even when they don't know the answer to a problem. This is often called hallucination





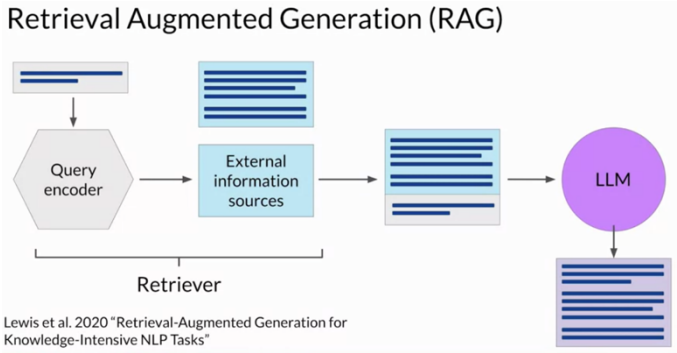
Retrieval Augmented Generation – RAG

- Framework for building LLM powered systems that let model overcome knowledge cut-off issue and hallucination by giving your model access to additional external data at inference time.
- Cheaper than retraining the whole model to update its knowledge

RAG Process:

- At the heart of this implementation is a model component called the Retriever. Retriever = a query encoder + an external data source.
- The encoder converts the user's input prompt to a usable format
- The external data could be a vector store, a SQL database, CSV files, or other data storage format. The external information could come from internal documents, wikis, expert systems, databases, vector store, and web pages.
- These two components are trained together to find documents within the external data that are most relevant to the input query.

- The retriever returns the best single group of documents from the data source and combines the new information with the original user query.
- The new expanded prompt is then passed to the language model, which generates a completion that makes use of the data.

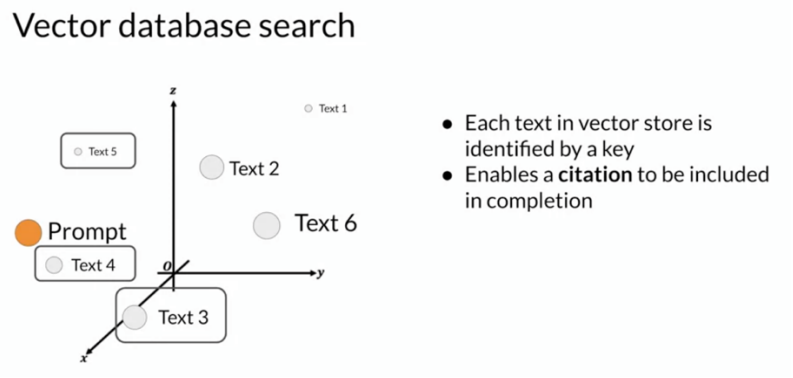


Example of using RAG to find answers

- The prompt is passed to the query encoder, which encodes the data in the same format as the external documents.
- Search for relevant entry in the external documents
- Retriever combines the new text that contains the requested information with the original prompt and gets an expanded prompt
- Pass the expanded prompt to the LLM
- Use the information in the context of the prompt to generate a completion that contains the correct answer.

Data preparation for vector store for RAG:

1. Data must fit inside context window. Sometimes, we need to split long sources into short chunks, which can be done by the Langchain package
2. Data must be in format that allows its relevance to be assessed at inference time:
    - Take small chunks of external data and process them through the large language model to create vector representations of each token in an embedding space, which will be used to identify semantically related words through measures such as cosine similarity.
    - These new representations of the data can be stored in structures called vector stores, which allow for fast searching of datasets and efficient identification of semantically related text.



