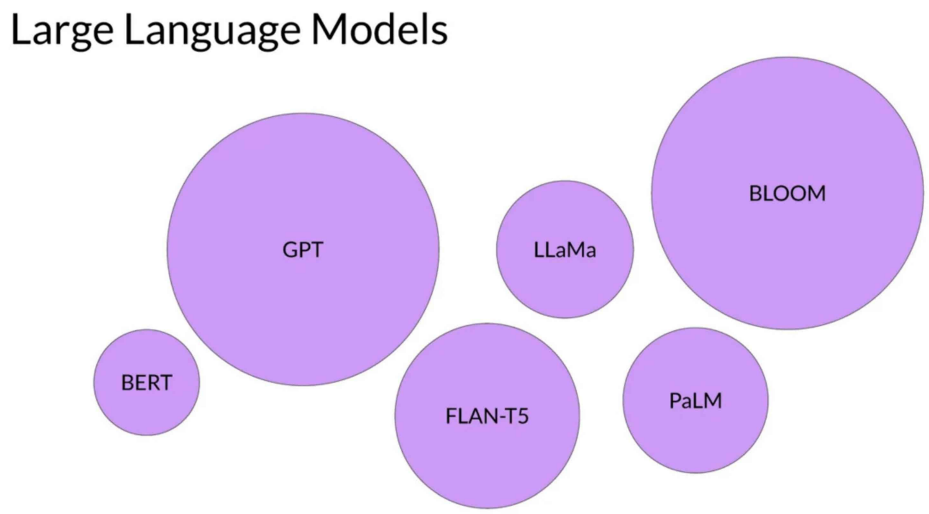
W1: GenAI, LLMs, Transformers, Prompt Engg

Wednesday, 7 August 2024 11:04 AM

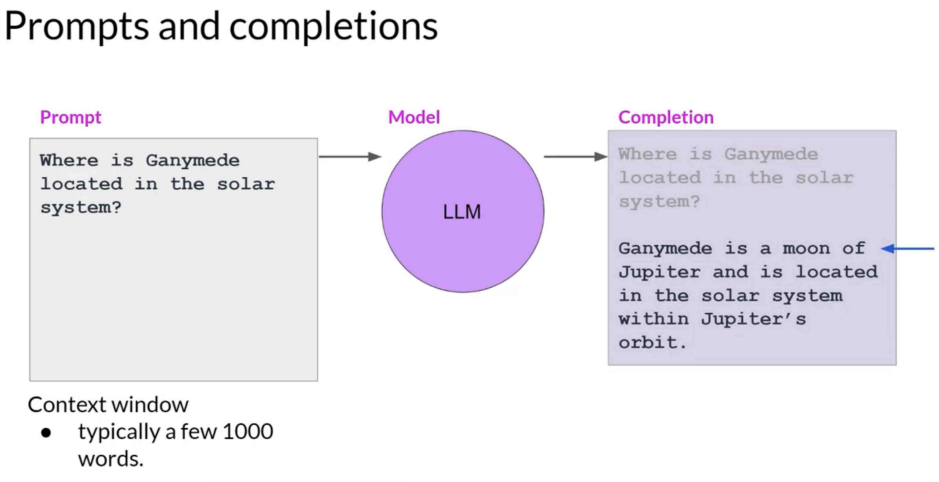
7th August.

Generative AI & LLMs

Collection of foundational models also called as Base models:



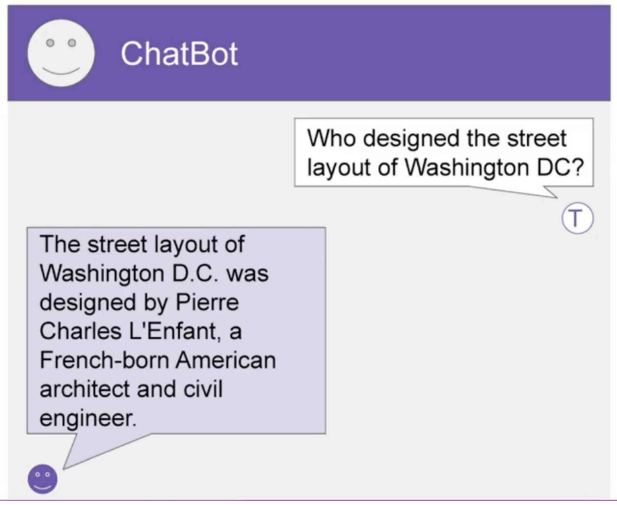
- circle shows their relative size in terms of their parameters
- the more parameters a model has, the more memory, and as it turns out, the more sophisticated the tasks it can perform.



- The prompt is passed to the model, the model then predicts the next words, and because your prompt contained a question, this model generates an answer.
- The output of the model is called a *completion*, and the act of using the model to generate text is known as *inference*.
- The completion is comprised of the text contained in the original prompt, followed by the generated text.

LLM use cases and tasks

LLM chatbot



- Next word prediction is the base concept behind a number of different capabilities, starting with a basic chatbot.
- You can use this conceptually simple technique for a variety of other tasks within text generation.
- For example, you can ask a model to
    - write an essay based on a prompt
    - summarize conversations where you provide the dialogue as part of your prompt
    - translation tasks
        - translate natural language to machine code
    - information retrieval (e.g., NER)
    - Augmenting LLMs by connecting them to external data sources or using them to invoke external APIs. You can use this ability to provide the model with information it doesn't know from its pre-training and to enable your model to power interactions with the real-world.

Text generation before transformers

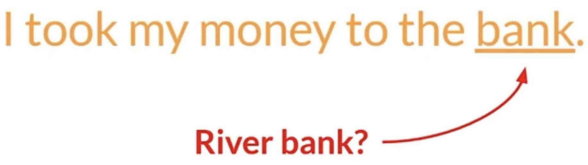
- Previously generation of language models used RNNs for carrying out a simple next-word prediction generative task. With just one previous words seen by the model, the prediction can't be very good.
- Also RNNs were limited by the amount of compute and memory needed to perform well at generative tasks.

- the model failed here. Even though you scale the model, it still hasn't seen enough of the input to make a good prediction.

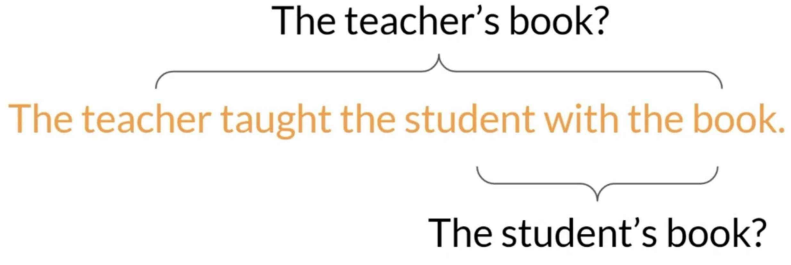
★- To successfully predict the next word, models need to see more than just the previous few words. Models needs to have an understanding of the whole sentence or even the whole document.

- The problem here is that language is complex. In many languages, one word can have multiple meanings. These are *homonyms*.

Here context matters



syntactic ambiguity



? **How can an algorithm make sense of human language if sometimes we can't?**

- Well in 2017, after the publication of the paper, Attention is All You Need, from Google and the University of Toronto, everything changed. The transformer architecture had arrived.
- This novel approach unlocked the progress in generative AI that we see today. It can be scaled efficiently to use multi-core GPUs, it can parallel process input data, making use of much larger training datasets, and crucially, it's able to learn to pay attention to the meaning of the words it's processing.
- And attention is all you need. It's in the title.

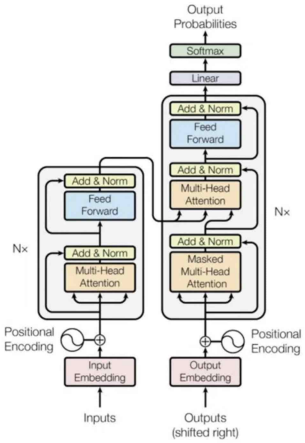
Transformers

Attention Is All You Need

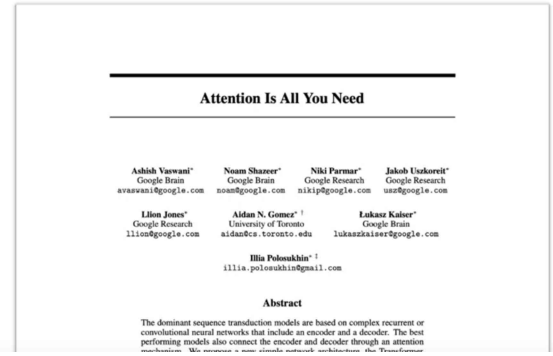
Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to



Transformers



Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

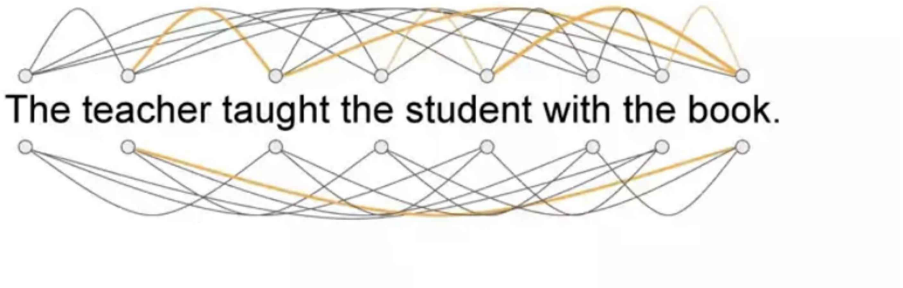
Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention

- Scale efficiently
- Parallel process
- Attention to input meaning

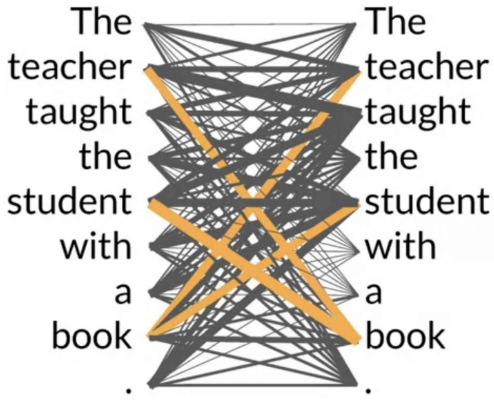
Transformers architecture

- The power of the transformer architecture lies in its ability to learn the relevance and context of all of the words in a sentence.



- Here we see how each word is connected with other word to capture the context, to apply attention weights to those relationships so that the model learns the relevance of each word to every other word no matter where they are in the input.

Self-attention



- This diagram is called an *attention map* and can be useful to illustrate the attention weights between each word and every other word.
- These attention weights are learned during LLM training

- The transformer architecture is split into two distinct parts, the *encoder* and the *decoder*. These components work in conjunction with each other and they share a number of similarities.

- Also, note here, the diagram you see is derived from the original attention is all you need paper. Notice how the inputs to the model are at the bottom and the outputs are at the top,

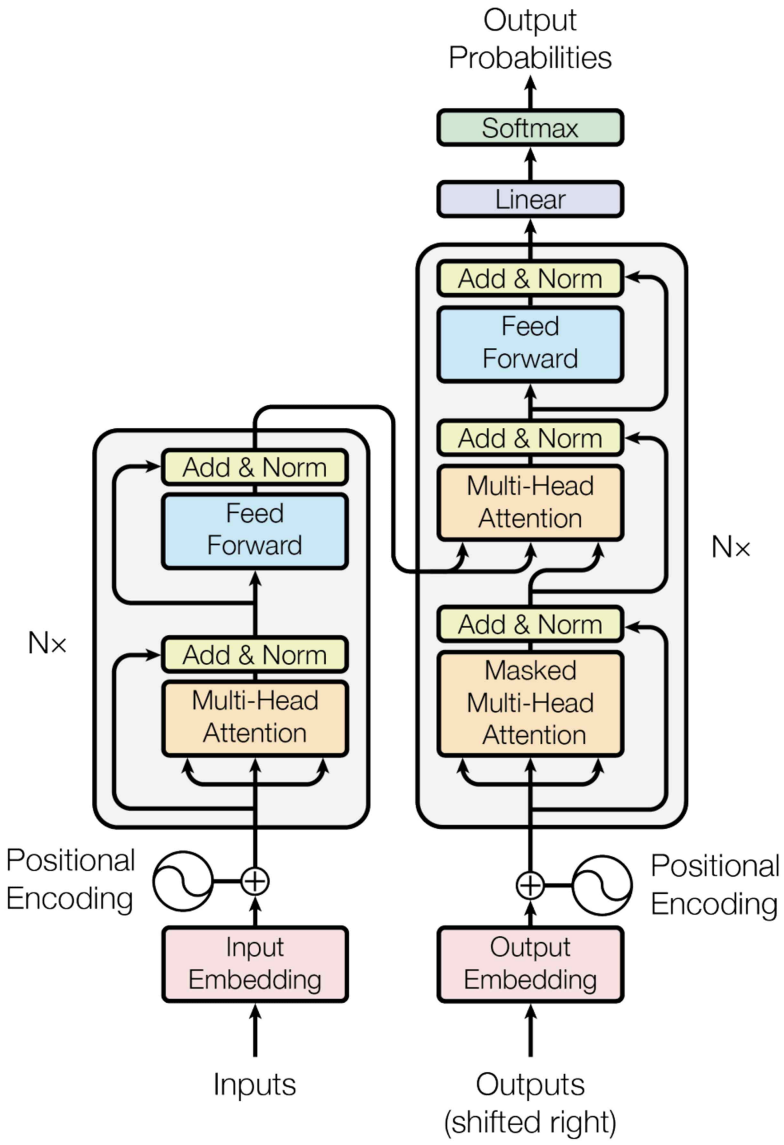
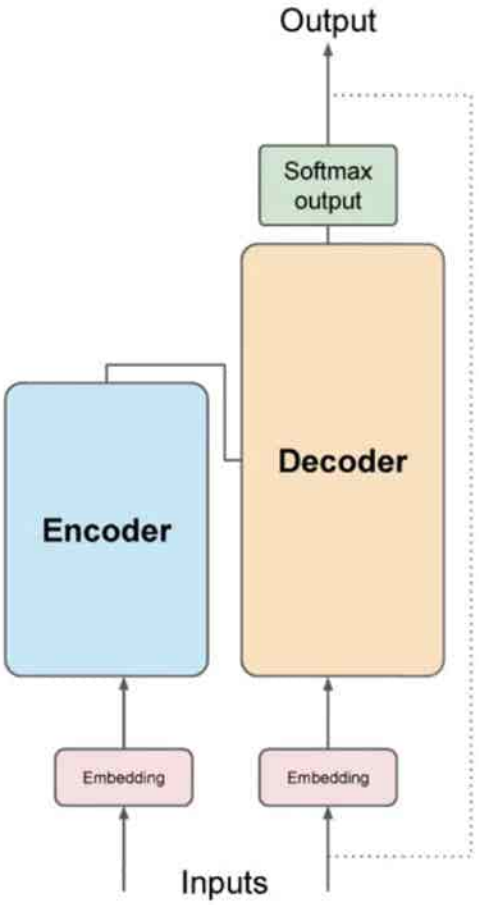


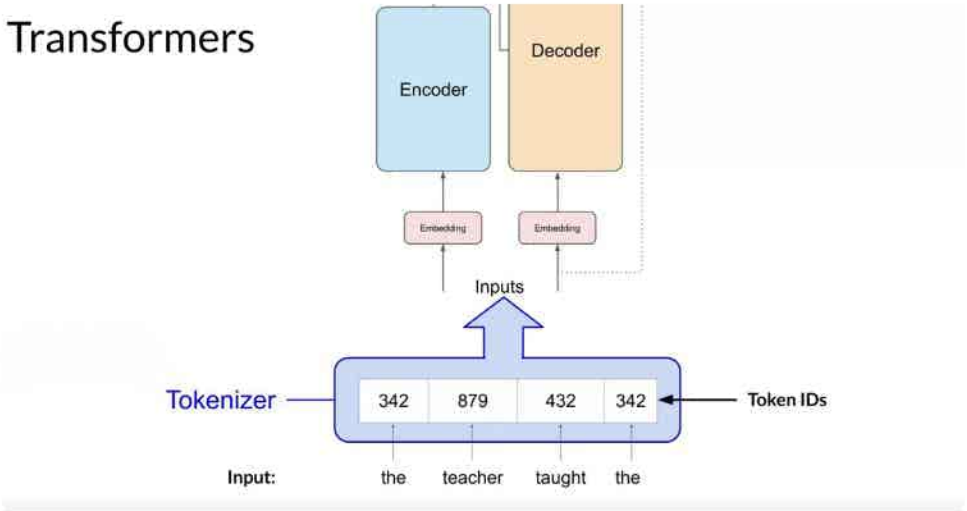
Figure 1: The Transformer - model architecture.

Simplified transformers architecture:

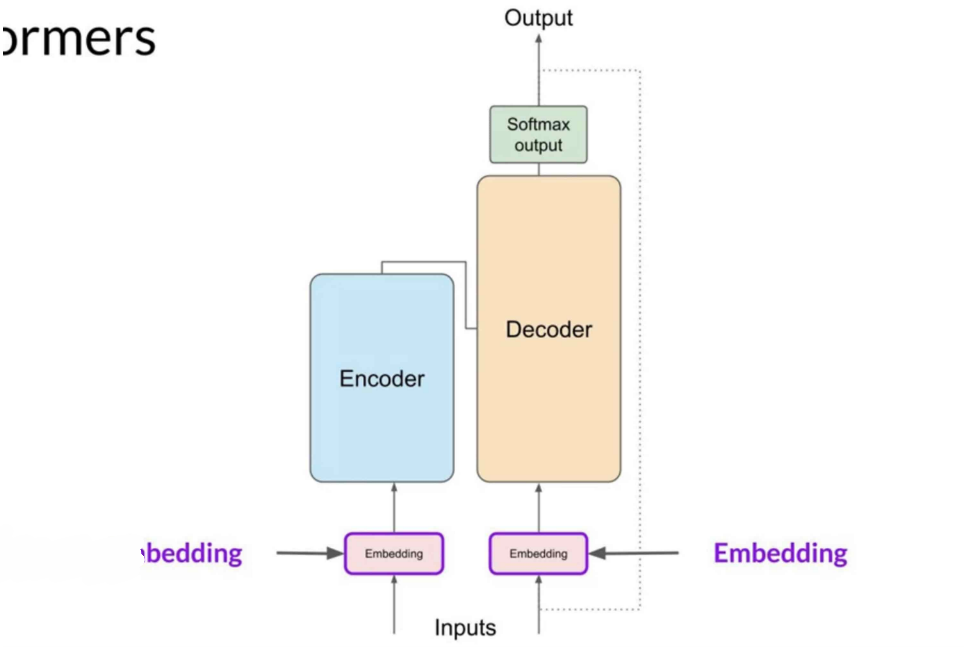


1. **Tokenized Input-**

- before passing texts into the model to process, you must first tokenize the words.
    - This converts the words into numbers, with each number representing a position in a dictionary of all the possible words that the model can work with.



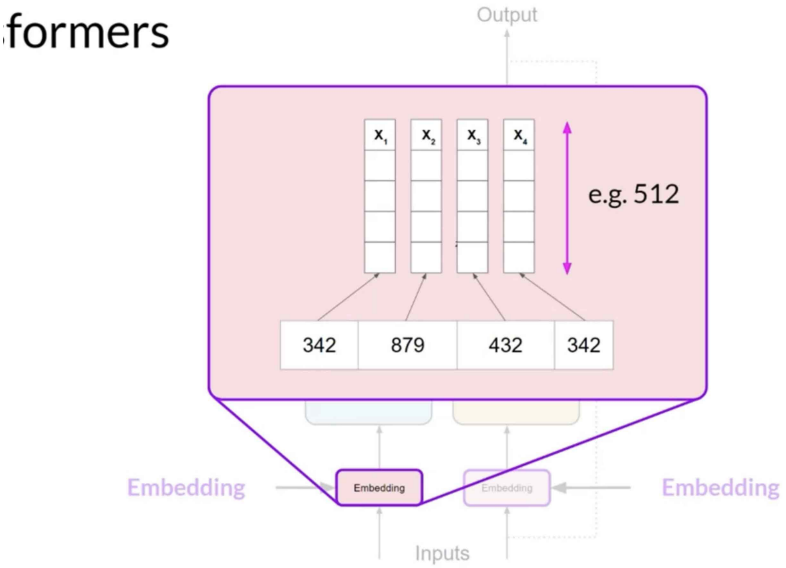
2. **Embedding layer -**

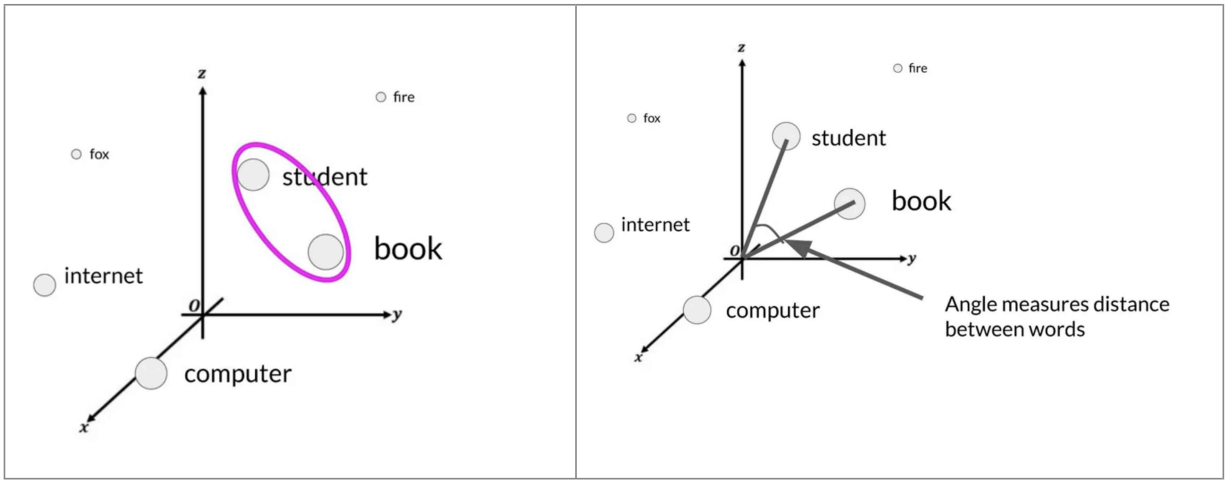


- Now that your input is represented as numbers, you can pass it to the embedding layer. This layer is a trainable vector embedding space, a high-dimensional space where each token is represented as a vector and occupies a unique location within that space.

★- Each token ID in the vocabulary is matched to a multi-dimensional vector, and the intuition is that these vectors learn to encode the meaning and context of individual tokens in the input sequence.

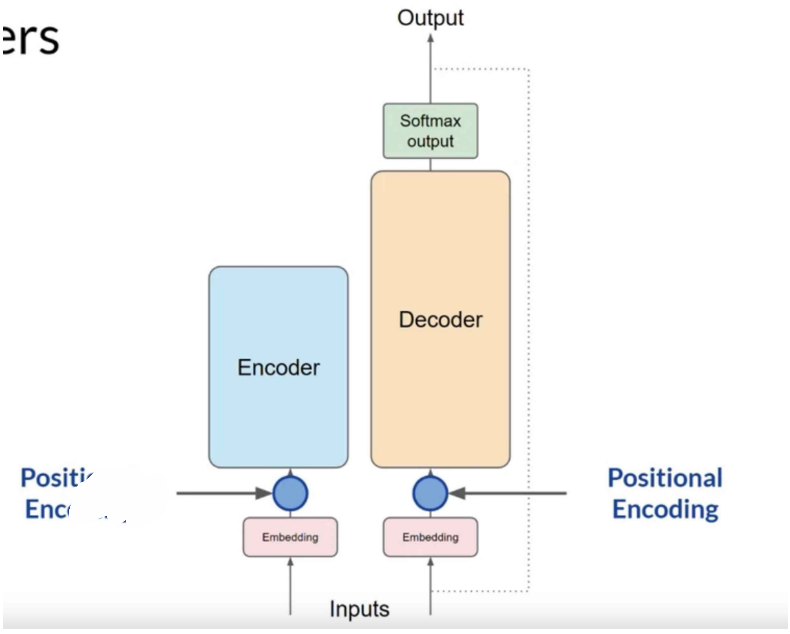
- below fig shows that each word has been matched to a token ID, and each token is mapped into a 512 dimensional vector here.