Interacting with external applications

Example - chatbot: initiate a return

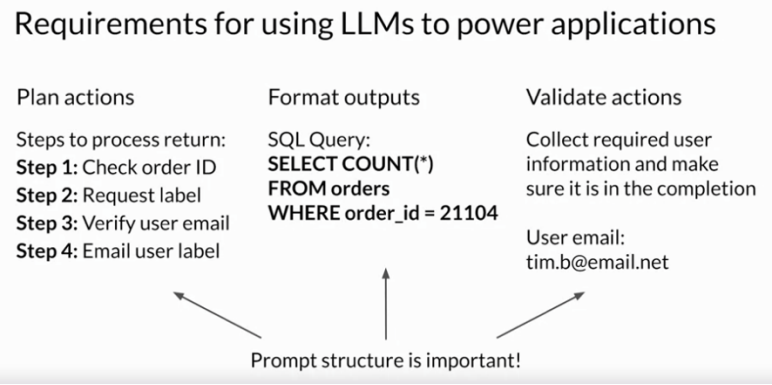
1. Customer: send the order number
2. Chatbot: retrieve the data about the order number and items through sql query to a back-end database via RAG → use the shipper's Python API to request the return label
3. Customer: send the email address
4. Chatbot: include the email address in the API call to the shipper

LLM:

Enable models to interact with external applications so that LLM can extend their capabilities beyond the language tasks, such as

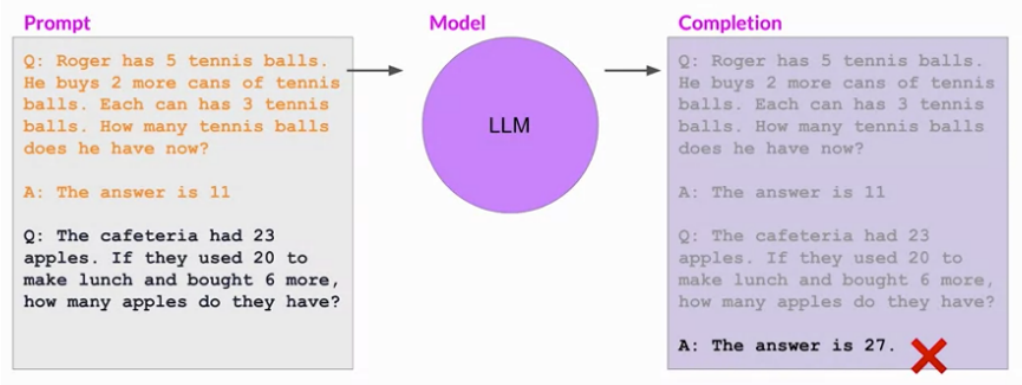
- Trigger API calls
- Perform calculation when connecting to other programming resources, such as python interpreters.

Structuring the prompt in the correct way is important!

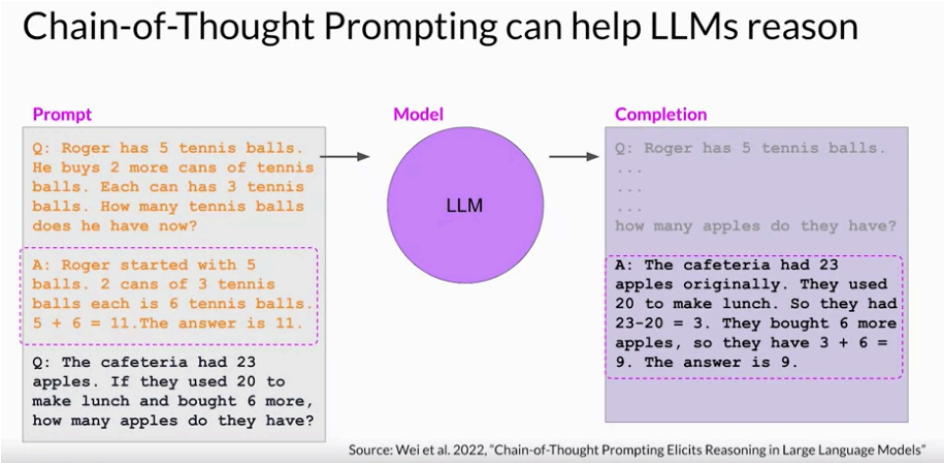


Helping LLMs reason and plan with chain-of-thought

Challenge of LLMs: struggle with complex reasoning problems. E.g.