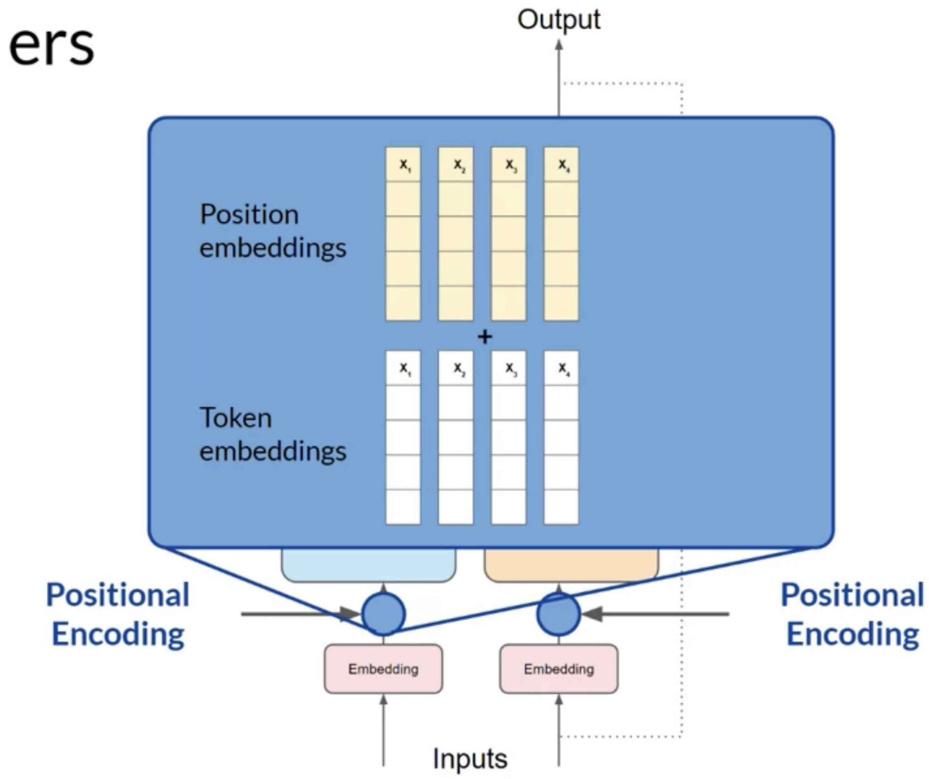


★- You can see now how you can relate words that are located close to each other in the embedding space, and how you can calculate the distance between the words as an angle, which gives the model the ability to mathematically understand language.

3. Positional embedding



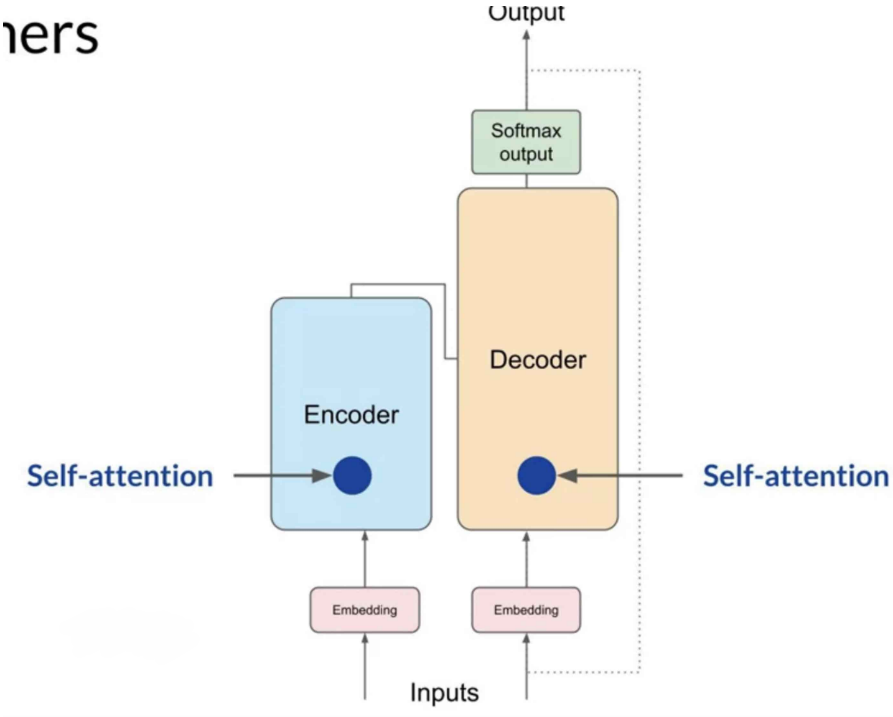
- As you add the token vectors into the base of the encoder or the decoder, you also add positional encoding.



★- The model processes each of the input tokens in parallel. So by adding the positional encoding, you preserve the information about the word order and don't lose the relevance of the position of the word in the sentence.

4. **Self-attention layer**

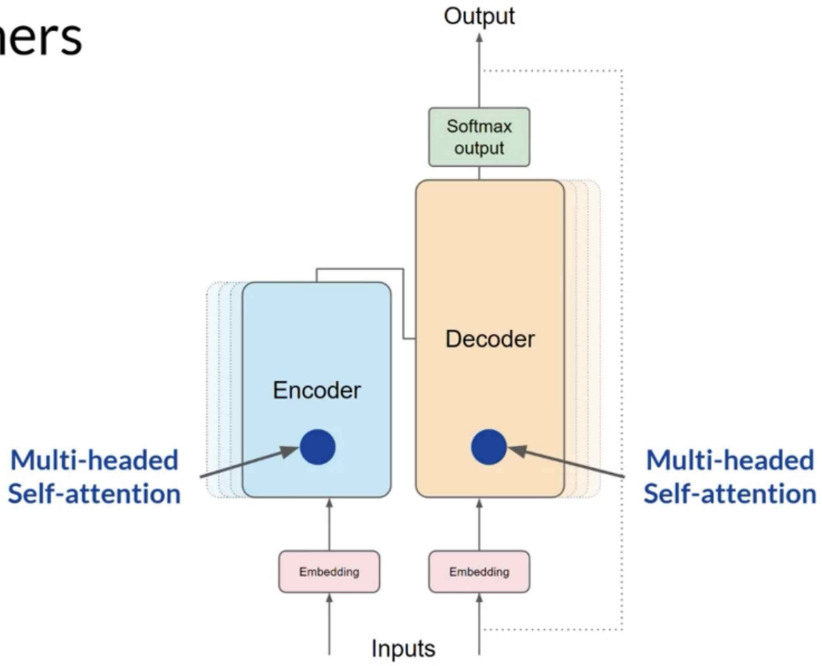
- Once you've summed (concatenated) the input tokens and the positional encodings, you pass the resulting vectors to the self-attention layer.



- Self attention layer allows the model to attend to different parts of the input sequence to better capture the contextual dependencies between the words.
- The self-attention weights that are learned during training and stored in these layers that reflect the importance of each word in that input sequence to all other words in the sequence.
    - self attention weights reflects the importance of each word wrt all other words in the sequence.

ners

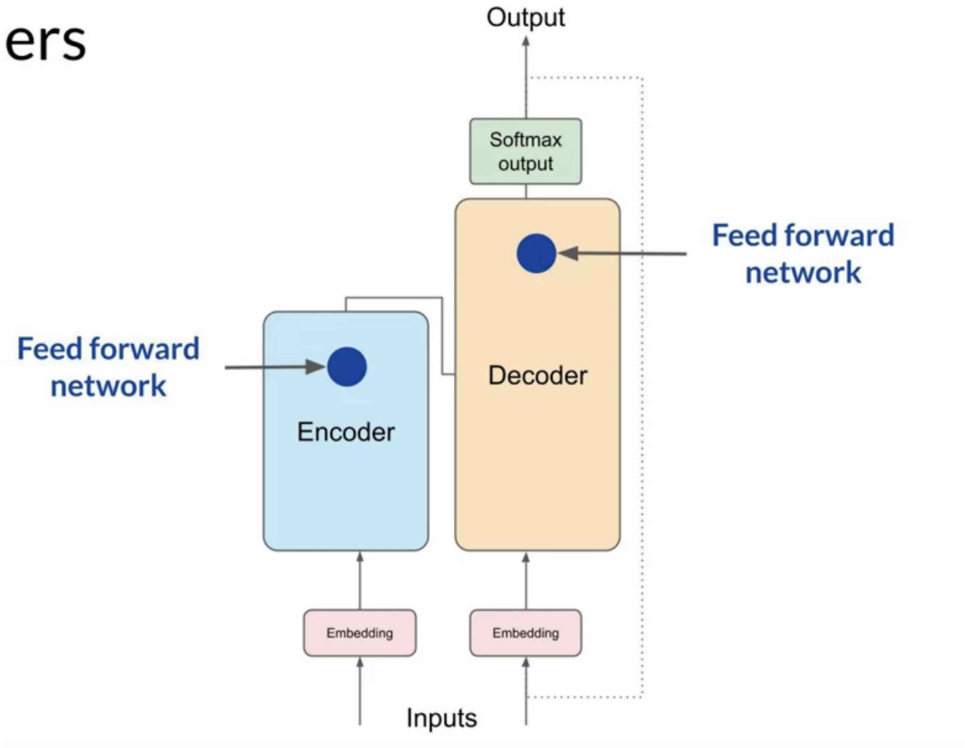
-



★- the transformer architecture actually has multi-headed self-attention. This means that multiple sets of self-attention weights or heads are learned in parallel independently of each other. (the number of attention heads in the range of 12-100 are common)
    - The intuition here is that each self-attention head will learn a different aspect of language.

5. **Feed-forward layer**

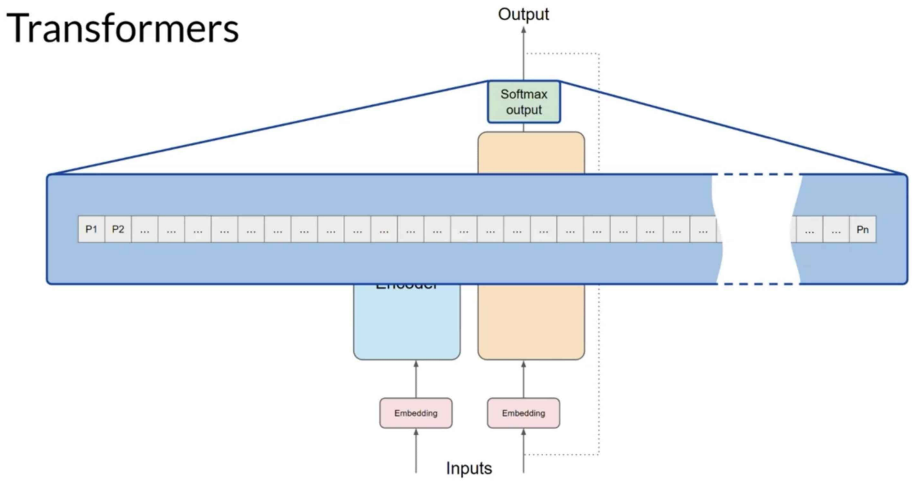
- Now that all of the attention weights have been applied to your input data, the output is processed through a fully-connected feed-forward network.



- The output of this layer is a vector of logits proportional to the probability score for each and every token in the tokenizer dictionary.

6. **SoftMax layer**

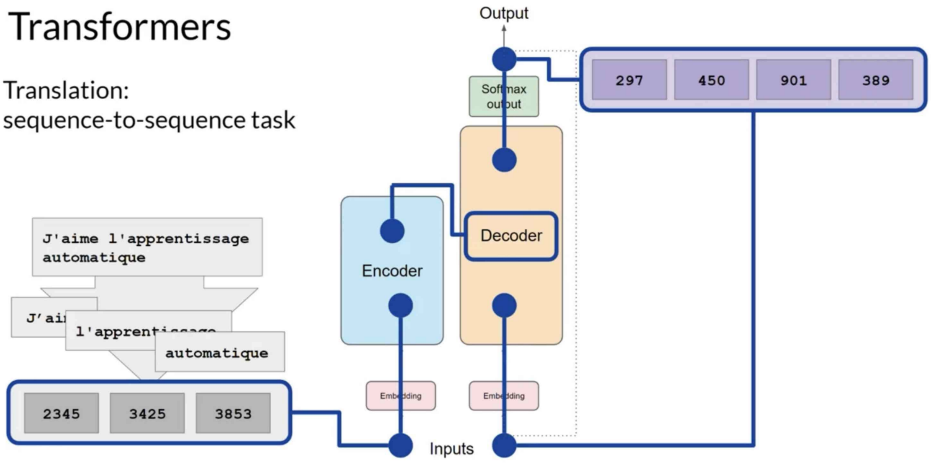
- You can then pass these logits to a final SoftMax layer, where they are normalized into a probability score for each word.



- So this output includes a probability for every single word in the vocabulary, so there's likely to be thousands of scores here.(depending on vocabulary size)
- One single token will have a score higher than the rest. This is the most likely predicted token.

Generating text with transformers

- Here is an example of Translation task or a sequence-to-sequence task (which incidentally was the original objective of the transformer architecture designers)
- The task is to use a transformer model to translate the French phrase into English.



Steps:

- First, you'll tokenize the input words using this same tokenizer that was used to train the network.

- These tokens are then added into the input on the encoder side of the network, then they are passed through the embedding layer, and then fed into the multi-headed attention layers.

- The outputs of the multi-headed attention layers are fed through a feed-forward network to the output of the encoder. At this point, the data that leaves the encoder is a deep representation of the structure and meaning of the input sequence.

- This representation is inserted into the middle of the decoder to influence the decoder's self-attention mechanisms.

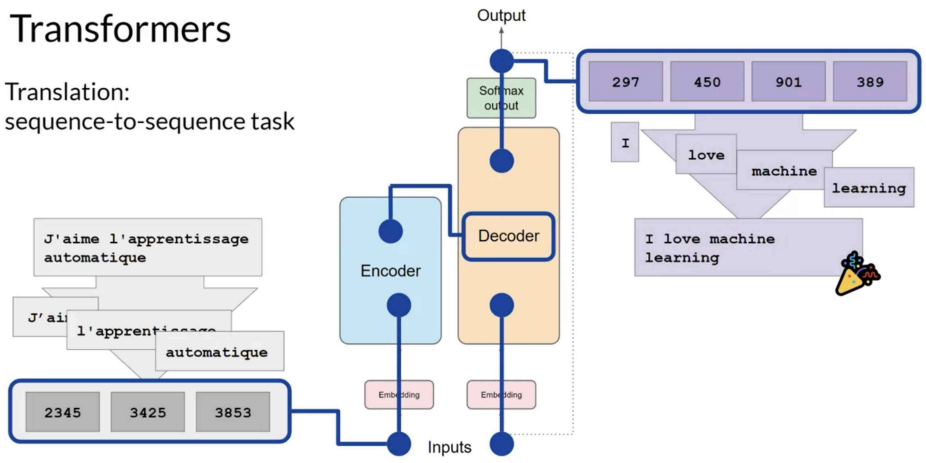
- Next, a start of sequence token is added to the input of the decoder. This triggers the decoder to predict the next token, which it does based on the contextual understanding that it's being provided from the encoder.

- The output of the decoder's self-attention layers gets passed through the decoder feed-forward network and through a final softmax output layer. At this point, we have our first token.

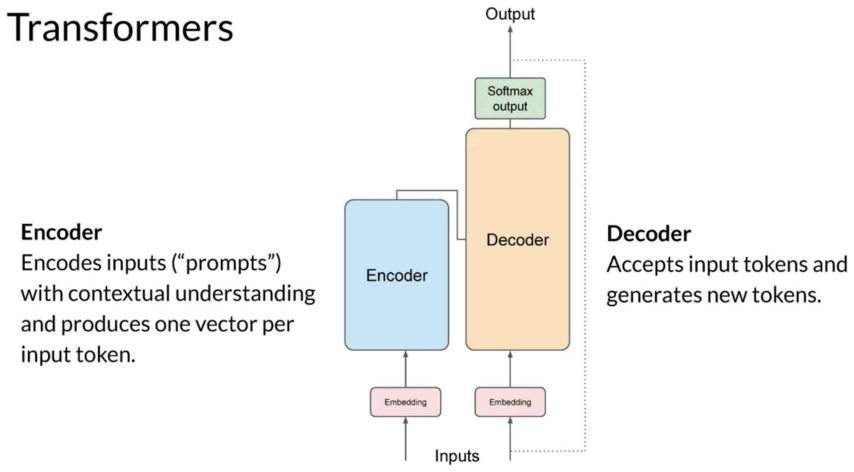
- Then we continue this loop, passing the output token back to the input to trigger the generation of the next token, until the model predicts an end-of-sequence token.

- At this point, the final sequence of tokens can be detokenized into words, and you have your output. In this case, I love machine learning.

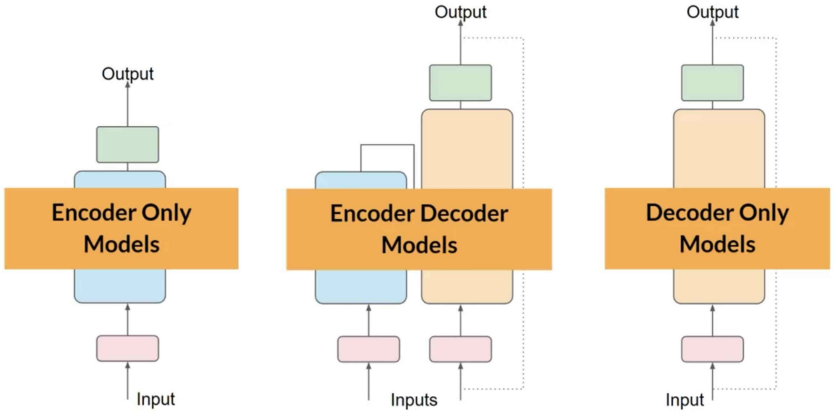
★- There are multiple ways to use the output from the softmax layer to predict the next token and this can influence how creative your generated text is.



Summary: The complete 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, 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



Transformers



Encoder-only Models	**Encoder Decoder Models**	**Decode only Models**
Encoder-only models also work as sequence-to-sequence models, but without further modification, the input sequence and the output sequence or the same length. By adding additional layers to the architecture, you can train encoder-only models to perform classification tasks such as sentiment analysis, examples: **BERT**	Encoder-decoder 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. examples: BART, **T5**	decoder-only models are some of the most commonly used today. As they have scaled, their capabilities have grown. Popular decoder-only models include the GPT family of models, BLOOM, Jurassic, LLaMA, and many more…

Summary from the paper:

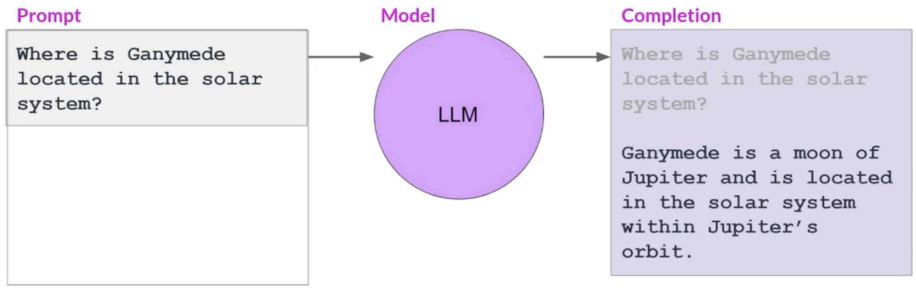
The Transformer architecture consists of an encoder and a decoder, each of which is composed of several layers. Each layer consists of two sub-layers: a multi-head self-attention mechanism and a feed-forward neural network. The multi-head self-attention mechanism allows the model to attend to different parts of the input sequence, while the feed-forward network applies a point-wise fully connected layer to each position separately and identically.

The Transformer model also uses residual connections and layer normalization to facilitate training and prevent overfitting. In addition, the authors introduce a positional encoding scheme that encodes the position of each token in the input sequence, enabling the model to capture the order of the sequence without the need for recurrent or convolutional operations.

Prompting and prompt engineering

Terminology:

- The text that you feed into the model is called the *prompt*,
- the act of generating text is known as *inference*,
- and the output text is known as the *completion*.
- The full amount of text or the memory that is available to use for the prompt is called the *context window.*



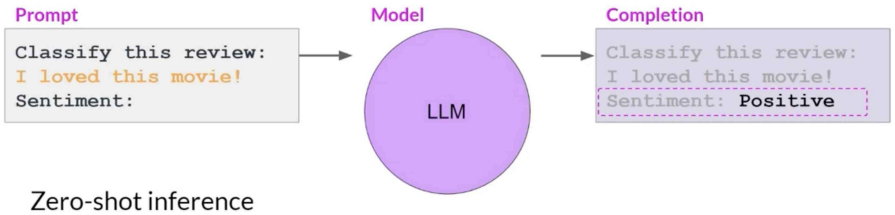
Context window: typically a few thousand words

- Sometimes the model doesn't produce the outcome that you want on the first try and you may have to revise the language in your prompt or the way that it's written several times to get the model to behave in the way that you want. *This work to develop and improve the prompt is known as prompt engineering.*

- One powerful strategy to get the model to produce better outcomes is to include examples of the task that you want the model to carry out inside the prompt. Providing examples inside the context window is called *in-context learning.*

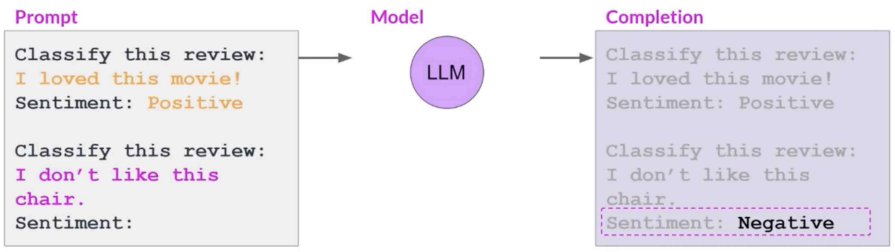
- Types of in-context learning (ICL):

In-context learning (ICL) - zero shot inference

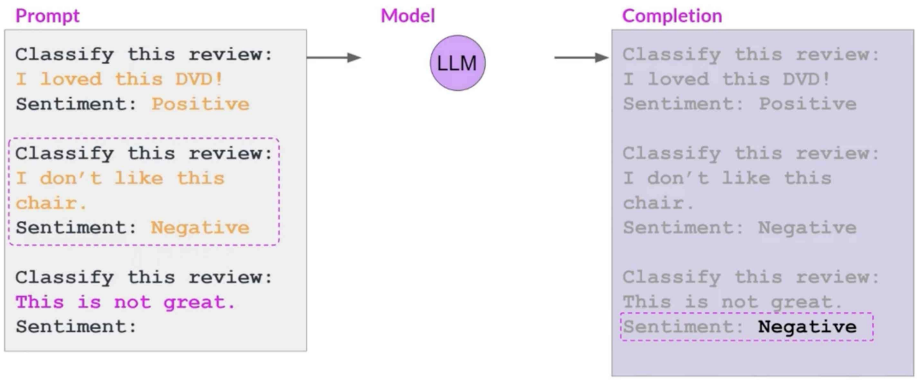


Zero-shot inference

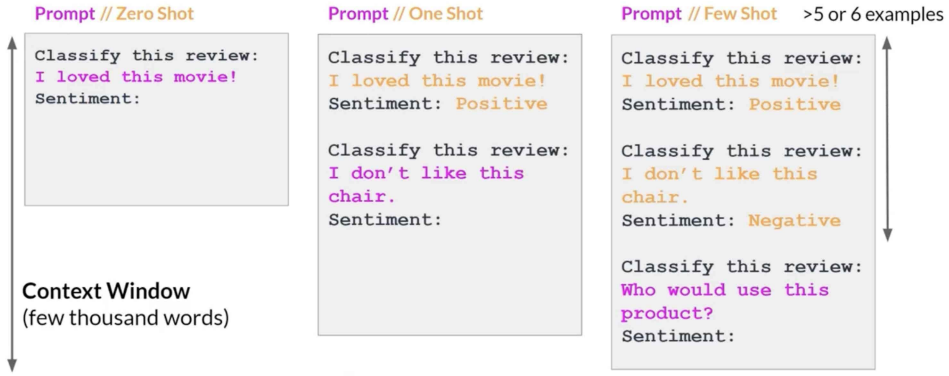
In-context learning (ICL) - one shot inference



In-context learning (ICL) - few shot inference

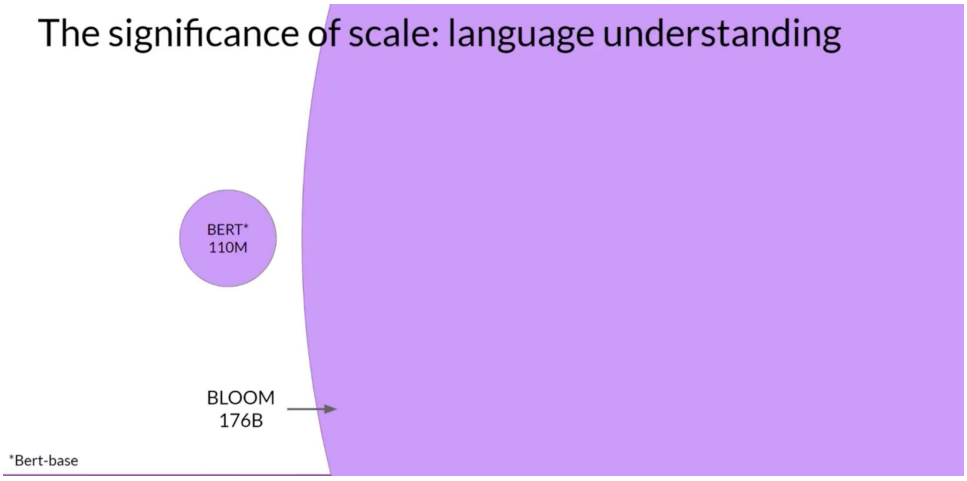


Summary of in-context learning (ICL)



- if you find that your model isn't performing well when, say, including five or six examples, you should try fine-tuning your model instead.
- Fine-tuning performs additional training on the model using new data to make it more capable of the task you want it to perform.

- The ability of models to perform multiple tasks and how well they perform on those tasks depends strongly on the scale of the model.



Generative configuration

Which configuration parameter for inference can be adjusted to either increase or decrease randomness within the model output layer?

- ○ Max new tokens
- ○ Top-k sampling
- ◉ Temperature
- ○ Number of beams & beam search

✓ **Correct**
Temperature is used to affect the randomness of the output of the softmax layer. A lower temperature results in reduced variability while a higher temperature results in increased randomness of the output.

- lower temp - reduced variability
- higher temp - increased randomness

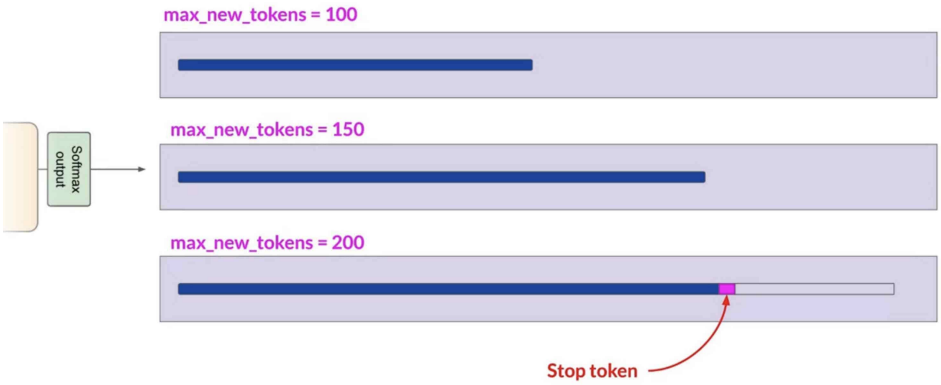
- We can use *configuration parameters* to influence the way that the model makes the final decision about next-word generation.

Generative configuration - inference parameters

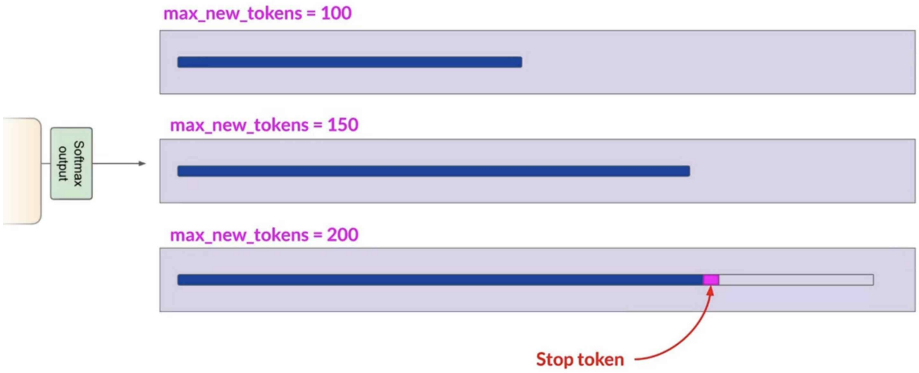
- Configuration parameters are invoked at inference time and give you control over things like the maximum number of tokens in the completion, and how creative the output is.

1. **Max new tokens** : limits the number of tokens that the model will generate

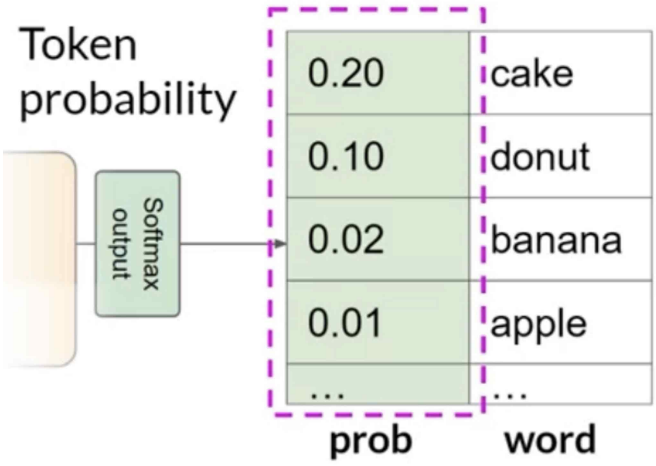
Generative config - max new tokens



Generative config - max new tokens



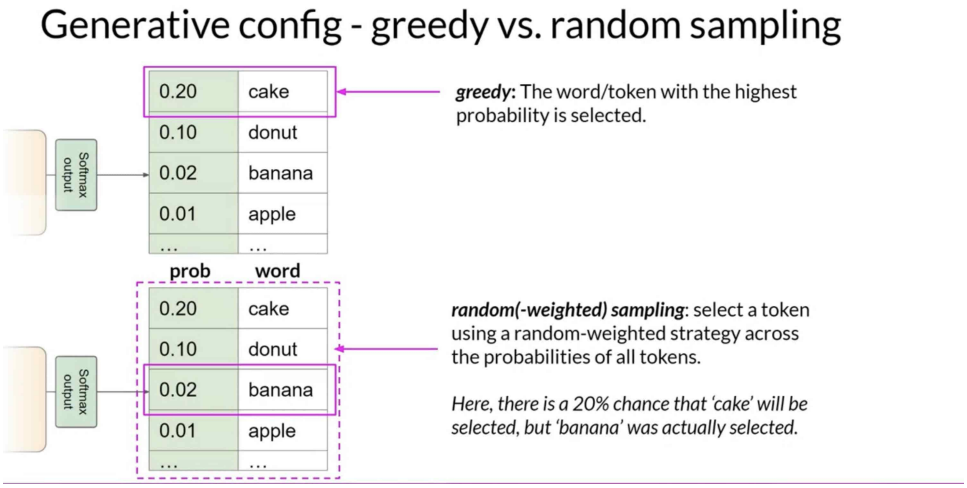
- max new tokens being set to 100, 150, or 200. The length of the completion in the example for 200 is shorter because another stop condition was reached, such as the model predicting and end of sequence token.
- The output from the transformer's softmax layer is a probability distribution across the entire dictionary of words that the model uses.



- Here you can see a selection of words and their probability score next to them

- Most large language models by default will operate with so-called ***greedy decoding***. In this the model will always choose the word with the highest probability. This method can work very well for short generation but is susceptible to repeated words or repeated sequences of words.

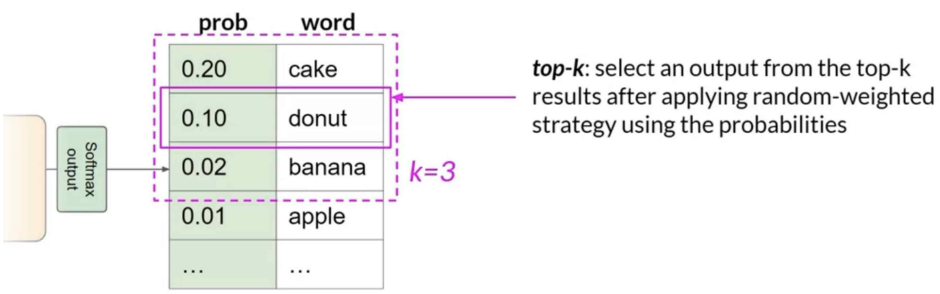
-



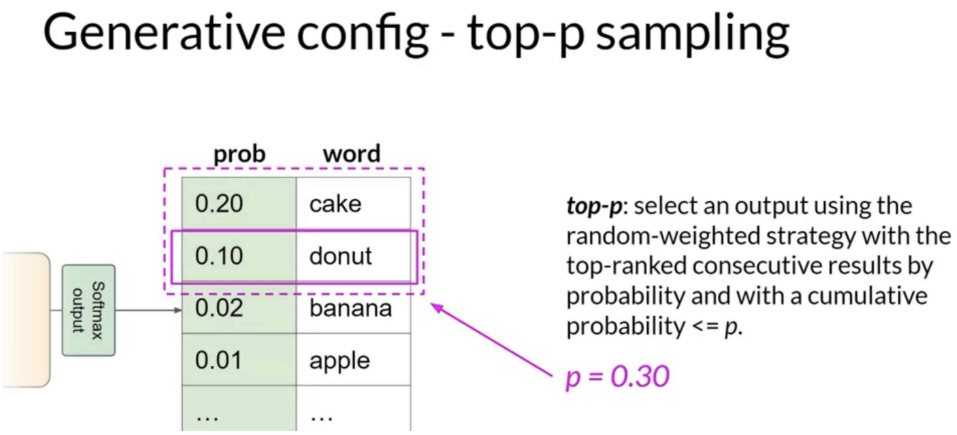
- If you want to generate text that's more natural, more creative and avoids repeating words, you need to use some other controls. You can do that by using ***Random sampling***.
- Here instead of selecting the most probable word every time, with random sampling the model chooses an output word at random using the probability distribution to weight the selection.
- By using this sampling technique, we reduce the likelihood that words will be repeated.

- Top K and Top P are sampling techniques that we can use to help limit the random sampling and increase the chance that the output will be sensible.

2. **Sample top K:** choose from only the k tokens with the highest probability.



3. **Sample top P:** limit the random sampling to the predictions whose combined probabilities do not exceed p.

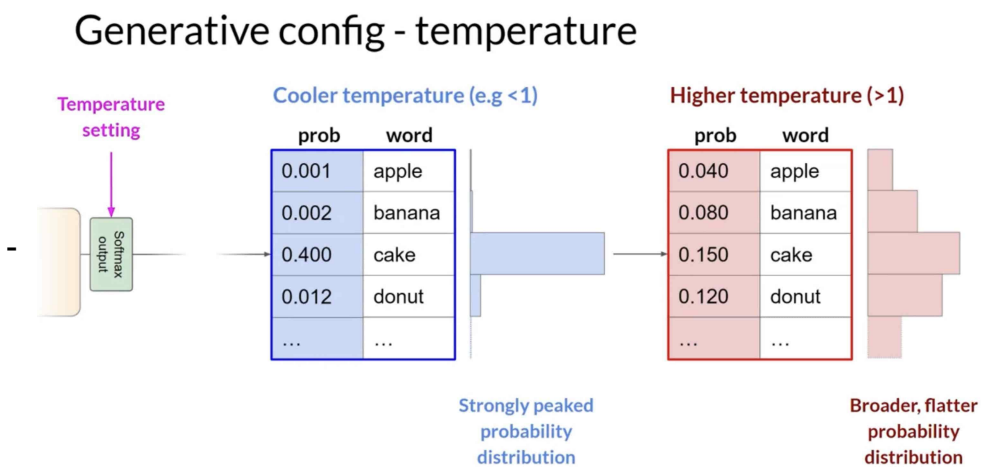


- If you set p to equal 0.3, the options are cake and donut since their probabilities of 0.2 and 0.1 add up to 0.3. The model then uses the random probability weighting method to choose from these tokens.

Summary: With top k, you specify the number of tokens to randomly choose from, and with top p, you specify the total probability that you want the model to choose from.

4. **Temperature:** Influences the shape of the probability distribution that the model calculates for the next token. the higher the temperature, the higher the randomness, and the lower the temperature, the lower the randomness.

- temperature value is a scaling factor that's applied within the final softmax layer that impacts the shape of the probability distribution of the next token.



- **If temperature <1** - the resulting probability distribution from the softmax layer is more strongly peaked with the probability being concentrated in a smaller number of words. Most of the probability here is concentrated on the word cake.

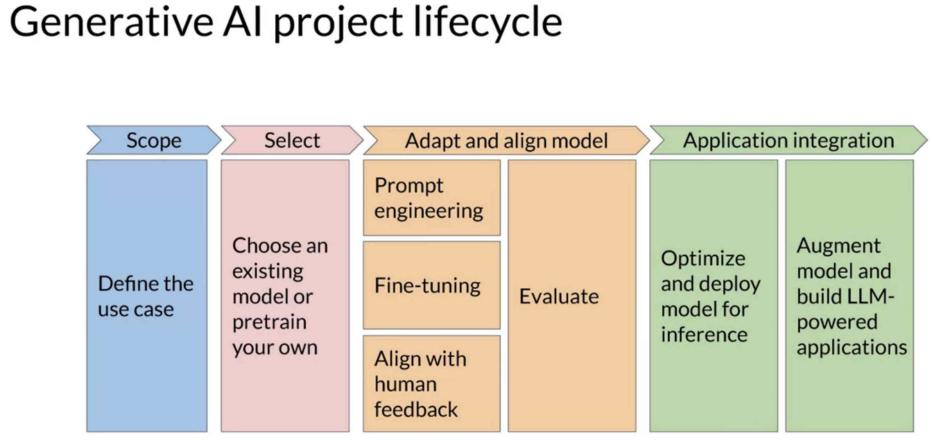
- The model will select from this distribution using random sampling and the resulting text will be less random and will more closely follow the most likely word sequences that the model learned during training.

- **If temperature > 1** - The model will calculate a broader flatter probability distribution for the next token. In contrast to the blue bars, the probability is more evenly spread across the tokens.

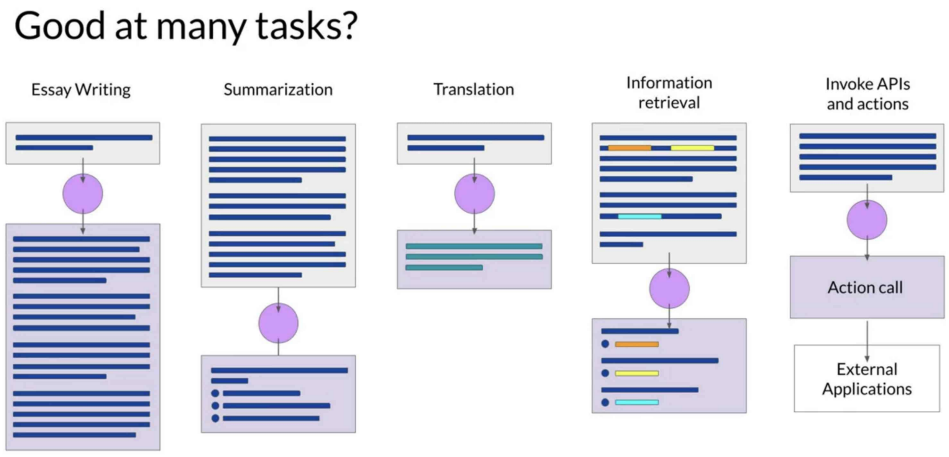
- This leads the model to generate text with a higher degree of randomness and more variability in the output compared to a cool temperature setting. This can help you generate text that sounds more creative.

- If you leave the temperature value == 1, this will leave the softmax function as default and the unaltered probability distribution will be used.

Generative AI project lifecycle

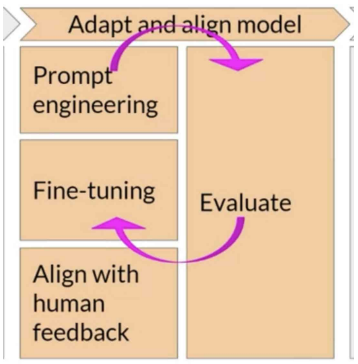


- **Scope**: LLMs are capable of carrying out many tasks, but their abilities depend strongly on the size and architecture of the model. You should think about what function the LLM will have in your specific application



- **Select**: deciding whether to train your own model from scratch or work with an existing base model. In general, you'll start with an existing model, although there are some cases where you may find it necessary to train a model from scratch.

- **Adapt and align model**: assess models performance and carry out additional training if needed for your application
    - *prompt engineering* can sometimes be enough to get your model to perform well, so you'll likely start by trying in-context learning, using examples suited to your task and use case.
    - There are still cases, however, where the model may not perform as well as you need, even with one or a few short inference, and in that case, you can try *fine-tuning your model.*
    - *Evaluate* - explore some metrics and benchmarks that can be used to determine how well your model is performing or how well aligned it is to your preferences.
    - Note that this adapt and aligned stage of app development can be highly iterative.



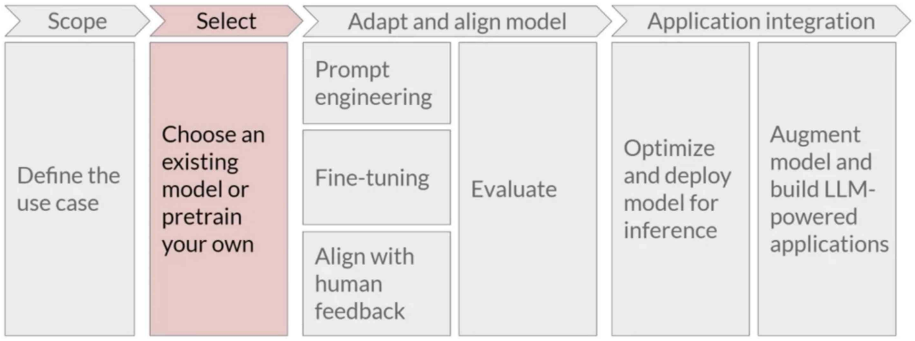
- **Application integration:**
    - Finally, when you've got a model that is meeting your performance needs and is well aligned, you can deploy it into your infrastructure and integrate it with your application.

- At this stage, an important step is to *optimize your model for deployment.* This can ensure that you're making the best use of your compute resources and providing the best possible experience for the users of your application.
- consider any additional infrastructure that your application will require to work well. (There are some fundamental limitations of LLMs that can be difficult to overcome through training alone like their tendency to invent information when they don't know an answer, or their limited ability to carry out complex reasoning and mathematics.)

LLM Pretraining and Scaling laws

Thursday, 8 August 2024 3:39 PM

Pre-training large language models



- Once you have scoped out your use case, your next step is to select a model to work with. Your first choice will be to either work with an existing model, or train your own from scratch.
    - Hugging Face and Pytorch, have curated hubs where you can browse pretrained models.
- The exact model that you'd choose will depend on the details of the task you need to carry out

<u>How large language models are trained :</u>

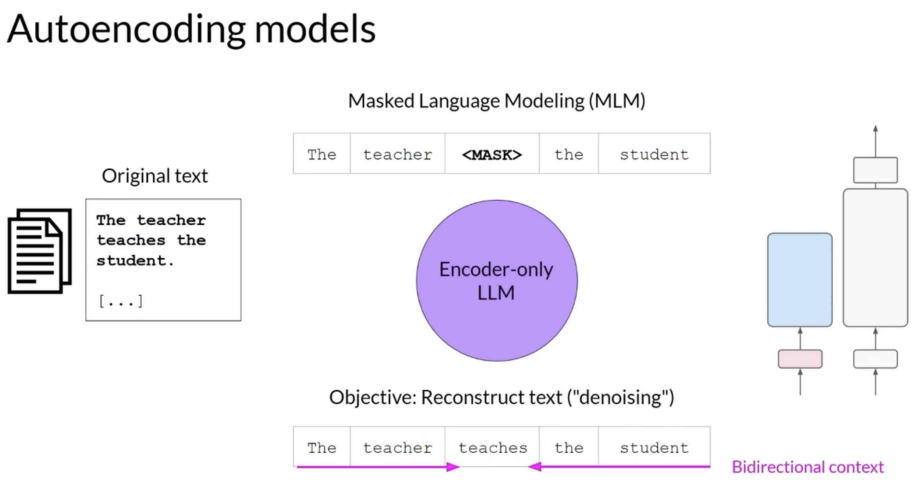
- The initial training process for LLMs is often referred to as pre-training.
- LLMs encode a deep statistical representation of language. This understanding is developed during the models pre-training phase when model learns from vast amounts of unstructured textual data(gigabytes, terabytes, and even petabytes of text). This data is pulled from many sources, including scrapes off the Internet and corpora of texts that have been assembled specifically for training language models.

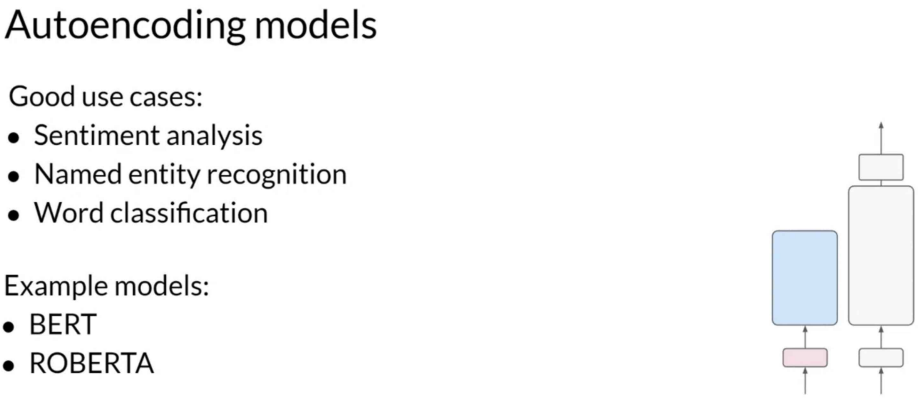
★- In this <u>self-supervised learning step, the model internalizes the patterns and structures present in the language.</u> These patterns then enable the model to complete its training objective, which depends on the architecture of the model.
- During pre-training, the model weights get updated to minimize the loss of the training objective

- There are three variants of the transformer model; encoder-only, encoder-decoder models, and decode-only. Each of this is trained on a different objective and they learn how to carry out different tasks

1. **Encoder-only models :**

- Also known as Autoencoding models and are pre-trained using masked language modeling (MLM)
- Here, tokens in the input sequence are randomly masked, and the training objective is to *predict the masked tokens in order to reconstruct the original sentence.* This is also called a *denoising* objective.
- The model has an understanding of the full context of a token and not just of the words that come before.
- These models are ideally suited to task that can have benefit from bi-directional contexts.

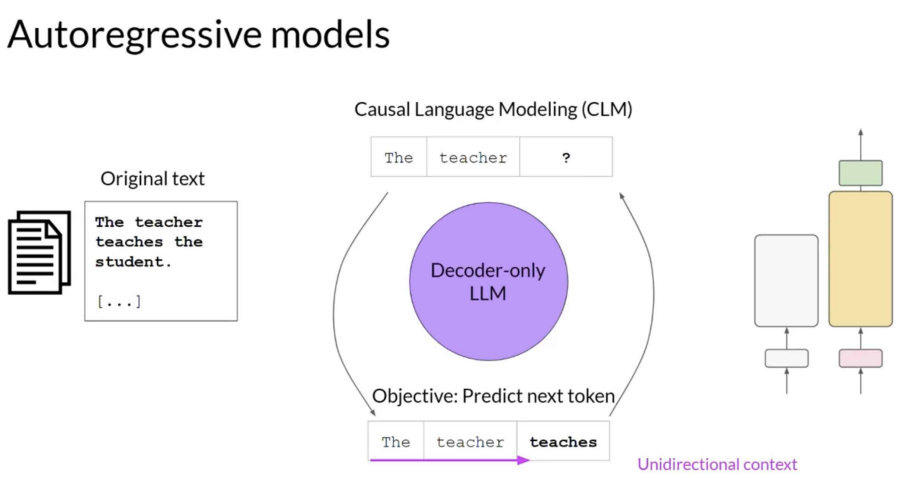


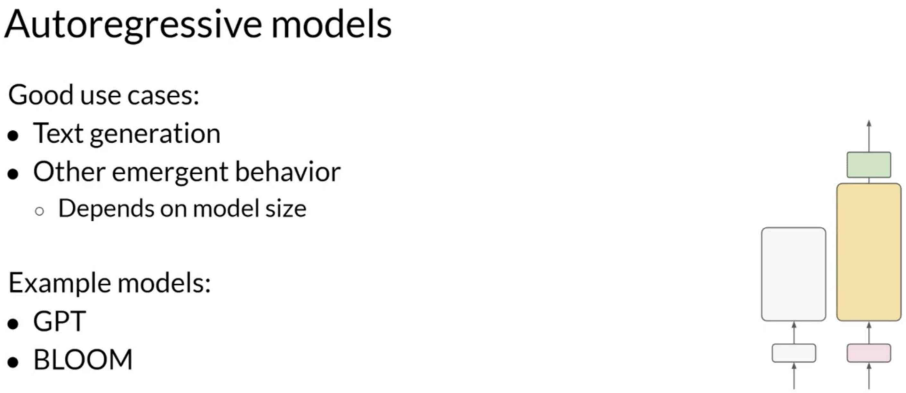


2. **Decoder-only models:**

- decoder-only or autoregressive models are pre-trained using causal language modeling (CLM).
- Here, the training objective is to *predict the next token based on the previous sequence of tokens*. Predicting the next token is sometimes called full language modeling by researchers.
- This model masks the input sequence and can only see the input tokens leading up to the token in question. The model has no knowledge of the end of the sentence. The model then iterates over the input sequence one by one to predict the following token.