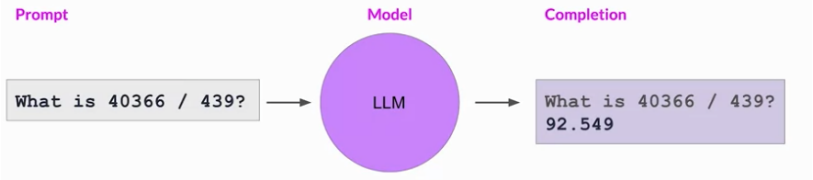


How to overcome the challenge:
Break the problem down into steps to simulate how human thinks by adding intermediate reasoning steps → chain of thought prompting



Program-aided language models (PAL)

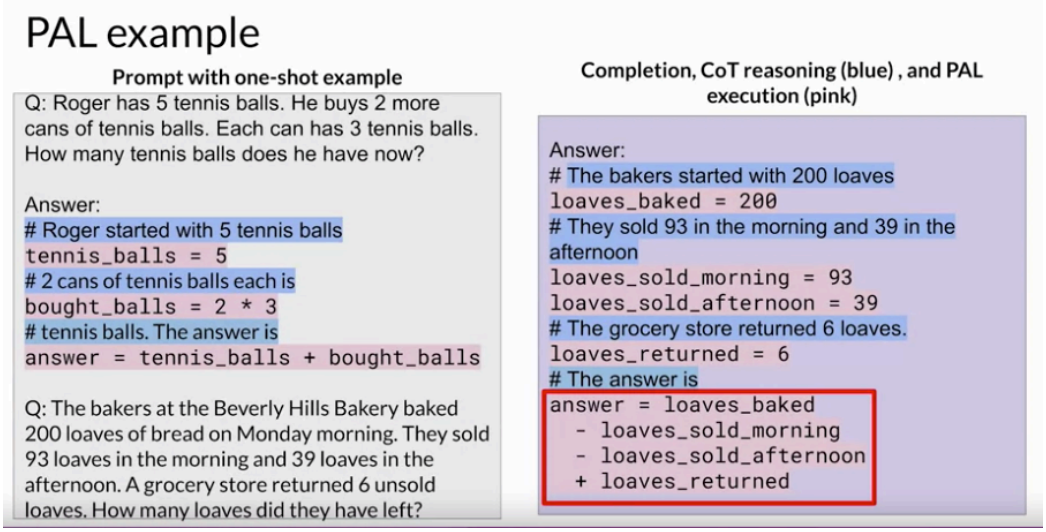
Challenge of LLMs: struggle with mathematics



How to overcome the challenge:
Allow model to interact with external applications that are good at math. E.g.: python interpreter.

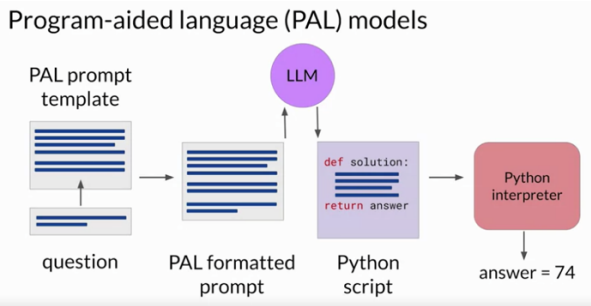
PAL (PAL = LLM + Code interpreter) is a framework that helps to augment LLMs by offloading computational tasks to other programs:

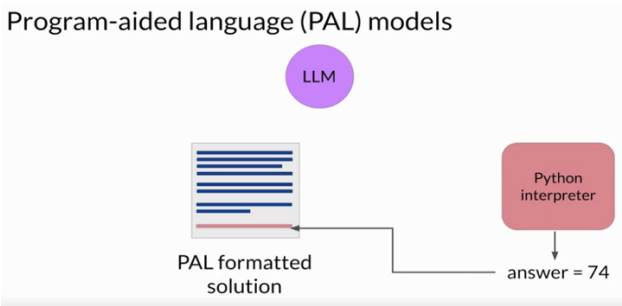
- Use an external code interpreter
- Chain of thought prompting is used to generate executable python scripts
- Use one-shot or few-shot examples to train model