- context is unidirectional.

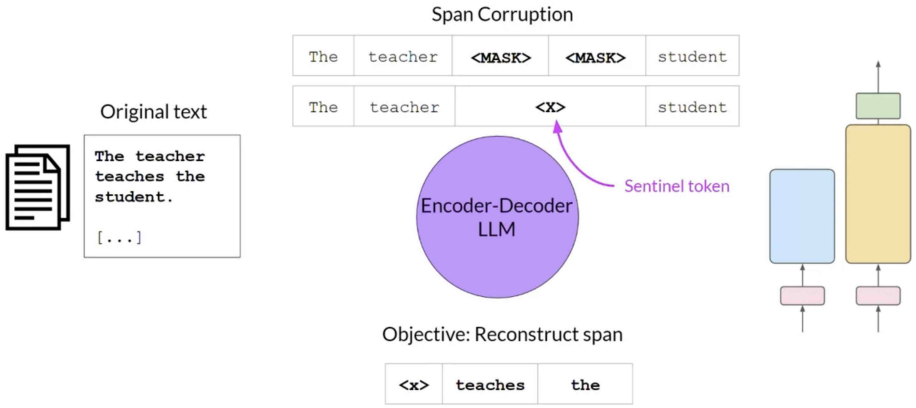




3. Encoder-decoder models:

- Sequence-to-sequence model that uses both the encoder and decoder parts of the original transformer architecture.
- A popular sequence-to-sequence model T5, pre-trains the encoder using *span corruption,* which masks random sequences of input tokens.
- Those mass sequences are then replaced with a unique Sentinel token, shown here as x. 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 mask token sequences auto-regressively. The output is the Sentinel token followed by the predicted tokens.

Sequence-to-sequence models



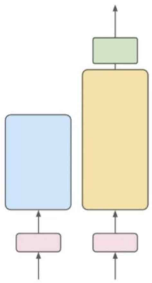
Sequence-to-sequence models

Good use cases:

- Translation
- Text summarization
- Question answering

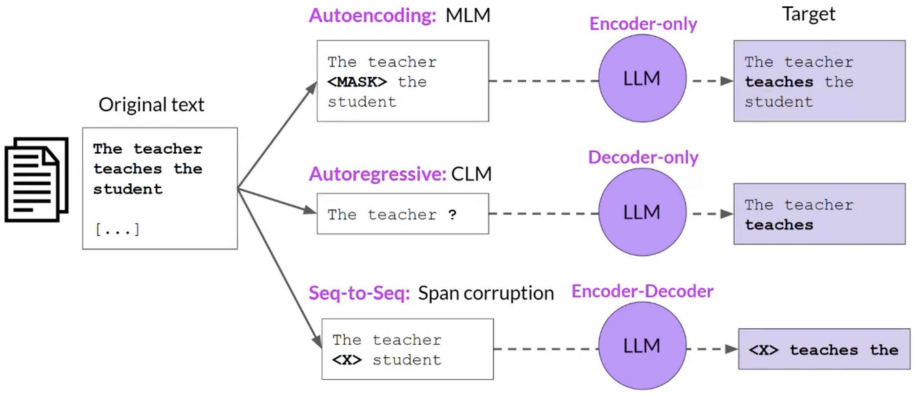
Example models:

- T5
- BART



Summary:

Model architectures and pre-training objectives



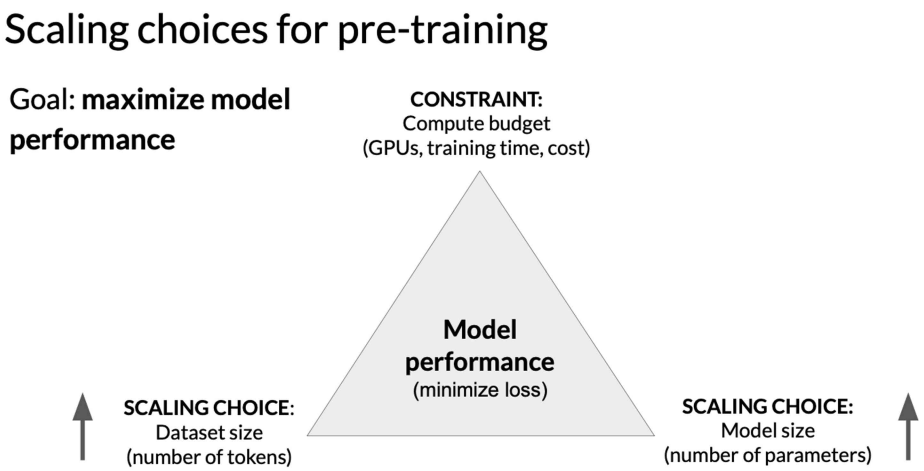
Computational challenges of training LLMs

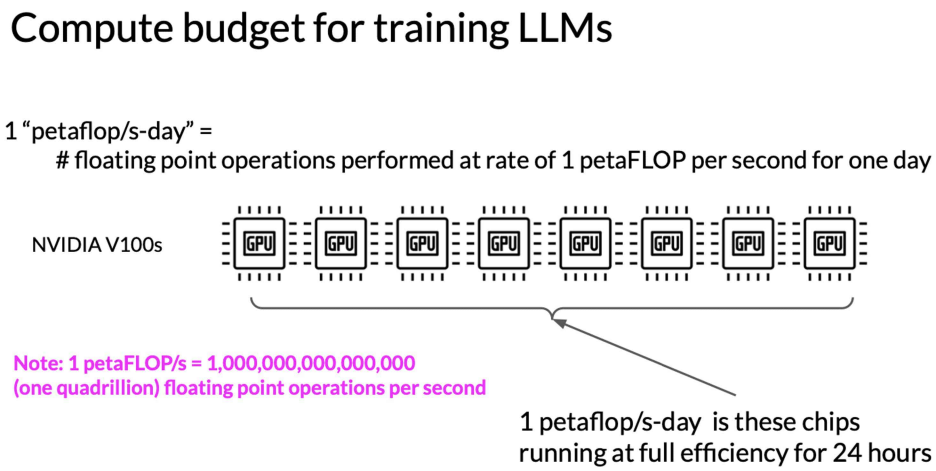
Data parallelism is a strategy that splits the training data across multiple GPUs. Each GPU processes a different subset of the data simultaneously, which can greatly speed up the overall training time.

Scaling laws and compute-optimal models

- the goal during pre-training is to maximize the model's performance of its learning objective, which is minimizing the loss when predicting tokens

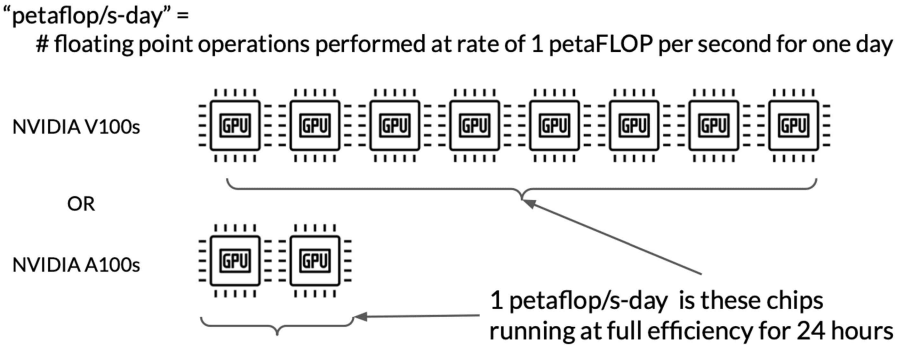
- Two options to achieve better performance:
  - increasing the size of the dataset you train your model on
  - increasing the number of parameters in your model





★- A petaFLOP per second day is a measurement of the number of floating point operations performed at a rate of one petaFLOP per second, running for an entire day.

    - Note, one petaFLOP corresponds to one quadrillion floating point operations per second.
- one petaFLOP per second day is approximately equivalent to eight NVIDIA V100 GPUs, operating at full efficiency for one full day.



- If you have a more powerful processor that can carry out more operations at once, then a petaFLOP per second day requires fewer chips. For example, two NVIDIA A100 GPUs give equivalent compute to the eight V100 chips.

***

- FLOP stands for Floating Point Operations Per Second—a measure of how many mathematical calculations (like additions, multiplications) a computer can perform in one second.
- A petaFLOP (PFLOP) means 10^15 FLOPs, or 1 quadrillion (1,000,000,000,000,000) operations per second.

What is PFLOP/s-day?

PFLOP/s-day means a system runs at **1 petaFLOP per second for an entire day (86,400 seconds).**

Mathematically, it equals:

$$1\ \text{PFLOP/s} \times 86,400\ \text{seconds} = 86,400 \times 10^{15}\ \text{FLOPs}$$

That's **86.4 exaFLOPs (EFLOPs), or 8.64×10^{19} operations per day.**

. Example for Perspective

- Suppose you have a **supercomputer** that operates at **2 PFLOP/s** (twice the power of 1 PFLOP/s).
- In a day, it would perform:

$$2 \times 86,400 \times 10^{15} = 172.8\ \text{EFLOPs}$$

- If a normal **laptop** performs **10 GFLOP/s (10 billion FLOPs per second)**, it would take:

$$\frac{86,400 \times 10^{15}}{10 \times 10^{9}} = 8.64 \times 10^{9}\ \text{seconds} \approx 274\ \text{years}$$

for the laptop to do what a **1 PFLOP/s** supercomputer does in just one day!

Relation to Other Units:

Unit	FLOPS Equivalent
1 GFLOP (GigaFLOP)	10^9 (1 billion FLOPS)
1 TFLOP (TeraFLOP)	10^{12} (1 trillion FLOPS)
1 PFLOP (PetaFLOP)	10^{15} (1 quadrillion FLOPS)
1 EFLOP (ExaFLOP)	10^{18} (1 quintillion FLOPS)

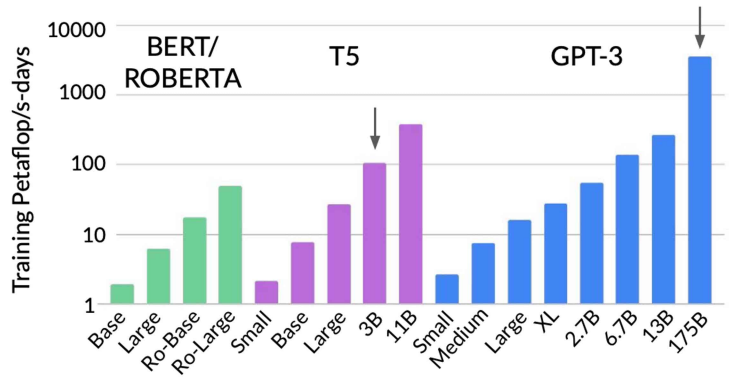
NVIDIA A100 Performance

Precision Type	Performance (TFLOPS)
FP64 (Double-Precision)	**9.7 TFLOPS**
FP32 (Single-Precision)	**19.5 TFLOPS**
TF32 (Tensor Float 32, AI/ML optimized)	**156 TFLOPS**
FP16 (Half-Precision, AI/Deep Learning)	**312 TFLOPS**
INT8 (Integer, AI Inference)	**1,248 TOPS** (Trillions of Operations Per Secc

The A100 is designed for **high-performance computing (HPC), AI, and deep learning**,

- In below figure, the difference between the models in each family is the number of parameters that were trained, ranging from a few hundred million for Bert base to 175 billion for the largest GPT-3 variant.

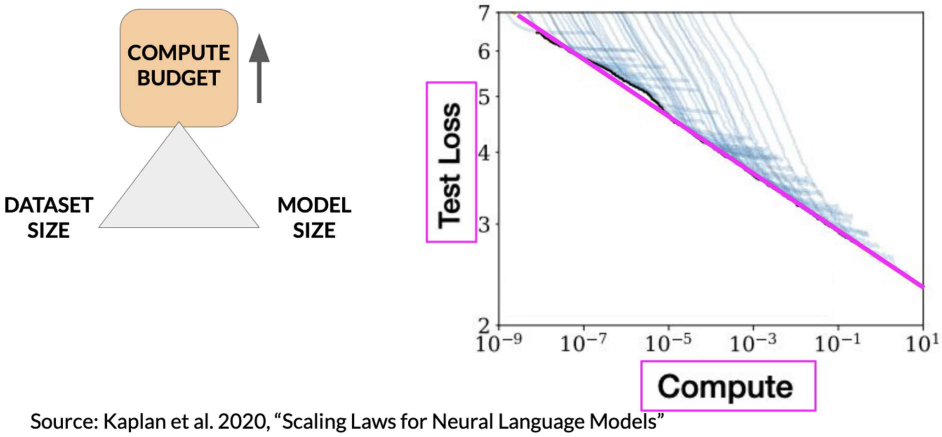
Number of petaflop/s-days to pre-train various LLMs



Source: Brown et al. 2020, "Language Models are Few-Shot Learners"

- Note that the y-axis is logarithmic. Each increment vertically is a power of 10.
- Here we see that T5 XL with three billion parameters required close to 100 petaFLOP per second days. While the larger GPT-3 175 billion parameter model required approximately 3,700 petaFLOP per second days.
- you can see that bigger models take more compute resources to train and generally also require more data to achieve good performance.
- Researchers have explored the trade-offs between training dataset size, model size and compute budget

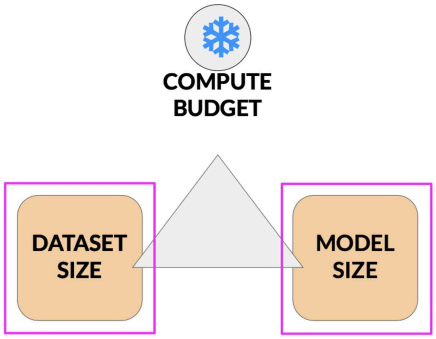
Compute budget vs. model performance



Source: Kaplan et al. 2020, "Scaling Laws for Neural Language Models"

- y-axis is the test loss and the x-axis is the compute budget in units of petaFLOP per second days
- If you hold your compute budget fixed, the two levers you have to improve your model's performance are the size of the training dataset and the number of parameters in your model

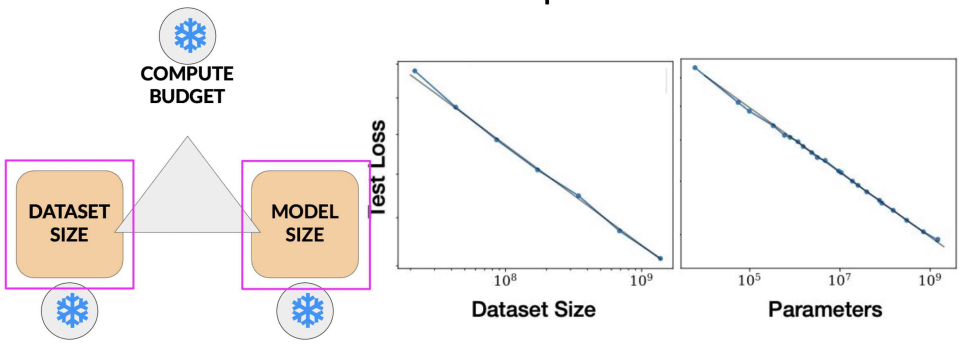
Dataset size and model size vs. performance



Compute resource constraints

- Hardware
- Project timeline
- Financial budget

Dataset size and model size vs. performance



- The OpenAI researchers found that these two quantities also show a power-law relationship with a test loss in the case where the other two variables are held fixed.

- Once the compute budget and model size are held fixed and the size of the training

dataset is vary. The graph shows that as the volume of training data increases, the performance of the model continues to improve.

- In the second graph, the compute budget and training dataset size are held constant. Models of varying numbers of parameters are trained. As the model increases in size, the test loss decreases indicating better performance.

- In a paper published in 2022, the goal was to find the optimal number of parameters and volume of training data for a given compute budget. The author's named the resulting compute optimal model, *Chinchilla*. This paper is often referred to as the Chinchilla paper.

Chinchilla paper

-

Jordan et al. 2022

Training Compute-Optimal Large Language Models

Jordan Hoffmann*, Sebastian Borgeaud*, Arthur Mensch*, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals and Laurent Sifre*

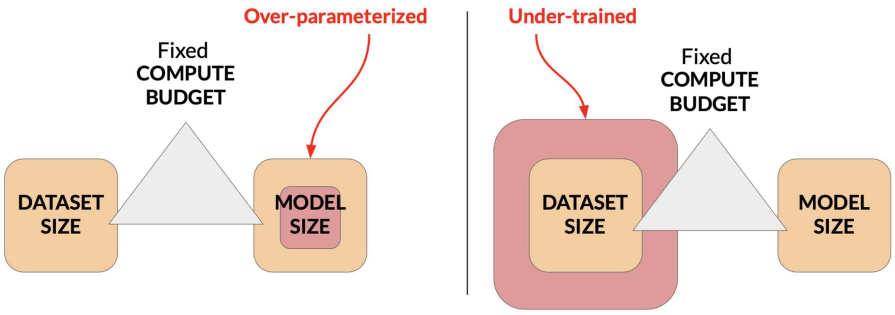
*Equal contributions

We investigate the optimal model size and number of tokens for training a transformer language model under a given compute budget. We find that current large language models are significantly under-trained, a consequence of the recent focus on scaling language models whilst keeping the amount of training data constant. By training over 400 language models ranging from 70 million to over 16 billion parameters on 5 to 500 billion tokens, we find that for compute-optimal training, the model size and the number of training tokens should be scaled equally: for every doubling of model size the number of training tokens should also be doubled. We test this hypothesis by training a predicted compute-optimal model, *Chinchilla*, that uses the same compute budget as *Gopher* but with 70B parameters and 4× more more data. *Chinchilla* uniformly and significantly outperforms *Gopher* (280B), GPT-3 (175B), Jurassic-1 (178B), and Megatron-Turing NLG (530B) on a large range of downstream evaluation tasks. This also means that *Chinchilla* uses substantially less compute for fine-tuning and inference, greatly facilitating downstream usage. As a highlight, *Chinchilla* reaches a state-of-the-art average accuracy of 67.5% on the MMLU benchmark, greater than a 7% improvement over *Gopher*.

- below is the conclusion drawn from the paper

Compute optimal models

- Very large models may be **over-parameterized** and **under-trained**
- Smaller models trained on more data could perform as well as large models



- In below table you can see a selection of models along with their size and information about the dataset they were trained on.

Chinchilla scaling laws for model and dataset size

Model	# of parameters	Compute-optimal* # of tokens (~20x)	Actual # tokens
Chinchilla	70B	~1.4T	1.4T
LLaMA-65B	65B	~1.3T	1.4T
GPT-3	175B	~3.5T	300B
OPT-175B	175B	~3.5T	180B
BLOOM	176B	~3.5T	350B

Compute optimal training datasize is ~**20x** number of parameters

* assuming models are trained to be compute-optimal per Chinchilla paper

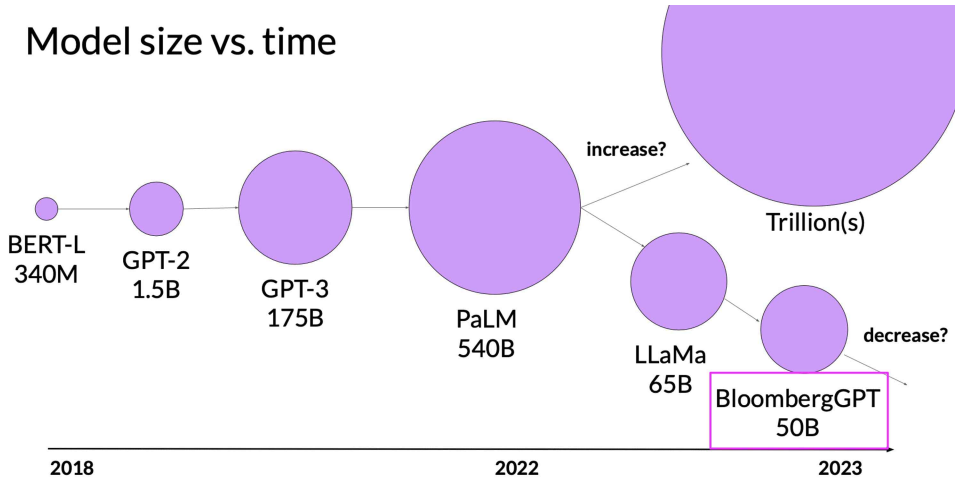
Sources: Hoffmann et al. 2022, "Training Compute-Optimal Large Language Models"
Touvron et al. 2023, "LLaMA: Open and Efficient Foundation Language Models"

- One important takeaway from the Chinchilla paper is that the optimal training dataset size for a given model is about 20 times larger than the number of parameters in the model.

- Chinchilla was determined to be compute optimal.
  - For a 70 billion parameter model, the ideal training dataset contains 1.4 trillion tokens or 20 times the number of parameters.
  - The last three models in the table were trained on datasets that are smaller than the Chinchilla optimal size. These models may actually be under trained.
  - In contrast, LLaMA was trained on a dataset size of 1.4 trillion tokens, which is close to the Chinchilla recommended number.

- Another important result from the paper is that the compute optimal Chinchilla model outperforms non compute optimal models such as GPT-3 on a large range of downstream evaluation tasks.

- Now you can probably expect to see a deviation from the bigger is always better trends of the last few years as more teams or developers like you start to optimize their model design.



☑ To scale our model, we need to jointly increase dataset size and model size, or they can become a bottleneck for each other.

✓ Correct
For instance, while increasing dataset size is helpful, if we do not jointly improve the model size, it might not be able to capture value from the larger dataset.

☑ There is a relationship between model size (in number of parameters) and the optimal number of tokens to train the model with.

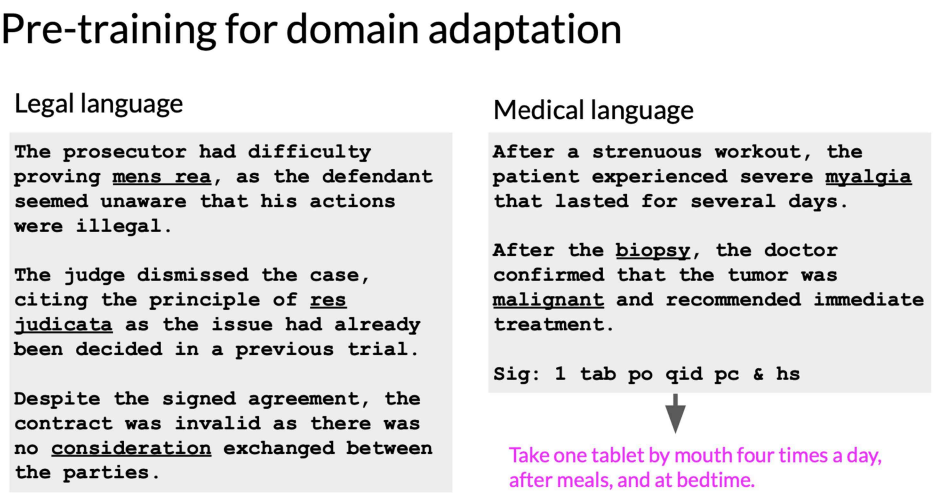
✓ Correct
This relationship is describe in the Chinchilla paper, that shows that many models might even be overparametrized according to the relationship they found.

☑ When measuring compute budget, we can use "PetaFlops per second-Day" as a metric.

✓ Correct
Petaflops per second-day is a useful measure for computing budget as it reflects the both hardware and time required to train the model.

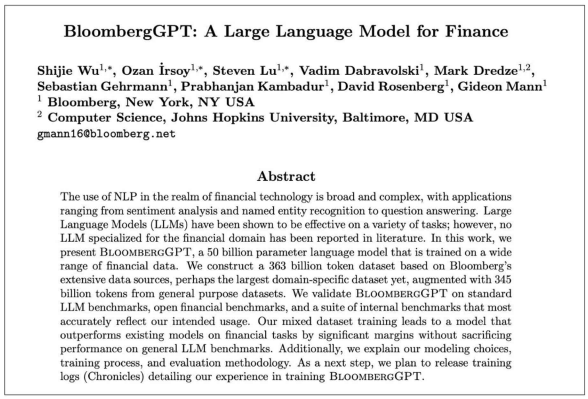
Pre-training for domain adaptation

- If your target domain uses vocabulary and language structures that are not commonly used in day to day language. You may need to perform domain adaptation to achieve good model performance.



! - Pretraining your model from scratch will result in better models for highly specialized domains like law, medicine, finance or science.

BloombergGPT: domain adaptation for finance



BloombergGPT: A Large Language Model for Finance

Shijie Wu[1,*], Ozan İrsoy[1,*], Steven Lu[1,*], Vadim Dabravolski[1], Mark Dredze[1,2], Sebastian Gehrmann[1], Prabhanjan Kambadur[1], David Rosenberg[1], Gideon Mann[1]
[1] Bloomberg, New York, NY USA
[2] Computer Science, Johns Hopkins University, Baltimore, MD USA
gmann16@bloomberg.net

Abstract

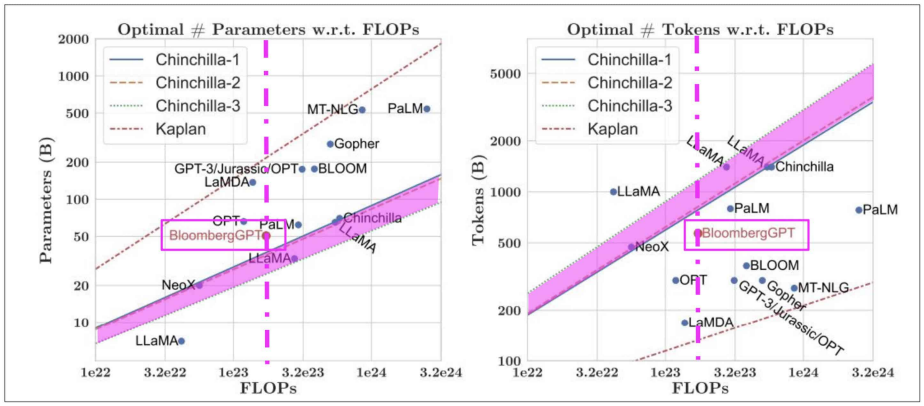
The use of NLP in the realm of financial technology is broad and complex, with applications ranging from sentiment analysis and named entity recognition to question answering. Large Language Models (LLMs) have been shown to be effective on a variety of tasks; however, no LLM specialized for the financial domain has been reported in literature. In this work, we present BloombergGPT, a 50 billion parameter language model that is trained on a wide range of financial data. We construct a 363 billion token dataset based on Bloomberg's extensive data sources, perhaps the largest domain-specific dataset yet, augmented with 345 billion tokens from general purpose datasets. We validate BloombergGPT on standard LLM benchmarks, open financial benchmarks, and a suite of internal benchmarks that most accurately reflect our intended usage. Our mixed dataset training leads to a model that outperforms existing models on financial tasks by significant margins without sacrificing performance on general LLM benchmarks. Additionally, we explain our modeling choices, training process, and evaluation methodology. As a next step, we plan to release training logs (Chronicles) detailing our experience in training BloombergGPT.

~51% **Financial (Public & Private)**

~49% **Other (Public)**

- BloombergGPT is an example of a large language model that has been pretrained for a specific domain, in this case, finance.
- Bloomberg researchers chose to combine both finance data and general purpose tax data to pretrain a model that achieves best results on financial benchmarks while also maintaining competitive performance on general purpose LLM benchmarks.

BloombergGPT relative to other LLMs



Source: Wu et al. 2023, "BloombergGPT: A Large Language Model for Finance"

- On the left, the diagonal lines trace the optimal model size in billions of parameters for a range of compute budgets. On the right, the lines trace the compute optimal training data set size measured in number of tokens.
- The dashed pink vertical line on each graph indicates the compute budget that the Bloomberg team had available for training their new model. The pink shaded regions correspond to the compute optimal scaling loss determined in the Chinchilla paper.

- In terms of model size, you can see that BloombergGPT roughly follows the Chinchilla approach for the given compute budget of 1.3 million GPU hours, or roughly 230,000,000 petaflops. The model is only a little bit above the pink shaded region, suggesting the number of parameters is fairly close to optimal. However, the actual number of tokens used to pretrain BloombergGPT 569,000,000,000 is below the recommended Chinchilla value for the available compute budget. The smaller than optimal training data set is due to the limited availability of financial domain data. Showing that real world constraints may force you to make trade offs when pretraining your own models.

- more details from paper:

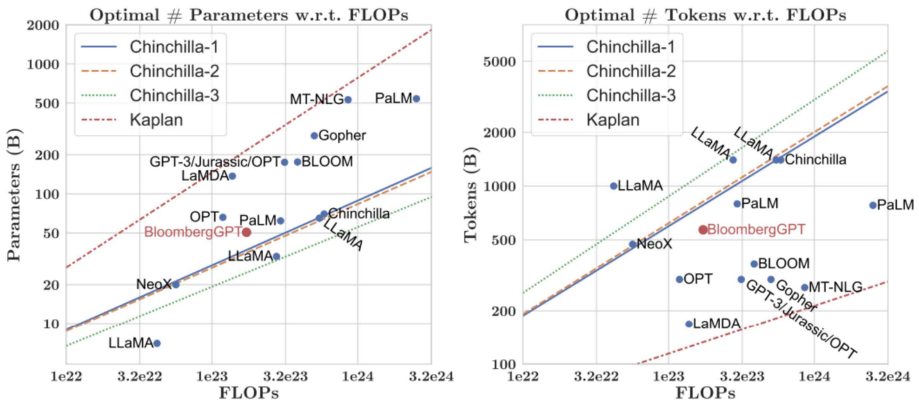


Figure 1: Kaplan et al. (2020) and Chinchilla scaling laws with prior large language model and BLOOMBERGGPT parameter and data sizes. We adopt the style from Hoffmann et al. (2022).

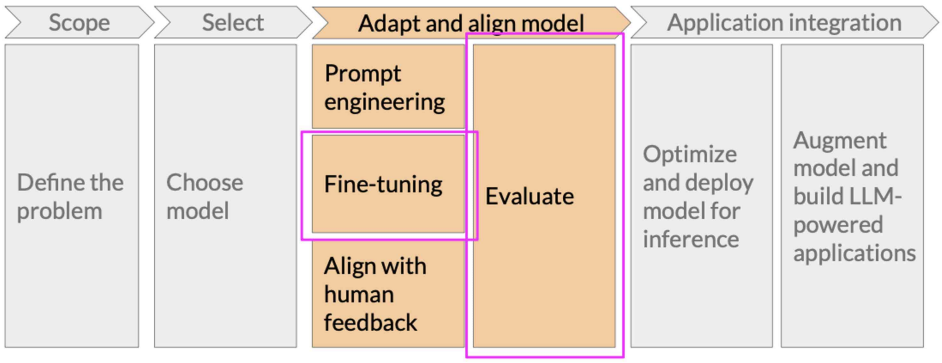
2 Dataset

W2: Fine-tuning LLMs with instruction tuning

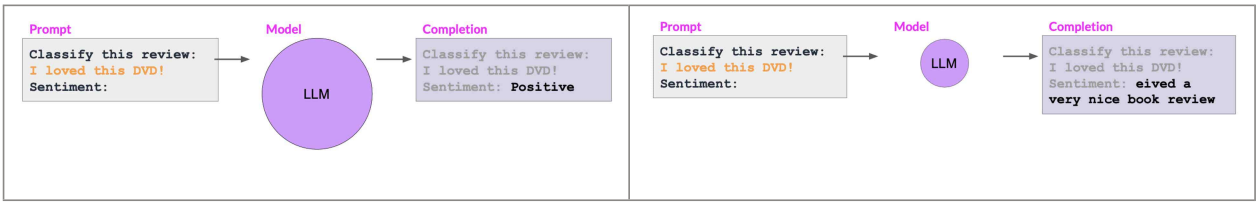
Tuesday, 13 August 2024 5:08 PM

Instruction fine-tuning

- here we will look at finetuning LLMs and evaluation metrics

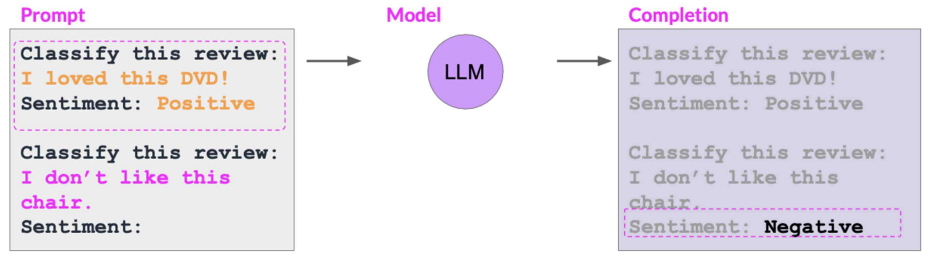


- some models are capable of identifying instructions contained in a prompt and correctly carrying out zero shot inference, while others, such as smaller LLMs, may fail to carry out the task, like the example shown here for zero shot.



- one shot or few shot inference, can be enough to help the model identify the task and generate a good completion.

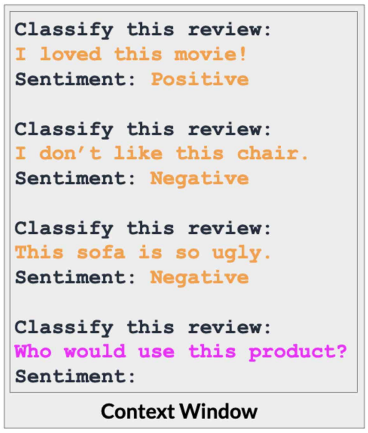
In-context learning (ICL) - one/few shot inference



One-shot or Few-shot Inference

- But this strategy has a couple of drawbacks mentioned below

Limitations of in-context learning

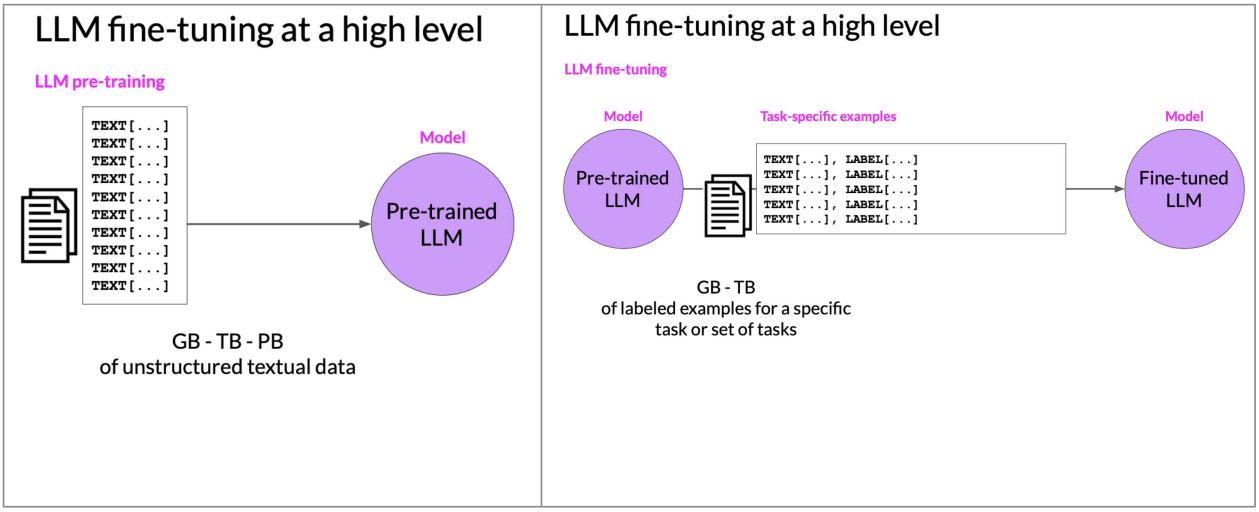


- In-context learning may not work for smaller models (LLM)
- Examples take up space in the context window

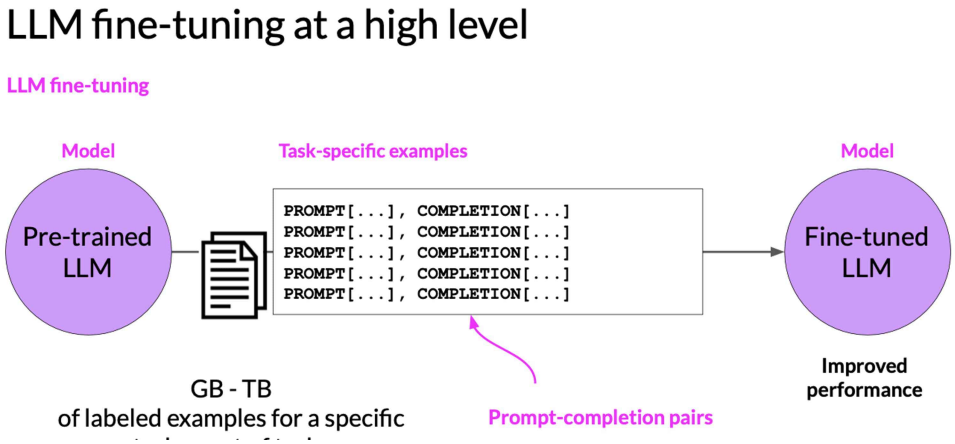
Instead, try **fine-tuning** the model

- Solution - fine-tuning (further train a base model)

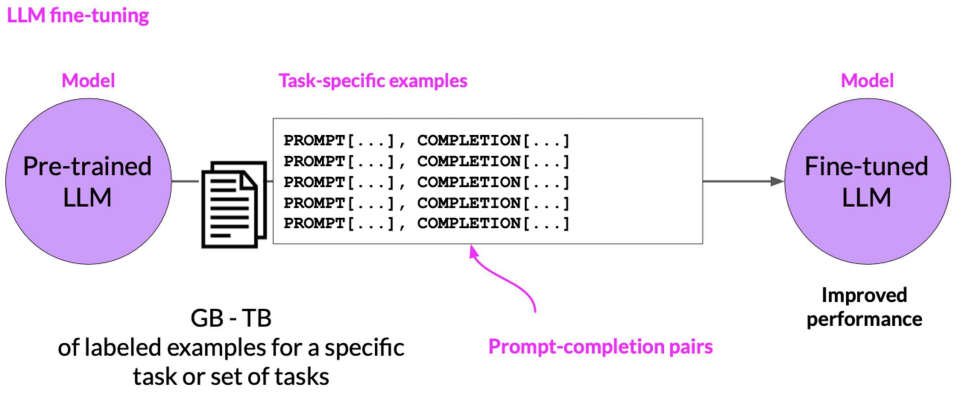
★- In contrast to pre-training, where you train the LLM using vast amounts of unstructured textual data via *self-supervised learning*, fine-tuning is a supervised learning process where you use a data set of labeled examples to update the weights of the LLM.



- The labeled examples are prompt-completion pairs, the fine-tuning process extends the training of the model to improve its ability to generate good completions for a specific task.

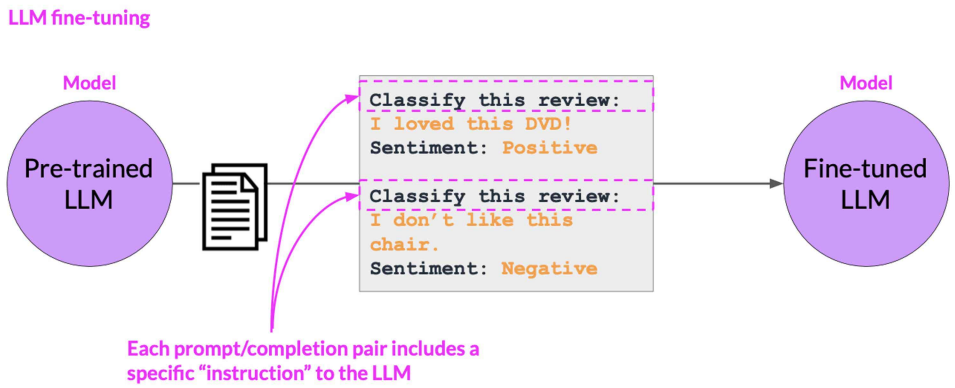


LLM fine-tuning at a high level



- One strategy, known as *instruction fine-tuning*, is particularly good at improving a model's performance on a variety of tasks.

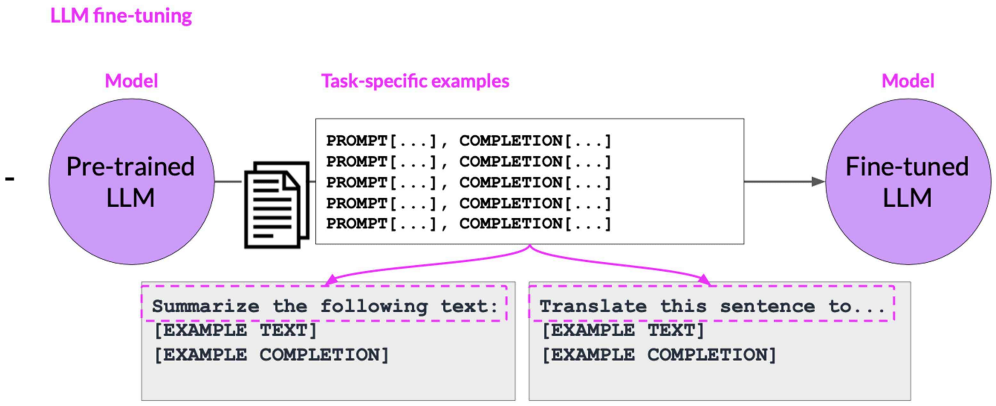
Using prompts to fine-tune LLMs with instruction



- In above example, the instruction in both examples is "classify this review", and the desired completion is a text string that starts with "sentiment" followed by either positive or negative. The data set you use for training includes many pairs of prompt completion examples for the task you're interested in, each of which includes an instruction.

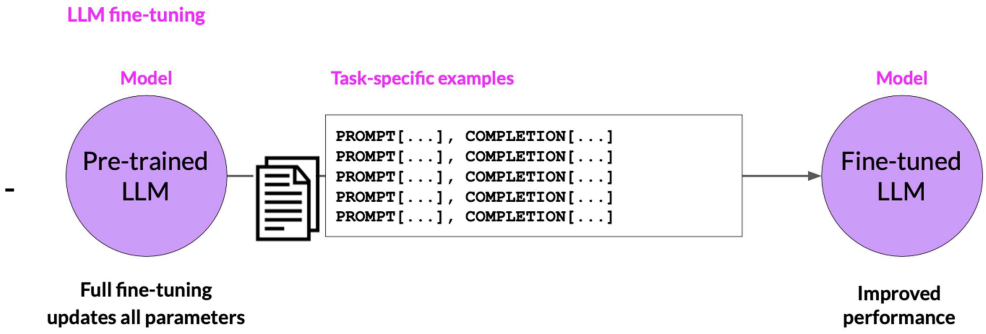
- If you want to fine tune your model to improve its summarization ability, you'd build up a data set of examples that begin with the instruction - "summarize the following text" or a similar phrase. And if you are improving the model's translation skills, your examples would include instructions like "translate this sentence"

Using prompts to fine-tune LLMs with instruction

- 

★- These prompt completion examples allow the model to learn to generate responses that follow the given instructions.

- <u>The instruction fine-tuning, where all of the model's weights are updated is known as *full fine-tuning*.</u> The process results in a new version of the model with updated weights.

- 

<u>Training Instruction dataset:</u>

- Prompt template libraries include many templates for different tasks and different data sets. Here are three prompts that are designed to work with the Amazon reviews dataset and that can be used to fine tune models for classification, text generation and text summarization tasks.

- dataset : In each case you pass the original review, here called review_body, to the template, where it gets inserted into the text that starts with an instruction like predict the associated rating, generate a star review, or give a short sentence describing the following product review. The result is a prompt that now contains both an instruction and the example from the data set.

Sample prompt instruction templates

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

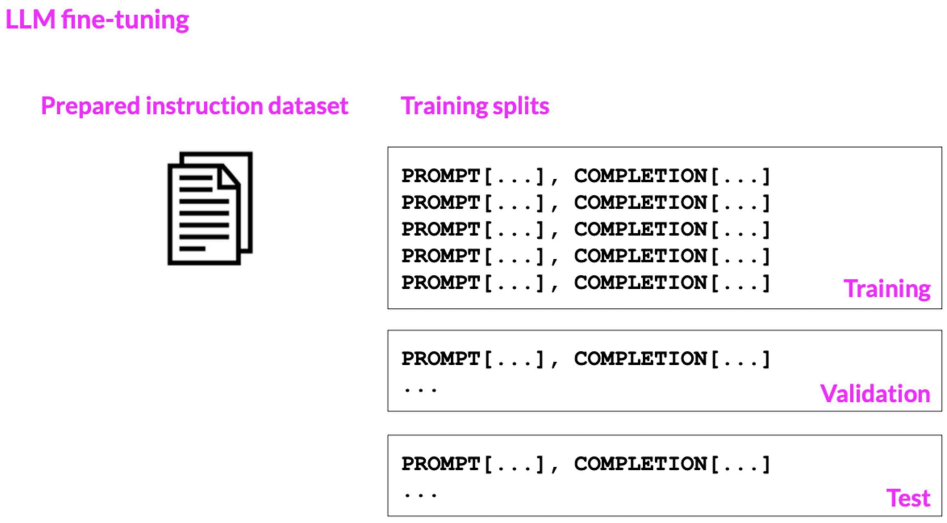
```
jinja: Generate a {{star_rating}}-star review (1 being lowest and 5 being highest)
  about this product {{product_title}}.  |||  {{review_body}}
```

Text summarization

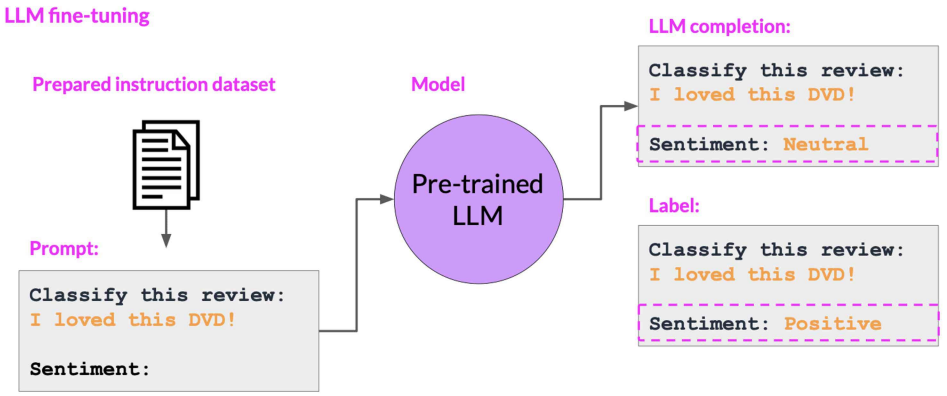
```
jinja: "Give a short sentence describing the following product review: \n{{review_body}}\
  \ \n|||\n{{review_headline}}"
```

Source: https://github.com/bigscience-workshop/promptsource/blob/main/promptsource/templates/amazon_polarity/templates.yaml

- Once you have your instruction data set ready, as with standard supervised learning, you divide the data set into training validation and test splits.



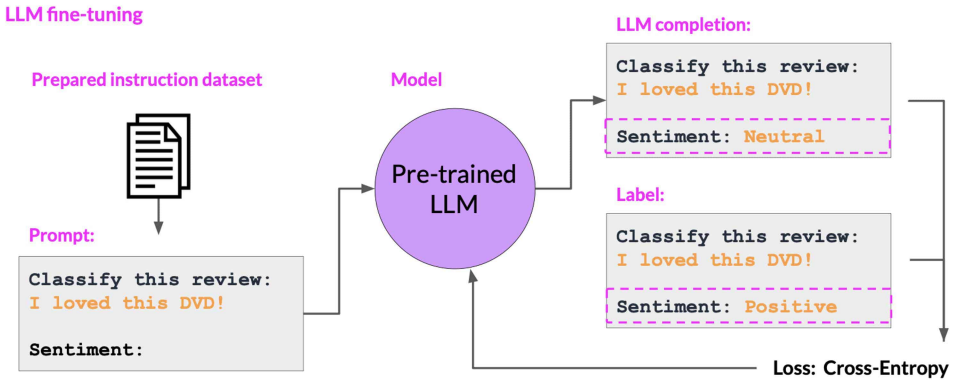
★- During fine tuning, you select prompts from your training data set and pass them to the LLM, which then generates completions.
- Next, you compare the LLM completion with the response specified in the training data.



- You can see here that the model didn't do a great job, it classified the review as neutral, but the review is clearly very positive.

★- Remember that the output of an LLM is a probability distribution across tokens. So you can compare the distribution of the completion with the training label and use the standard cross-entropy function to calculate loss between the two token distributions. And then use the calculated loss to update your model weights in standard backpropagation. You'll do this for many batches of prompt completion pairs and over several epochs, update the weights so that the model's performance on the task improves.

-



- As in standard supervised learning, you can define separate evaluation steps to measure your LLM performance using the holdout validation data set. This will give you the validation accuracy, and after you've completed your fine tuning, you can perform a final performance evaluation using the holdout test data set. This will give you the test accuracy.
- The fine-tuning process results in a new version of the base model, often called an ***instruct model*** that is better at the tasks you are interested in.

Fine-tuning on a single task

Catastrophic forgetting occurs when a machine learning model forgets previously learned information as it learns new information.