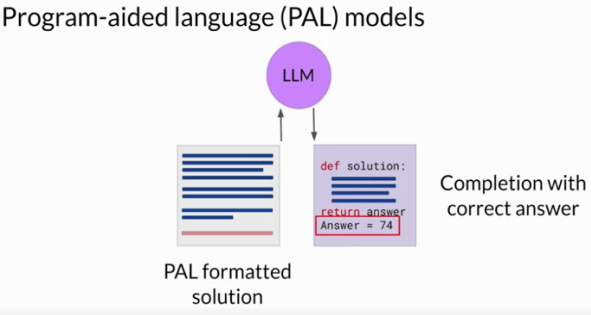


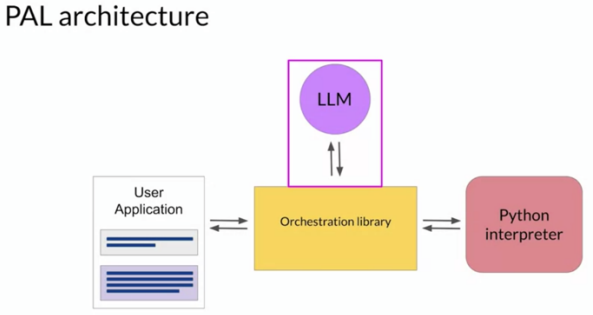
PAL framework:

- Create a prompt template with one or a few examples containing a question followed by reasoning steps in lines of Python code that solve the problem.
- Append the new question to the prompt template, which creates the PAL formatted prompt
- Pass this combined prompt to LLM, which generates completion in the form of a python script
- Pass the script to a Python interpreter which will run the code and generate the answer
- Append the text containing the answer
- When passing the updated prompt to the LLM, it generates a completion that contains the correct answer
- Orchestrator automatically pass the information between LLM (application reasoning engine) and the interpreter (place to make action):
    - manage the flow of information and the initiation of calls to external data sources or applications
    - decide what actions to take based on the information contained in the output of the LLM









ReAct: Combining reasoning and action

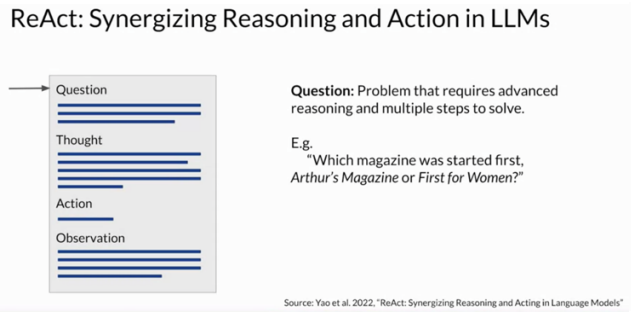
ReAct

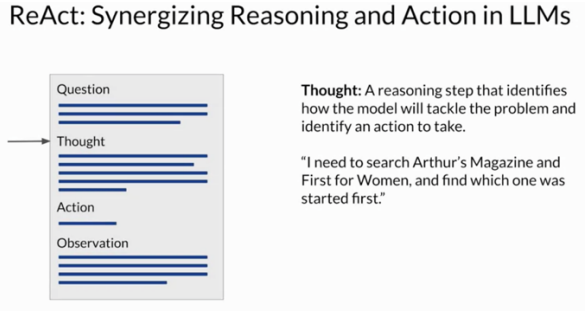
- Allows LLMs to plan and execute their courses of action with multiple interactions with other applications
- Used for more complex applications
- Uses structures examples to show the model how to reason through a problem and decide on actions to take
- The prompt consists of a question, along with the thought, action, and observation
    - Thought: Reasoning step that teacher the model how to tackle the problem
    - Action: An external task that the model can take from an allowed set of actions. Action can be three types:

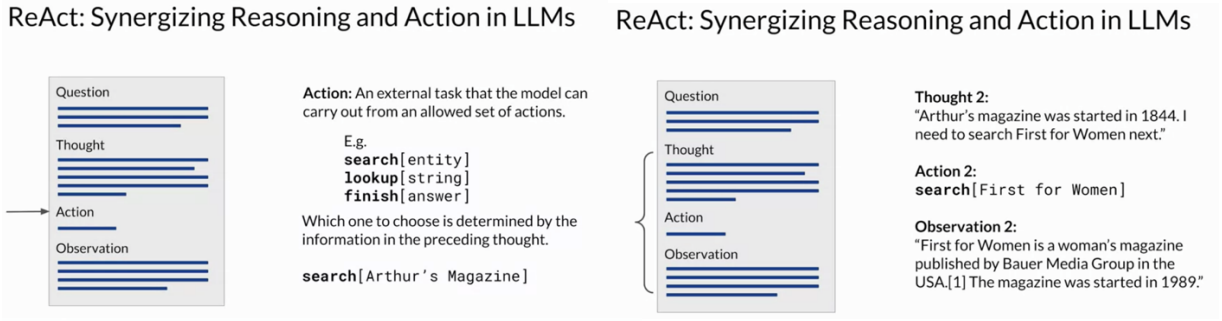
```
(1) Search[entity], which searches the exact entity on Wikipedia and
returns the first paragraph if it exists. If not, it will return
some similar entities to search.
(2) Lookup[keyword], which returns the next sentence containing
keyword in the current passage.
(3) Finish[answer], which returns the answer and finishes the task.
```

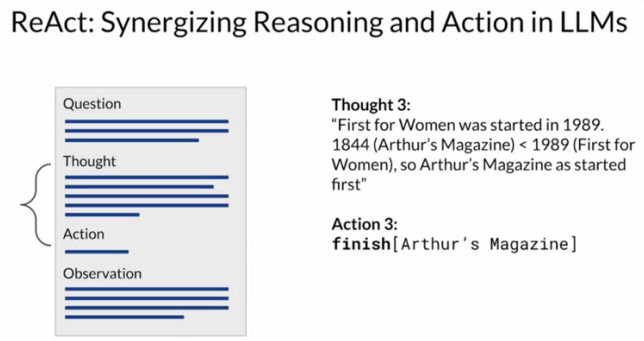
    - Observation: Result of carrying out the action
    - This cycle is repeated multiple times

Example of ReAct

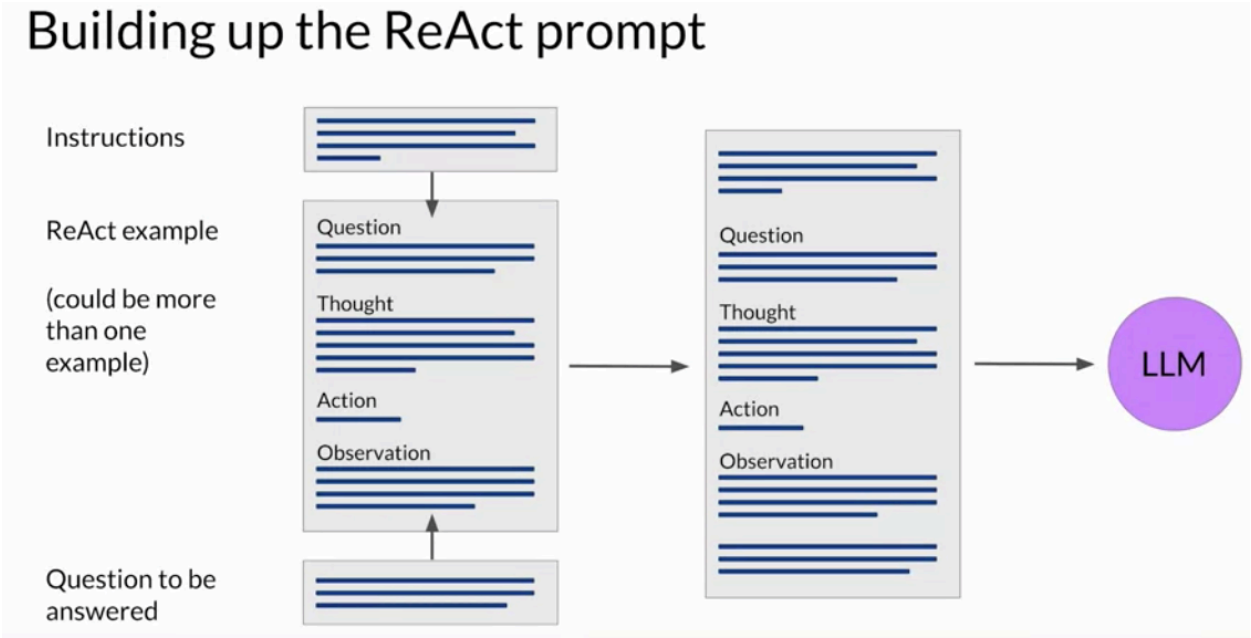








ReAct prompt: combine instructions, ReAct example(s) and question to be answer as one prompt to pass to LLM for inferencing.