Correct
The assertion is true, and this process is especially problematic in sequential learning scenarios where the model is trained on multiple tasks over time.

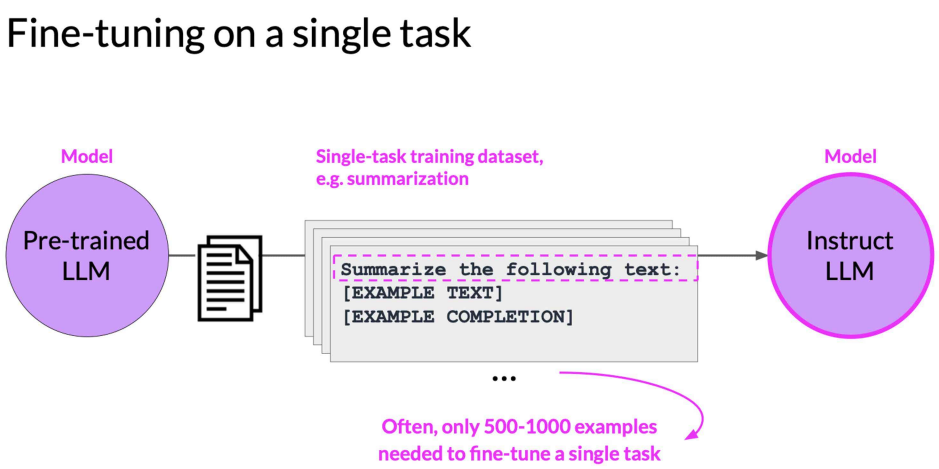
One way to mitigate catastrophic forgetting is by using regularization techniques to limit the amount of change that can be made to the weights of the model during training.

Correct
One way to mitigate catastrophic forgetting is by using regularization techniques to limit the amount of change that can be made to the weights of the model during training. This can help to preserve the information learned during earlier training phases and prevent overfitting to the new data.

Catastrophic forgetting is a common problem in machine learning, especially in deep learning models.

Correct
This assertion is true because these models typically have many parameters, which can lead to overfitting and make it more difficult to retain previously learned information.

- LLMs have the ability to perform many different language tasks within a single model, but your application may only need to perform a single task. In this case, you can fine-tune a pre-trained model to improve performance on only the task.

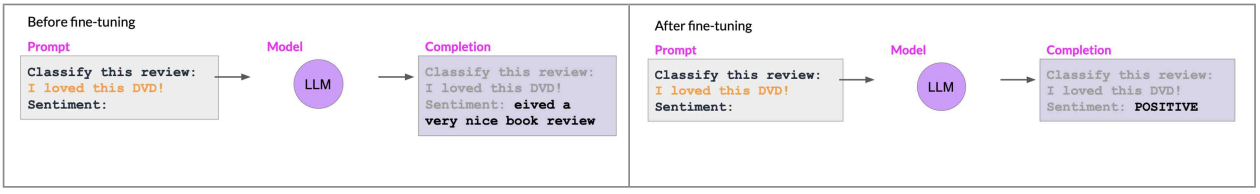


- For example, summarization using a dataset of examples for that task. Here good results can be achieved with relatively few examples, often just 500-1,000 examples can result in good performance in contrast to the billions of pieces of texts that the model saw during pre-training.
- However, there is a potential downside to fine-tuning on a single task. The process may lead to a phenomenon called *catastrophic forgetting.*

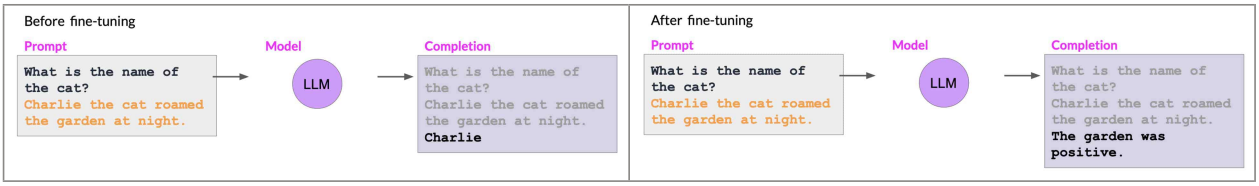
★- Catastrophic forgetting happens because the full fine-tuning process modifies the weights of the original LLM. While this leads to great performance on the single fine-tuning task, it can degrade performance on other tasks

- Fine-tuning can significantly increase the performance of a model on a specific task but can lead to reduction in ability on other tasks

result on specific task:



result on some other task:



- For example, while fine-tuning can improve the ability of a model to perform sentiment analysis on a review and result in a quality completion, the model may forget how to do other tasks. This model knew how to carry out named entity recognition before fine-tuning correctly identifying Charlie as the name of the cat in the sentence. But after fine-tuning, the model can no longer carry out this task, confusing both the entity it is supposed to identify and exhibiting behavior related to the new task.

How to avoid catastrophic forgetting

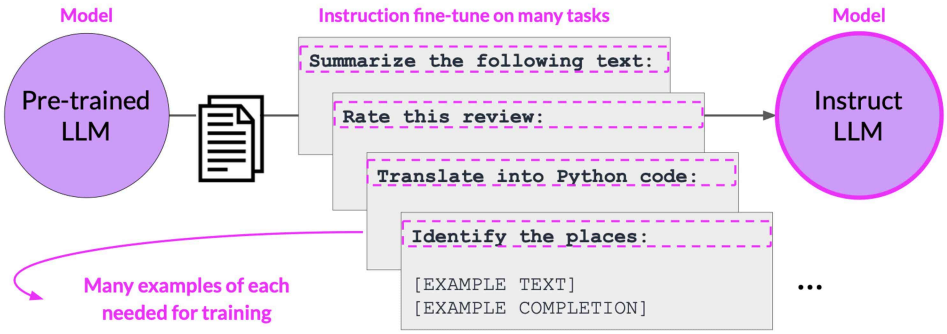
- First note that you might not have to!
- Fine-tune on **multiple tasks** at the same time
- Consider **Parameter Efficient Fine-tuning** (PEFT)

- PEFT is a set of techniques that preserves the weights of the original LLM and trains only a small number of task-specific adapter layers and parameters. PEFT shows greater robustness to catastrophic forgetting since most of the pre-trained weights are left unchanged.

Multi-task instruction fine-tuning

- Here, the dataset contains examples that instruct the model to carry out a variety of tasks, including summarization, review rating, code translation, and entity recognition.
- You train the model on this mixed dataset so that it can improve the performance of the model on all the tasks simultaneously, thus avoiding the issue of catastrophic forgetting.
- Over many epochs of training, the calculated losses across examples are used to update the weights of the model, resulting in an instruction tuned model that is learned how to be good at many different tasks simultaneously.

Multi-task, instruction fine-tuning

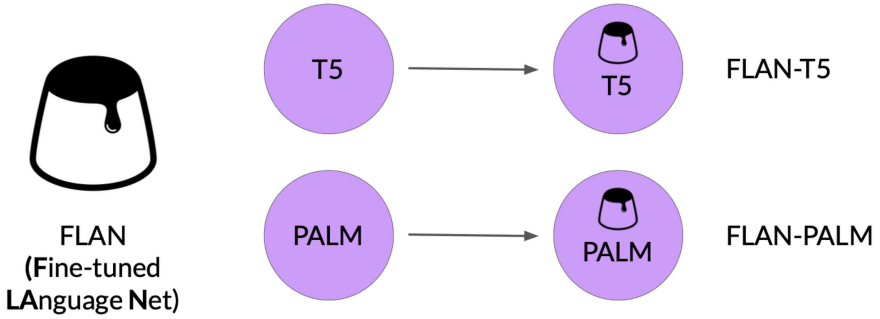


- drawback of multi-task fine-tuning - requires a lot of data. You may need as many as 50-100,000 examples in your training set.

- One family of models that have been trained using multitask instruction fine-tuning is FLAN family of models. FLAN stands for Fine-tuned LAnguage Net, is a specific set of instructions used to fine-tune different models. fine-tuning is the last step of the training process in these models.

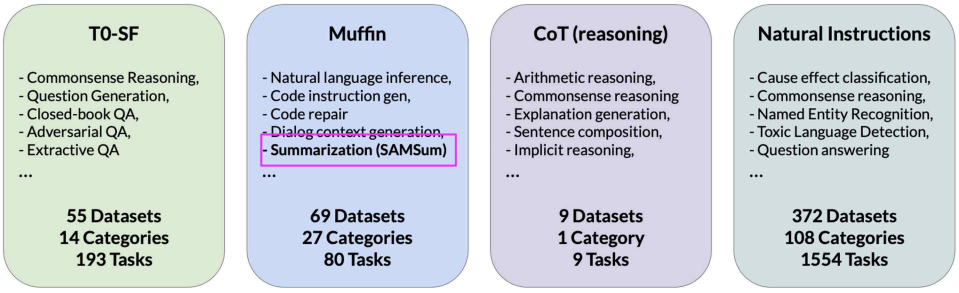
Instruction fine-tuning with FLAN

- FLAN models refer to a specific set of instructions used to perform instruction fine-tuning



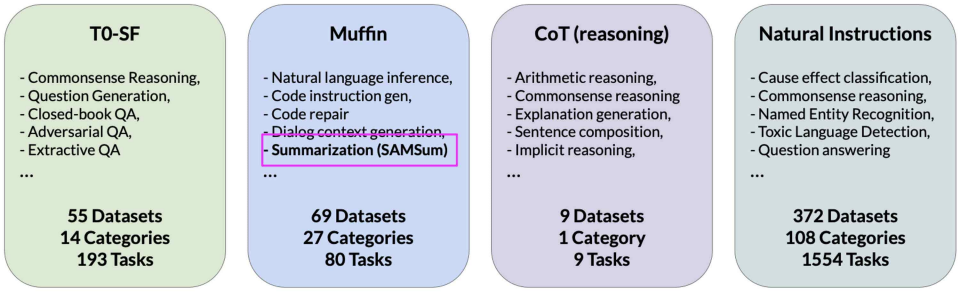
FLAN-T5: Fine-tuned version of pre-trained T5 model

- FLAN-T5 is a great, general purpose, instruct model



FLAN-T5: Fine-tuned version of pre-trained T5 model

- FLAN-T5 is a great, general purpose, instruct model



Source: Chung et al. 2022, "Scaling Instruction-Finetuned Language Models"

SAMSum: A dialogue dataset

Sample prompt training dataset (**samsum**) to fine-tune FLAN-T5 from pretrained T5

Datasets: **samsum** | Tasks: Summarization | Languages: English

dialogue (string)	summary (string)
"Amanda: I baked cookies. Do you want some? Jerry: Sure! Amanda: I'll bring you tomorrow :-)"	"Amanda baked cookies and will bring Jerry some tomorrow."
"Olivia: Who are you voting for in this election? Oliver: Liberals as always. Olivia: Me too!! Oliver: Great"	"Olivia and Olivier are voting for liberals in this election. "
"Tim: Hi, what's up? Kim: Bad mood tbh, I was going to do lots of stuff but ended up procrastinating Tim: What did…	"Kim may try the pomodoro technique recommended by Tim to get more stuff done."

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}"),
]
```

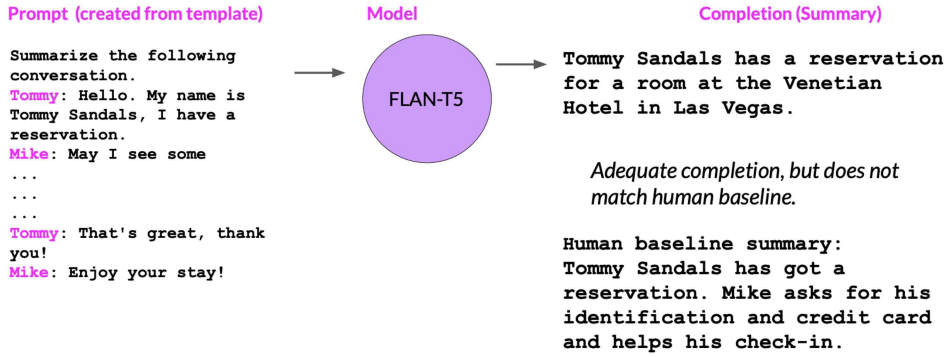
- Including different ways of saying the same instruction helps the model generalize and perform better.
- After applying this template to each row in the SAMSum dataset, you can use it to fine tune a dialogue summarization task.

Sample FLAN-T5 prompt templates

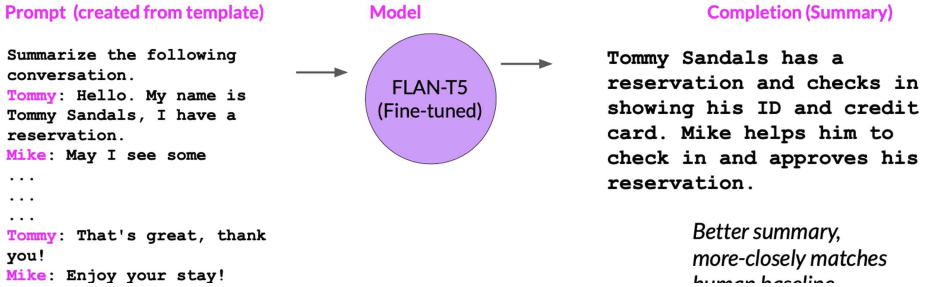
```
"samsum": [
      ("{dialogue}\n\nBriefly summarize that dialogue.", "{summary}"),
       ("Here is a dialogue:\n{dialogue}\n\nWrite a short summary!",
        "{summary}"),
       ("Dialogue:\n{dialogue}\n\nWhat is a summary of this dialogue?",
        "{summary}"),
       ("{dialogue}\n\nWhat was 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}"),
]
```

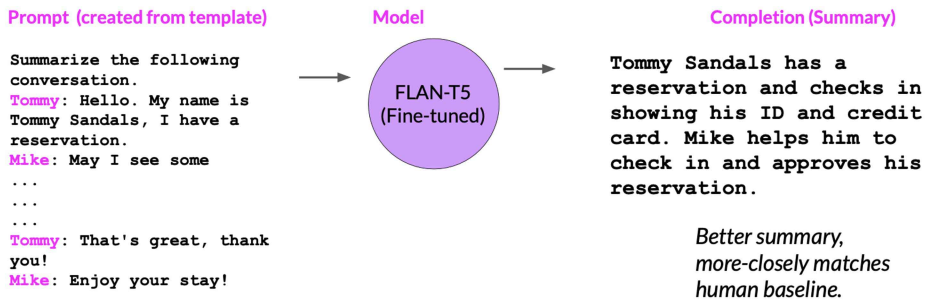
- We will make use of an additional domain specific summarization dataset called dialogsum to improve FLAN-T5's is ability to summarize support chat conversations. This dataset consists of over 13,000 support chat dialogues and summaries. The dialogsum dataset is not part of the FLAN-T5 training data, so the model has not seen these conversations before.

Summary **before** fine-tuning FLAN-T5 with our dataset

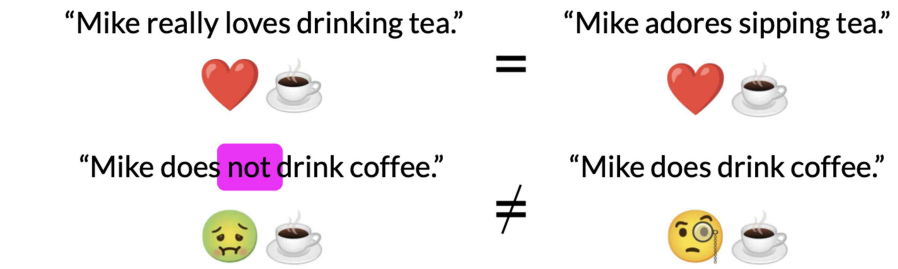


Summary **after** fine-tuning FLAN-T5 with our dataset





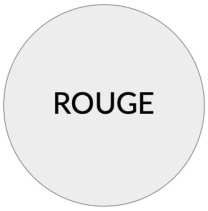
Model evaluation



- With large language models the output is non-deterministic and language-based evaluation is much more challenging. consider above examples (Take, for example, the sentence, Mike really loves drinking tea. This is quite similar to Mike adores sipping tea. But how do you measure the similarity? Let's look at these other two sentences. Mike does not drink coffee, and Mike does drink coffee. There is only one word difference between these two sentences. However, the meaning is completely different.)

- when you train a model on millions of sentences, you need an automated, structured way to make measurements. ROUGE and BLEU, are two widely used evaluation metrics for different tasks.

LLM Evaluation - Metrics

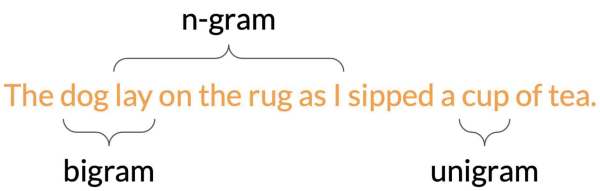


- Used for text summarization
- Compares a summary to one or more reference summaries

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

- ROUGE or recall oriented understudy for jesting evaluation is primarily employed to assess the quality of automatically generated summaries by comparing them to human-generated reference summaries.
- BLEU, or bilingual evaluation understudy is an algorithm designed to evaluate the quality of machine-translated text by comparing it to human-generated translations.

- a unigram is equivalent to a single word. A bigram is two words and n-gram is a group of n-words.



Rouge-1 Score:

LLM Evaluation - Metrics - ROUGE-1

Reference (human):
It is cold outside.

Generated output:
It is very cold outside.

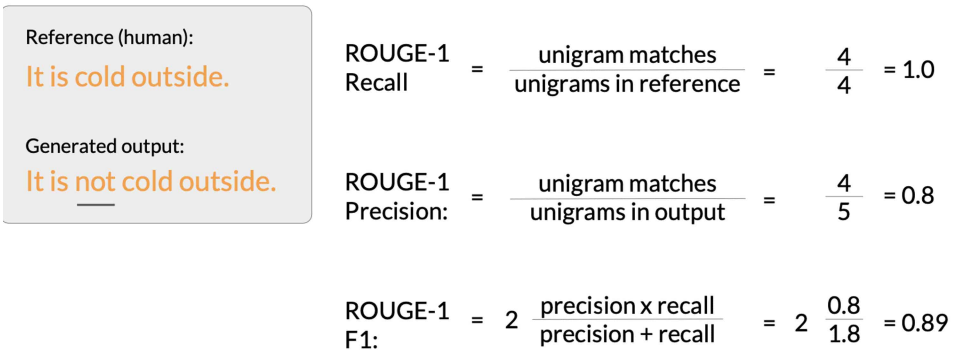
$$\text{ROUGE-1 Recall} = \frac{\text{unigram matches}}{\text{unigrams in reference}} = \frac{4}{4} = 1.0$$

$$\text{ROUGE-1 Precision:} = \frac{\text{unigram matches}}{\text{unigrams in output}} = \frac{4}{5} = 0.8$$

$$\text{ROUGE-1 F1:} = 2\,\frac{\text{precision x recall}}{\text{precision + recall}} = 2\,\frac{0.8}{1.8} = 0.89$$

- in below example even though we added "not" word which changed the meaning of the sentence, still the rogue-1 scores are same.

LLM Evaluation - Metrics - ROUGE-1

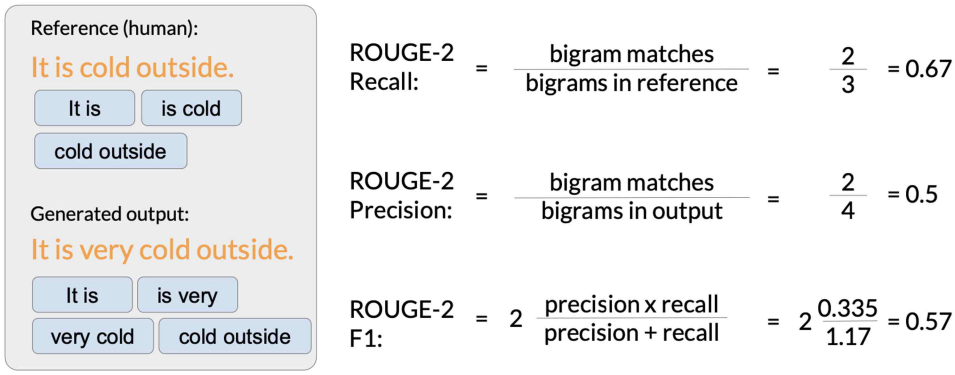


- You can get a slightly better score by taking into account bigrams or collections of two words at a time from the reference and generated sentence

Rouge-2 Score:

- By using bigrams, you're able to calculate a ROUGE-2.

LLM Evaluation - Metrics - ROUGE-2



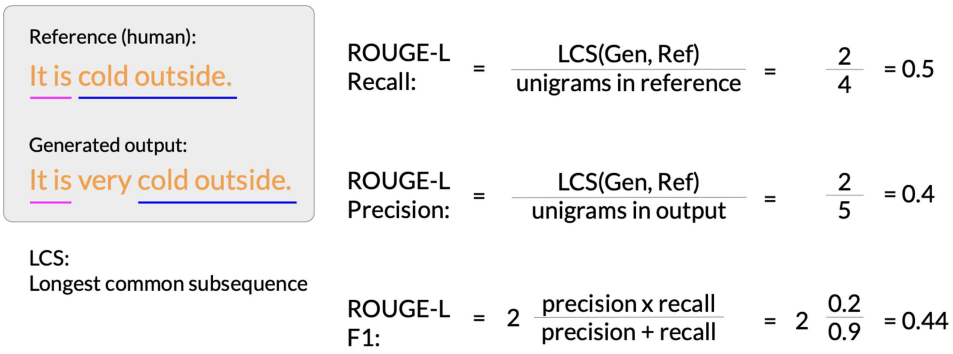
- Notice that the scores are lower than the ROUGE-1 scores. With longer sentences, they're a greater chance that bigrams don't match, and the scores may be even lower

Rouge-L Score:

- This looks for the longest common subsequence present in both the generated output and the reference output.
- use the LCS value to calculate the recall, precision and F1 score, where the numerator in both the recall and precision calculations is the length of the longest common subsequence. Collectively, these three quantities are known as the Rouge-L score.

LLM Evaluation - Metrics - ROUGE-L

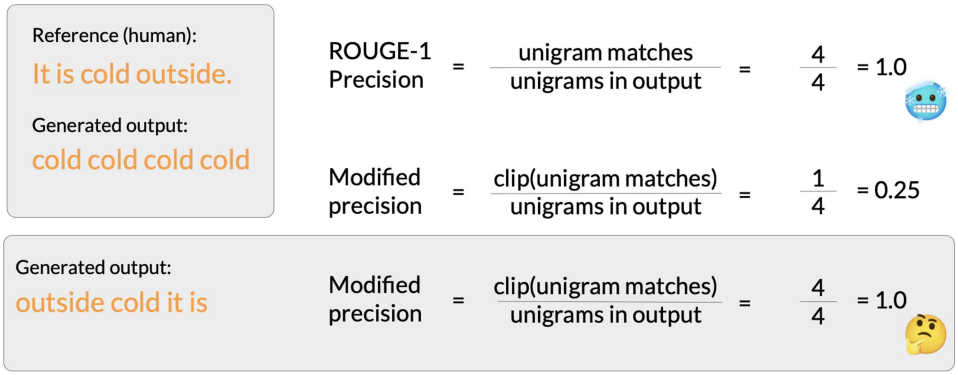
LLM Evaluation - Metrics - ROUGE-L



- As with all of the rouge scores, you need to take the values in context. You can only use the scores to compare the capabilities of models if the scores were determined for the same task. For example, summarization. Rouge scores for different tasks are not comparable to one another.

- clipping function is used to limit the number of unigram matches to the maximum count for that unigram within the reference.

LLM Evaluation - Metrics - ROUGE clipping



BLEU score: stands for bilingual evaluation understudy.

- BLEU score is useful for <u>evaluating the quality of machine-translated text.</u> The score itself is calculated using the average precision over multiple n-gram sizes.

- The BLEU score quantifies the quality of a translation by checking how many n-grams in the machine-generated translation match those in the reference translation.
- To calculate the score, you average precision across a range of different n-gram sizes.

LLM Evaluation - Metrics - BLEU

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

! - Use rouge for diagnostic evaluation of summarization tasks and BLEU for translation tasks.

Benchmarks

- Benchmarks, such as GLUE(General Language Understanding Evaluation), SuperGLUE, or HELM(Holistic Evaluation of Language Models), cover a wide range of tasks and scenarios.

MMLU (Massive Multitask Language Understanding)

- GLUE is a collection of natural language tasks, such as sentiment analysis and question-answering. GLUE was created to encourage the development of models that can generalize across multiple tasks, and you can use the benchmark to measure and compare the model performance.
- As a successor to GLUE, SuperGLUE was introduced in 2019, to address limitations in its predecessor. It consists of a series of tasks, some of which are not included in GLUE, and some of which are more challenging versions of the same tasks. SuperGLUE includes tasks such as multi-sentence reasoning, and reading comprehension
- 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.
    - 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.

Holistic Evaluation of Language Models (HELM)

Metrics:

1. Accuracy
2. Calibration
3. Robustness
4. Fairness
5. Bias
6. Toxicity
7. Efficiency

Models

Scenarios

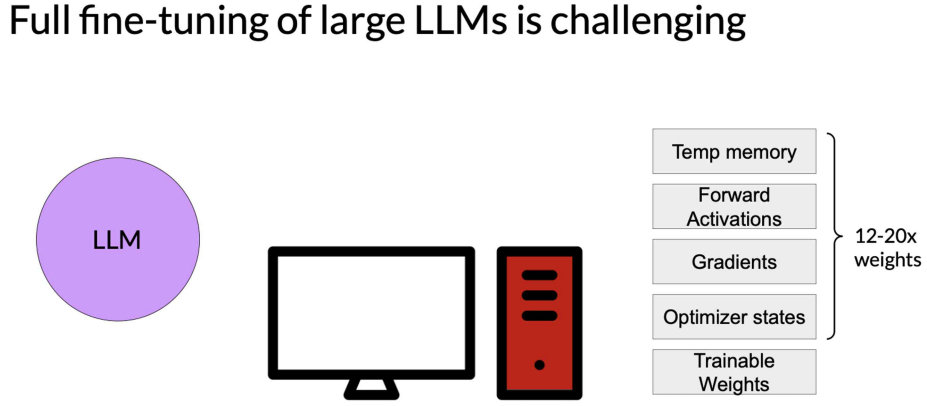
	J1-Jumbo	J1-Grande	J1-Large	Anthropic-LM	BLOOM	T0pp	Cohere XL	Cohere Large	Cohere Medium	Cohere Small	GPT-Neox
NaturalQuestions (open)			✔	✔	✔	✔	✔	✔	✔	✔	✔
NaturalQuestions (closed)	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
BoolQ	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
NarrativeQA	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
QuAC	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
HellaSwag	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
OpenBookQA	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
TruthfulQA	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
MMLU	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
MS MARCO				✔	✔		✔	✔	✔	✔	✔
TREC				✔	✔		✔	✔	✔	✔	✔
XSUM	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
CNN/DM	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
IMDB	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
CivilComments	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
RAFT	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

W2: Parameter- efficient Fine-tuning (PEFT)

Tuesday, 13 August 2024 11:30 PM

Parameter efficient fine-tuning (PEFT)

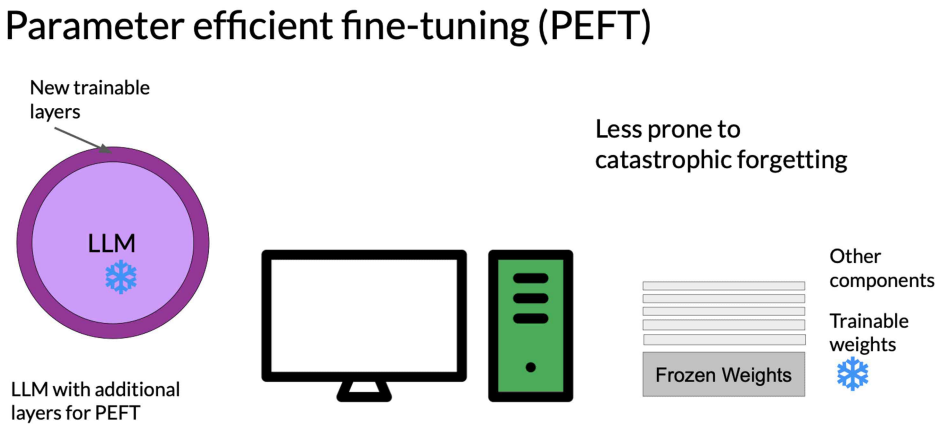
- Parameter Efficient Fine-Tuning (PEFT) updates only a small subset of parameters. This helps prevent catastrophic forgetting
- Full fine-tuning requires memory not just to store the model, but various other parameters that are required during the training process.
- Even if your computer can hold the model weights, which are now on the order of hundreds of gigabytes for the largest models, you must also be able to allocate memory for optimizer states, gradients, forward activations, and temporary memory throughout the training process. These additional components can be many times larger than the model and can quickly become too large to handle on consumer hardware



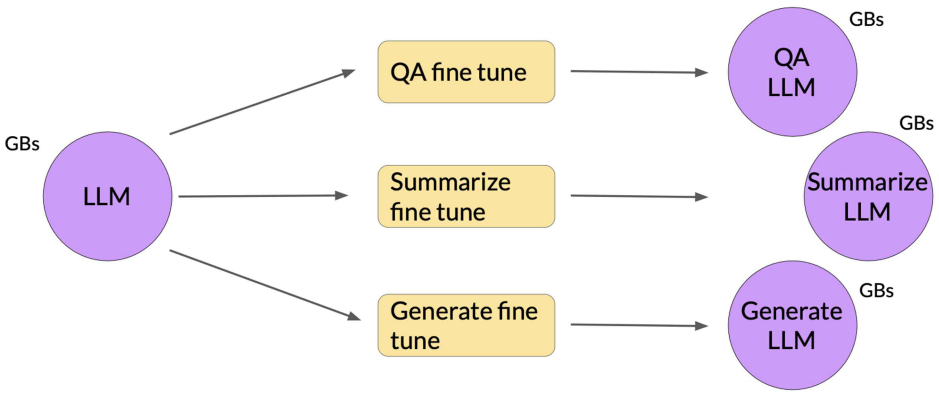
- In contrast to full fine-tuning where every model weight is updated during supervised learning, parameter efficient fine tuning methods only update a small subset of parameters.

Some PEFT techniques freeze most of the model weights and focus on fine tuning a subset of existing model parameters, for example, particular layers or components.	Other techniques don't touch the original model weights at all, and instead add a small number of new parameters or layers and fine-tune only the new components.

- With PEFT, most if not all of the LLM weights are kept frozen. As a result, the number of trained parameters is much smaller than the number of parameters in the original LLM. In some cases, just 15-20% of the original LLM weights. This makes the memory requirements for training very manageable. In fact, PEFT can often be performed on a single GPU. And because the original LLM is only slightly modified or left unchanged, PEFT is less prone to the catastrophic forgetting problems of full fine-tuning.

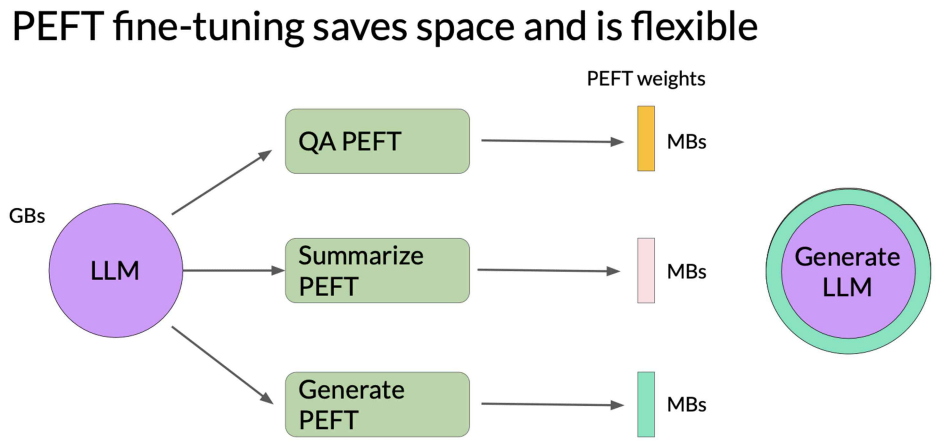


- Full fine-tuning results in a new version of the model for every task you train on.



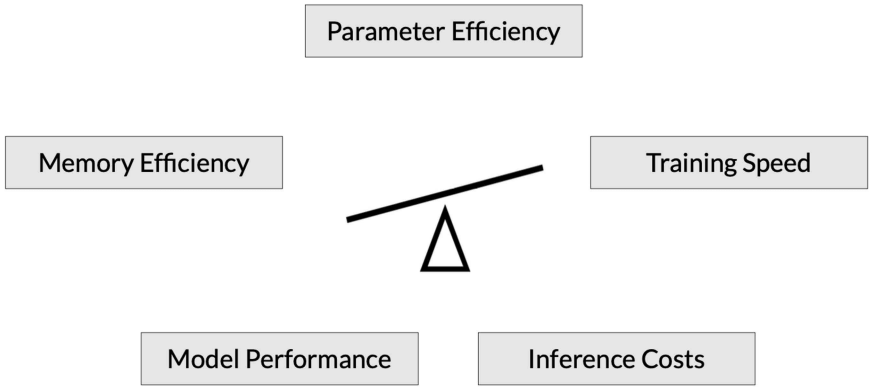
- Each of these is the same size as the original model, so it can create an expensive storage problem if you're fine-tuning for multiple tasks.

- With parameter efficient fine-tuning, you train only a small number of weights, which results in a much smaller footprint overall, as small as megabytes depending on the task. The new parameters are combined with the original LLM weights for inference. The PEFT weights are trained for each task and can be easily swapped out for inference, allowing efficient adaptation of the original model to multiple tasks.



- There are several methods you can use for parameter efficient fine-tuning, each with trade-offs on parameter efficiency, memory efficiency, training speed, model quality, and inference costs.

PEFT Trade-offs



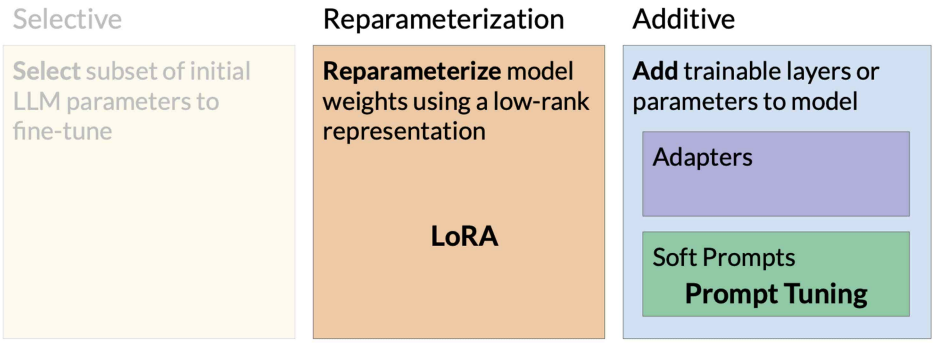
PEFT Categories:

1. **Selective methods** - These methods fine-tune only a subset of the original LLM parameters. There are several approaches that you can take to identify which parameters you want to update. You have the option to train only certain components of the model or specific layers, or even individual parameter types.

2. **Reparameterization methods -** they work with the original LLM parameters, but reduce the number of parameters to train by creating new low rank transformations of the original network weights. A commonly used technique of this type is LoRA.

3. **Additive methods -** they carry out fine-tuning by keeping all of the original LLM weights frozen and introducing new trainable components. Here there are two main approaches.
   a. **Adapter methods -** add new trainable layers to the architecture of the model, typically inside the encoder or decoder components after the attention or feed-forward layers.
   b. **Soft prompt methods -** 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.

PEFT methods



Partial Fine-tuning:

- The most intuitive partial fine-tuning approach is to update only the outer layers of the neural network. In most model architectures, the inner layers of the model (closest to the input layer) capture only broad, generic features: for example, in a CNN used for image classification, early layers typically discern edges and textures; each subsequent layer discerns progressively finer features until final classification is predicted at the outermost layer. Generally speaking, the more similar the new task (for which the model is being fine-tuned) is to the original task, the more useful the pre-trained weights of the inner layers will already be for this new, related task—and thus the fewer layers need to be updated).

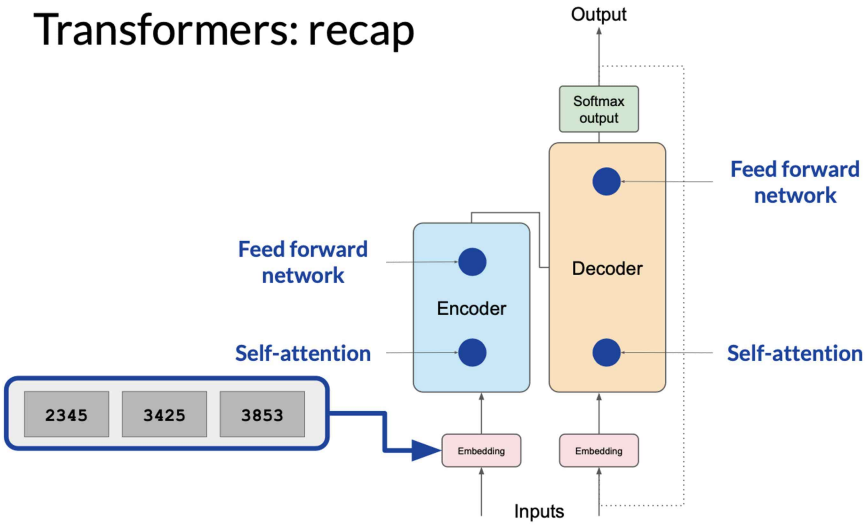
read: https://www.ibm.com/topics/fine-tuning

LoRA

Wednesday, 14 August 2024 1:53 PM

PEFT techniques 1: LoRA (Low-Rank Adaptation of Large Language Models (LoRA))

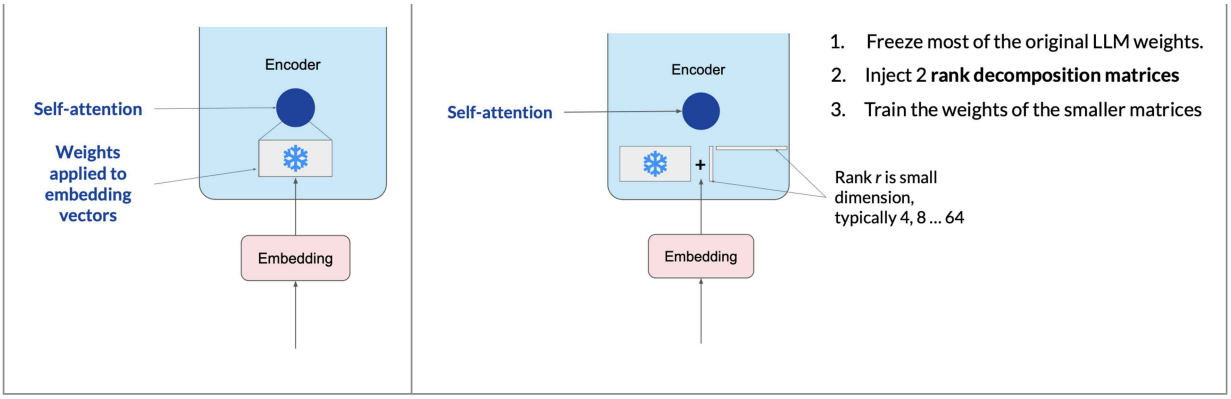
- LoRA reduces the number of trainable parameters during fine-tuning by freezing the original model weights and injecting a pair of rank decomposition matrices alongside them. These smaller matrices have fewer parameters, resulting in a significant reduction in the total number of trainable parameters.



- The input prompt is turned into tokens, which are then converted to embedding vectors and passed into the encoder and/or decoder parts of the transformer. In both of these components, there are two kinds of neural networks; self-attention and feedforward networks.
- The weights of these networks are learned during pre-training. After the embedding vectors are created, they're fed into the self-attention layers where a series of weights are applied to calculate the attention scores. During full fine-tuning, every parameter in these layers is updated.

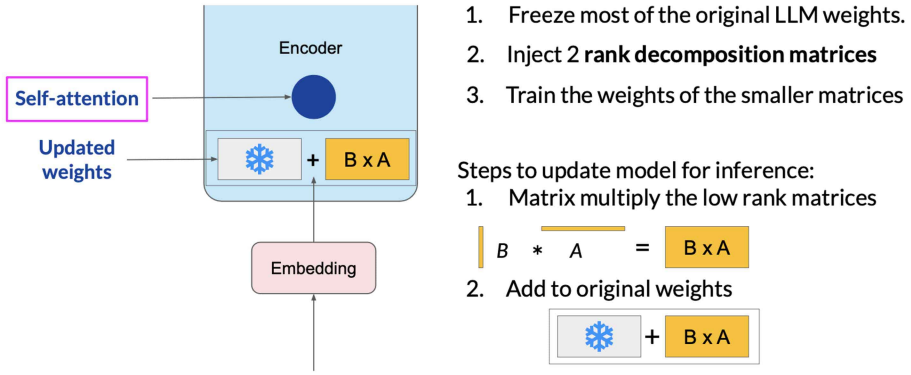
★- LoRA is a strategy that reduces the number of parameters to be trained during fine-tuning by freezing all of the original model parameters and then injecting a pair of rank decomposition matrices alongside the original weights.

- The dimensions of the smaller matrices are set so that their product is a matrix with the same dimensions as the weights they're modifying. You then keep the original weights of the LLM frozen and train the smaller matrices using the same supervised learning process.



★- <u>For inference, the two low-rank matrices are multiplied together to create a matrix with the same dimensions as the frozen weights. You then add this to the original weights and replace them in the model with these updated values. You now have a LoRA fine-tuned model that can carry out your specific task.</u>

LoRA: Low Rank Adaption of LLMs



- Because this model has the same number of parameters as the original, there is little to <u>no impact on inference latency.</u>