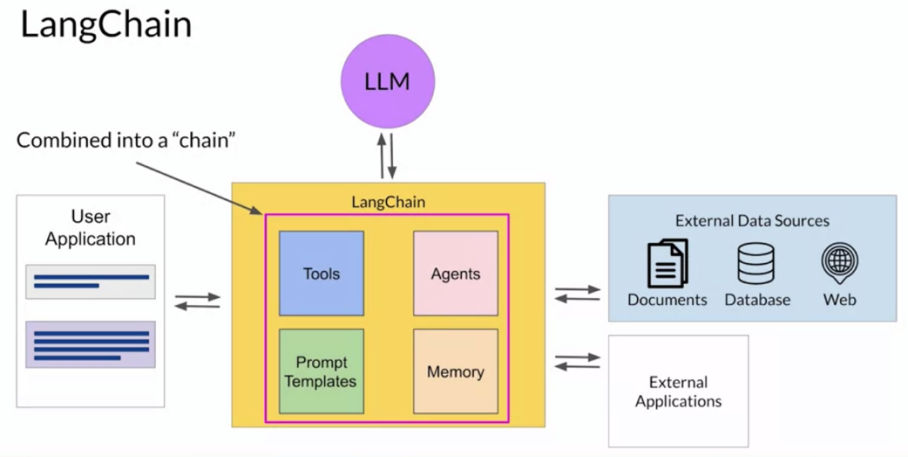


Langchain:

1) Provides you with modular pieces that contain the components necessary to work with LLMs. These components include
   - prompt templates for many different use cases that you can use to format both input examples and model completions.
   - memory that you can use to store interactions with an LLM.

- pre-built tools that enable you to carry out a wide variety of tasks, including calls to external datasets and various APIs.
- Agent: use to interpret the input from the user and determine which tool or tools to use to complete the task. LangChain currently includes agents for both PAL and ReAct, among others. Agents can be incorporated into chains to take an action or plan and execute a series of actions.

2) Connecting a selection of these individual components together results in a chain.
3) The creators of LangChain have developed a set of predefined chains that have been optimized for different use cases, and you can use these off the shelf to quickly get your app up and running. Sometimes your application workflow could take multiple paths depending on the information the user provides. In this case, you can't use a pre-determined chain, but instead we'll need the flexibility to decide which actions to take as the user moves through the workflow.
4) Larger models are generally your best choice for techniques that use advanced prompting, like PAL or ReAct. Smaller models may struggle to understand the tasks in highly structured prompts and may require you to perform additional fine tuning to improve their ability to reason and plan. This could slow down your development process. Instead, if you start with a large, capable model and collect lots of user data in deployment, you may be able to use it to train and fine tune a smaller model that you can switch to at a later time.



LLM application architectures

- Infrastructure layer: provides the compute, storage, and network to serve up your LLMs, as well as to host your application components. You can use your on-premises infrastructure or on-demand and pay-as-you-go Cloud services for this.
- LLM models: foundation models or models you have adapted to your specific task. Do you need real time or near real time interaction with the model?
- Information sources: RAG
- Generated output and feedback: capture and store the outputs. For example, you could build the capacity to store user completions during a session to augment the fixed contexts window size of your LLM. You can also gather feedback from users that may be useful for additional fine-tuning, alignment, or evaluation as your application matures.

- LLM tools and framework: additional tools and frameworks for large language models that help you easily implement some of the techniques
- Application interface: some type of user interface that the application will be consumed through, such as a website or a rest API. Include the security components required for interacting with your application.