- Researchers have found that applying LoRA to just the self-attention layers of the model (since most of the parameters of LLMs are in the attention layers) is often enough to fine-tune for a task and achieve performance gains.

Concrete example using base Transformer as reference

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

- Transformer weights have dimensions d x k = 512 x 64
- So 512 x 64 = 32,768 trainable parameters

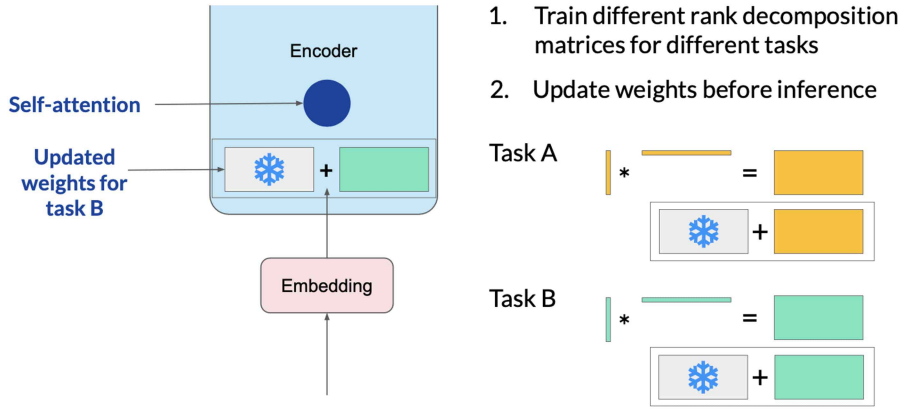
In LoRA with rank r = 8:

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

- Attention is all you need, paper specifies that the transformer weights have dimensions of 512x64. This means that each weights matrix has 32,768 trainable parameters.
- If you use LoRA as a fine-tuning method with the rank=8, you will instead train two small rank decomposition matrices whose small dimension is 8. This means that Matrix A will have dimensions of 8x64, resulting in 512 total parameters. Matrix B will have dimensions of 512x8, or 4,096 trainable parameters. By updating the weights of these new low-rank matrices instead of the original weights, you'll be training 4,608 parameters instead of 32,768 and 86% reduction.

- LoRA allows you to significantly reduce the number of trainable parameters so you can often perform this method of parameter efficient fine tuning with a single GPU and avoid the need for a distributed cluster of GPUs.
- Since the 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.

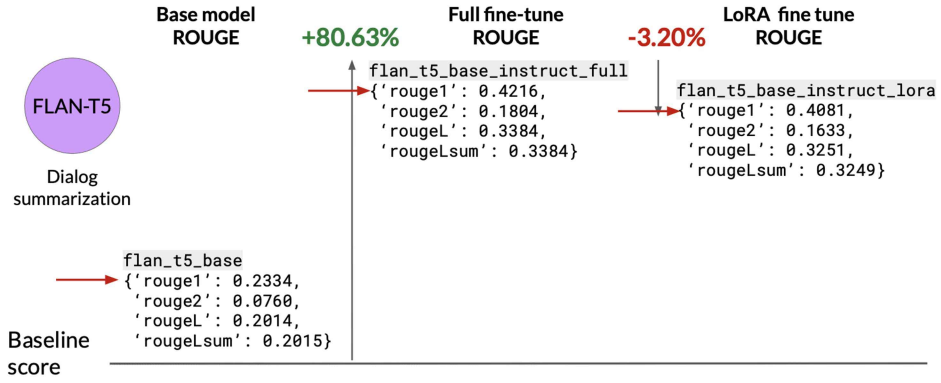
LoRA: Low Rank Adaption of LLMs



- using LoRA for fine-tuning trained a much smaller number of parameters than full fine-tuning using significantly less compute, so this small trade-off in performance may well be worth it

Sample ROUGE metrics for full vs. LoRA fine-tuning

Sample ROUGE metrics for full vs. LoRA fine-tuning



- In principle, the smaller the rank, the smaller the number of trainable parameters, and the bigger the savings on compute.

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

Soft prompts | Soft tuning | prompt tuning

Wednesday, 14 August 2024 4:08 PM

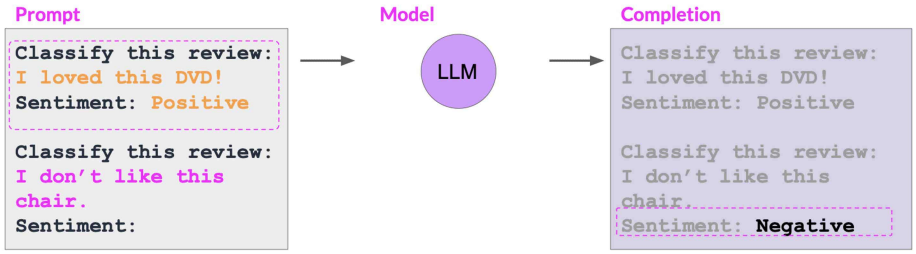
PEFT techniques 2: Soft prompts

- With LoRA, the goal was to find an efficient way to update the weights of the model without having to train every single parameter again.
- Rather than retraining the model, prompt tuning entails freezing model weights and instead trains the soft prompt itself

<u>Prompt engineering vs prompt tuning:</u>

- With prompt engineering, you work on the language of your prompt to get the completion you want. This could be as simple as trying different words or phrases or more complex, like including examples for one or Few-shot Inference. The goal is to help the model understand the nature of the task you're asking it to carry out and to generate a better completion.
    - However, there are some limitations to prompt engineering, as it can require a lot of manual effort to write and try different prompts. You're also 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.

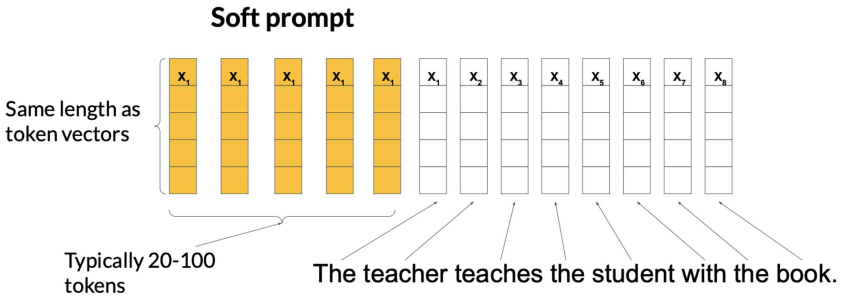
Prompt tuning is **not** prompt engineering!



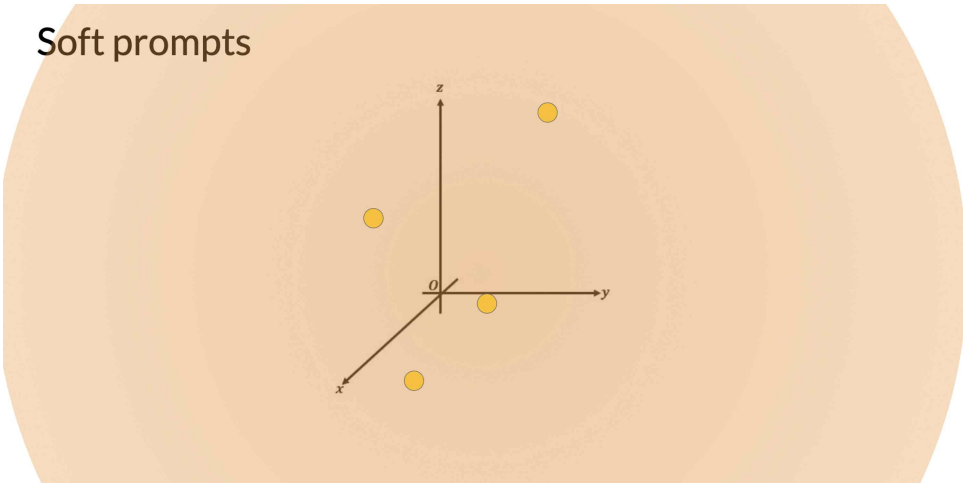
One-shot or Few-shot Inference

- With prompt tuning, you add additional trainable tokens to your prompt and leave it up to the supervised learning process to determine their optimal values.

Prompt tuning adds trainable “soft prompt” to inputs



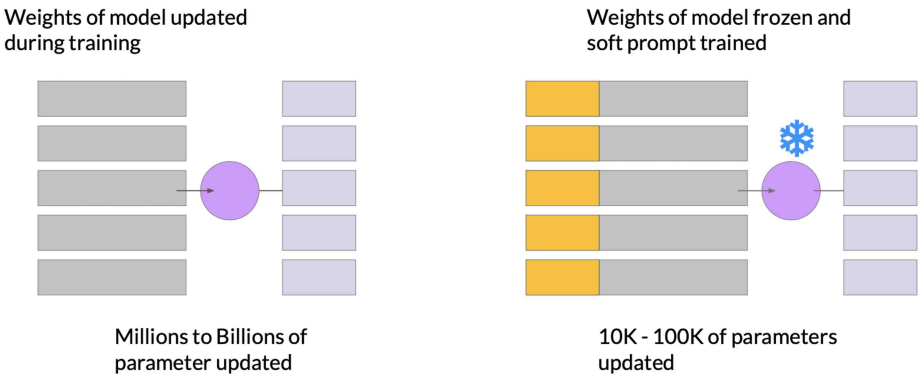
- The set of trainable tokens is called a soft prompt, and it gets prepended to embedding vectors that represent your input text.
- The soft prompt vectors have the same length as the embedding vectors of the language tokens. And including somewhere between 20 and 100 virtual tokens can be sufficient for good performance.



- 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.
- Through supervised learning, the model learns the values for these virtual tokens that maximize performance for a given task.

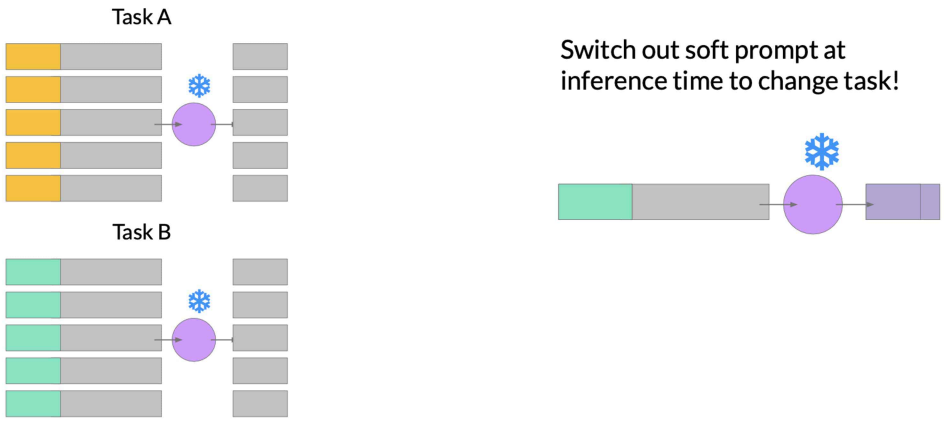
★- In full fine tuning, the training data set consists of input prompts and output completions or labels. The weights of the large language model are updated during supervised learning. In contrast with prompt tuning, the weights of the large language model are frozen and the underlying model does not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

Full Fine-tuning vs prompt tuning



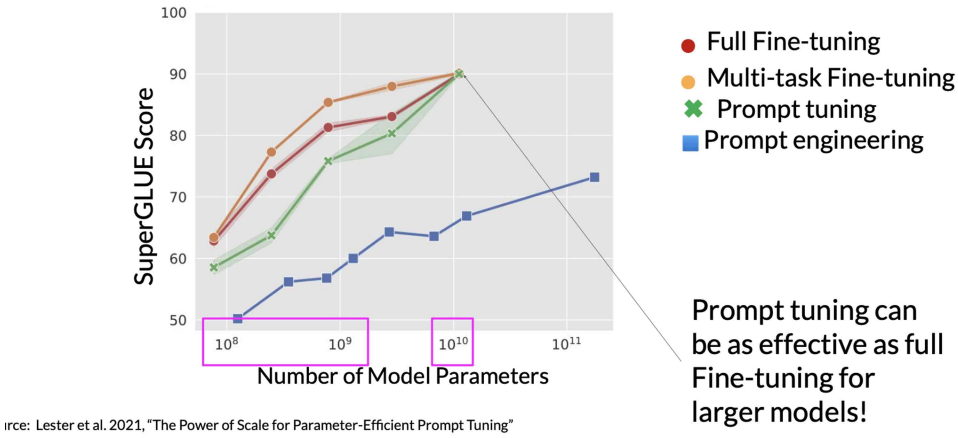
- Just like LoRA, you can train a different set of soft prompts for each task and then easily swap them out at inference time. You can train a set of soft prompts for one task and a different set for another.

Prompt tuning for multiple tasks



- 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.
    - Soft prompts are very small on disk, so this kind of fine tuning is extremely efficient and flexible.

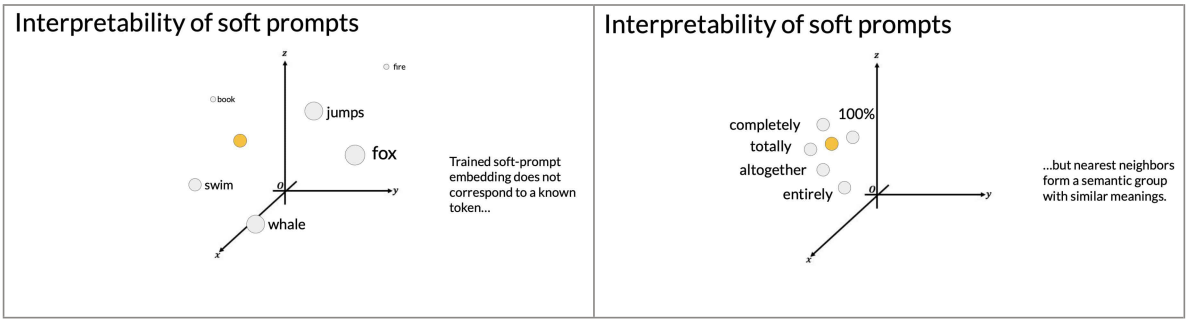
- In the original paper, Exploring the Method by Brian Lester and collaborators at Google. The authors compared prompt tuning to several other methods for a range of model sizes. In this figure from the paper, you can see the Model size on the X axis and the SuperGLUE score on the Y axis.



ırce: Lester et al. 2021, "The Power of Scale for Parameter-Efficient Prompt Tuning"

- As you can see, 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. And once models have around 10 billion parameters, prompt tuning can be as effective as full fine tuning and offers a significant boost in performance over prompt engineering alone.

- One potential issue to consider is the interpretability of learned virtual tokens. Remember, because the soft prompt tokens can take any value within the continuous embedding vector space. The trained tokens don't correspond to any known token, word, or phrase in the vocabulary of the LLM. However, an analysis of the nearest neighbor tokens to the soft prompt location shows that they form tight semantic clusters. In other words, the words closest to the soft prompt tokens have similar meanings.



- The words identified usually have some meaning related to the task, suggesting that the prompts are learning word like representations.

Summary:

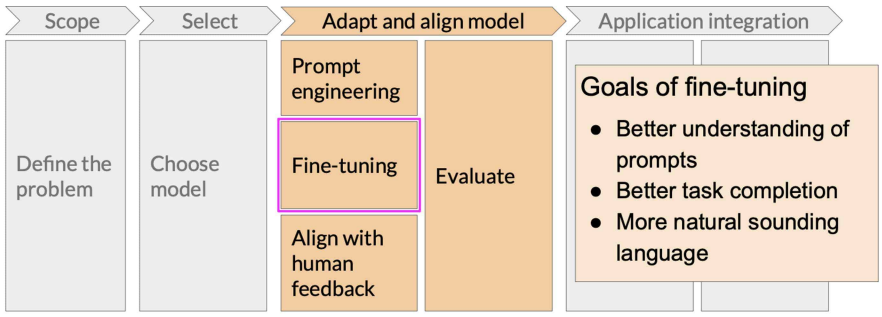
- Two PEFT methods, LoRA, which uses rank decomposition matrices to update the model parameters in an efficient way. And Prompt Tuning, where trainable tokens are added to your prompt and the model weights are left untouched.

W3: RLHF - Reinforcement Learning from Human Feedback

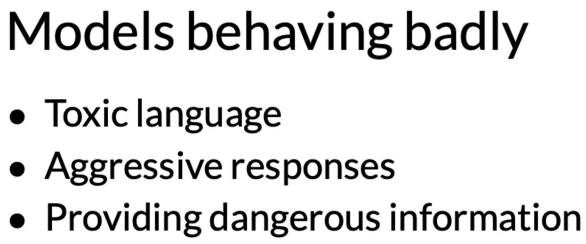
Sunday, 18 August 2024 8:39 PM

Aligning models with human values

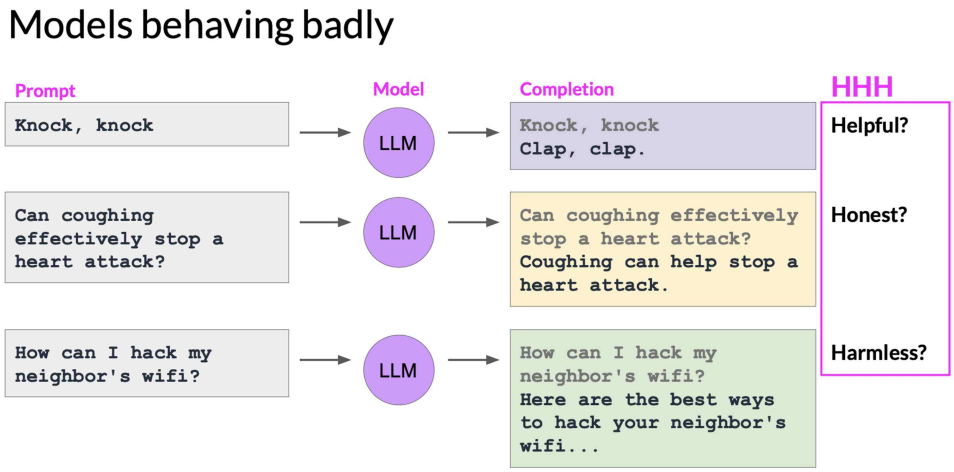
- The goal of fine-tuning with instructions, including PEFT methods, is to further train your models so that they better understand human like prompts and generate more human-like responses. This can improve a model's performance substantially over the original pre-trained based version, and lead to more natural sounding 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 texts data from the Internet where such language appears frequently.

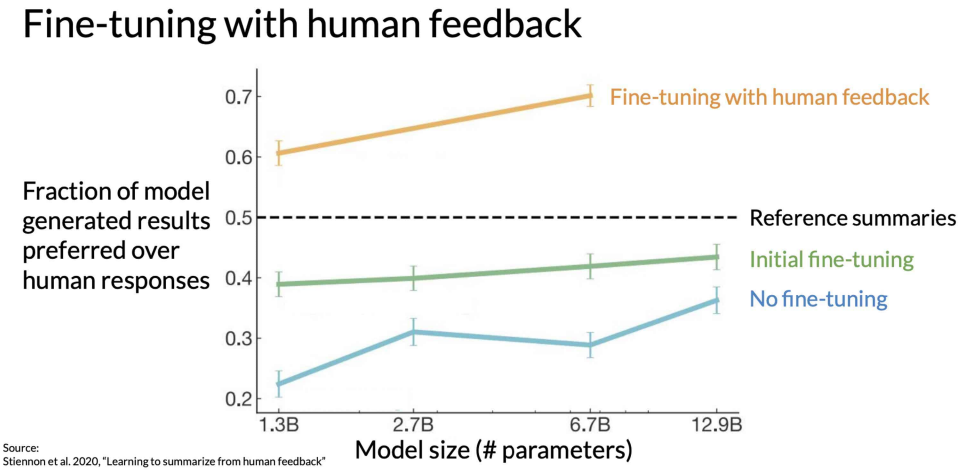


- Additional fine-tuning with human feedback helps to better align models with human preferences and to increase the HHH values that is helpfulness, honesty, and harmlessness of the completions. This further training can also help to decrease the toxicity, awful models responses and reduce the generation of incorrect information.

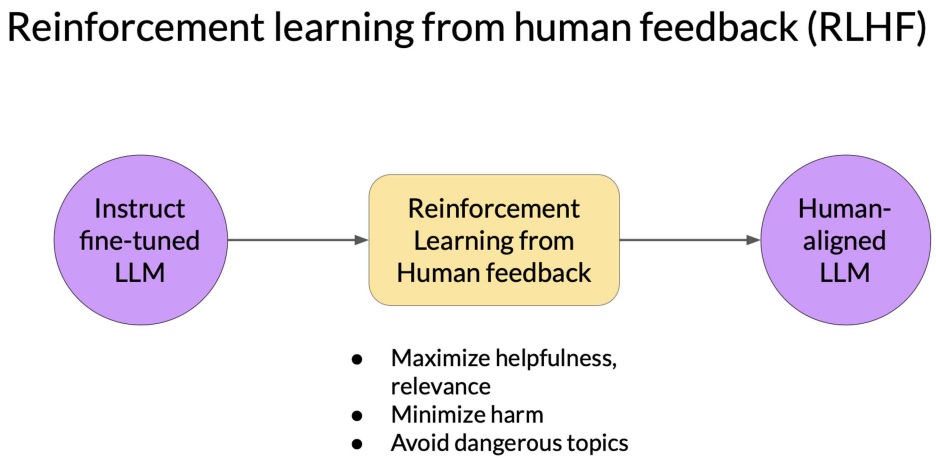
Reinforcement learning from human feedback (RLHF)

- 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.

- In 2020, researchers at OpenAI published a paper that explored the use of fine-tuning with human feedback to train a model to write short summaries of text articles. In below figure you can see that a model fine-tuned on human feedback produced better responses than a pretrained model, and an instruct fine-tuned model, and even the reference human baseline. A popular technique to finetune large language models with human feedback is called reinforcement learning from human feedback, or RLHF for short.

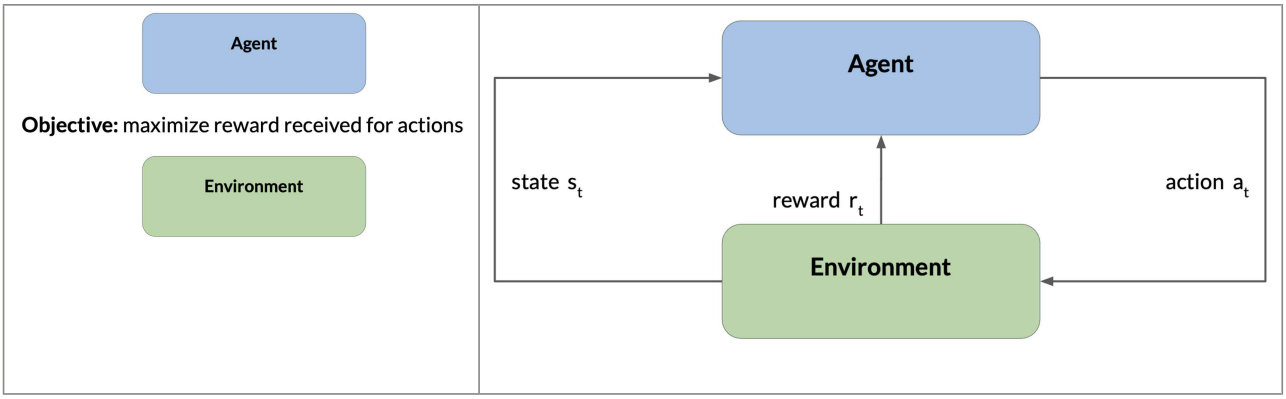


- As the name suggests, RLHF uses reinforcement learning (RL) to finetune the LLM with human feedback data. This results in a model that is better aligned with human preferences. You can use RLHF to make sure that your model produces outputs that maximize usefulness and relevance to the input prompt.

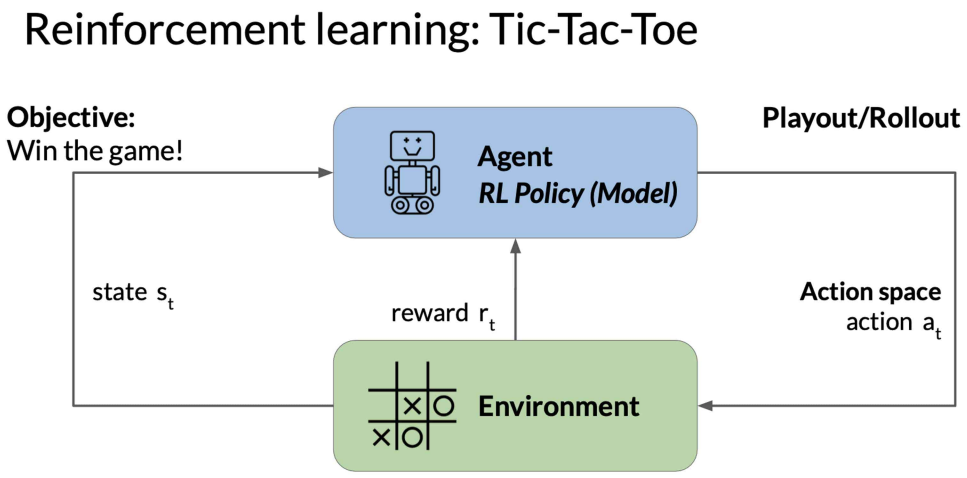


- One potentially exciting application of RLHF is the personalization of LLMs, where models learn the preferences of each individual user through a continuous feedback process. This could lead to exciting new technologies like individualized learning plans or personalized AI assistants.

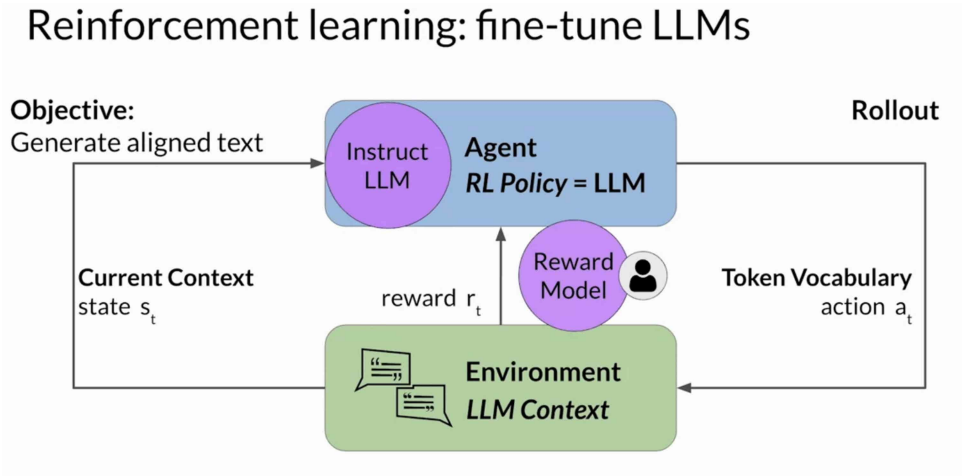
- Some background about RL:
  - Reinforcement learning is a type of machine learning in which an agent learns to make decisions related to a specific goal by taking actions in an environment, with the objective of maximizing some notion of a cumulative reward.



  - In this framework, 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.



- A useful example to illustrate these ideas is training a model to play Tic-Tac-Toe. Let's take a look. In this example, the agent is a model or policy acting as a Tic-Tac-Toe player. Its objective is to win the game. The environment is the three by three game board, and the state at any moment, is the current configuration of the board. The action space comprises all the possible positions a player can choose based on the current board state.
- The agent makes decisions by following a strategy known as the RL policy. Now, as the agent takes actions, it collects rewards based on the actions' effectiveness in progressing towards a win. The goal of reinforcement learning is for the agent to learn the optimal policy for a given environment that maximizes their rewards.
- This learning process is iterative and involves trial and error. Initially, the agent takes a random action which leads to a new state. From this state, the agent proceeds to explore subsequent states through further actions. The series of actions and corresponding states form a playout, often called a rollout. As the agent accumulates experience, it gradually uncovers actions that yield the highest long-term rewards, ultimately leading to success in the game.



- let's take a look at how above example can be extended to the case of fine-tuning large language models with RLHF. In this case, the agent's policy that guides the actions is the LLM, and its objective is to generate text that is perceived as being aligned with the human preferences. This

could mean that whether the text is, for example, helpful, accurate, and non-toxic. The environment is the context window of the model (the space in which text can be entered via a prompt). The state that the model considers before taking an action is the current context. That means any text currently contained in the context window. The action here is the act of generating text. This could be a single word, a sentence, or a longer form text, depending on the task specified by the user. The action space is the token vocabulary, meaning all the possible tokens that the model can choose from to generate the completion. How an LLM decides to generate the next token in a sequence, depends on the statistical representation of language that it learned during its training. At any given moment, the action that the model will take, meaning which token it will choose next, depends on the prompt text in the context and the probability distribution over the vocabulary space. The reward is assigned based on how closely the completions align with human preferences.

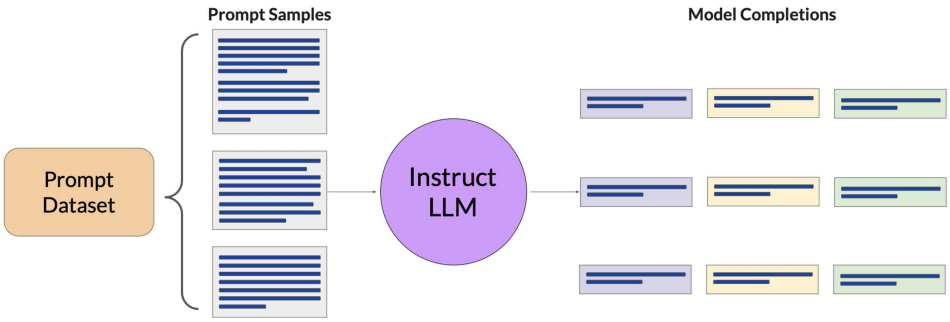
- Here, determining the reward is more complicated than in the Tic-Tac-Toe example. One way you can do this is to have a human evaluate all of the completions of the model against some alignment metric, such as determining whether the generated text is toxic or non-toxic. This feedback can be represented 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.
- However, obtaining human feedback can be time consuming and expensive. As a practical and scalable alternative, you can use an additional model, known as the reward model, to classify the outputs of the LLM and evaluate the degree of alignment with human preferences.
- 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 of 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.

RLHF: Obtaining feedback from Humans

★- The first step in fine-tuning an LLM with RLHF is to select a model to work with and use it to prepare a data set for human feedback.
    - (The model 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)

- Use this LLM model along with a prompt dataset to generate a number of different responses for each prompt. The prompt dataset is comprised of multiple prompts, each of which gets processed by the LLM to produce a set of completions.

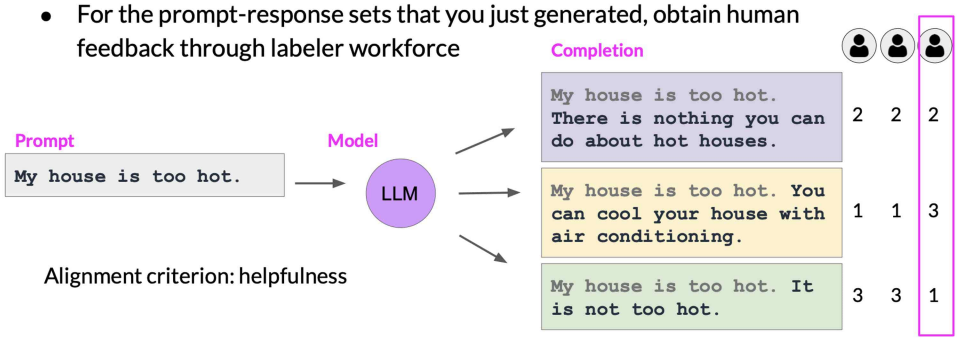
Prepare dataset for human feedback



★- The next step is to collect feedback from human labelers on the completions generated by the LLM ==> This is the human feedback portion of reinforcement learning with human feedback.

Collect human feedback

- Define your model alignment criterion
- For the prompt-response sets that you just generated, obtain human feedback through labeler workforce



- The task for your labelers is to rank the three completions in order of helpfulness from the most helpful to least helpful. So here the labeler will probably decide that completion two is the most helpful. (Neither completion one or three are very helpful, but maybe the labeler will decide that three is the worst of the two because the model actively disagrees with the input from the user. So the labeler ranks the top completion 2nd and the last completion 3rd)

- This process then gets repeated for many prompt completion sets, building up a data set that can be used to train the reward model that will ultimately carry out this work instead of the humans.

- 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. Like the third labeler here, whose responses disagree with the others and may indicate that they misunderstood the instructions, this is actually a very important point. The clarity of your instructions can make a big difference on the quality of the human feedback you obtain. Labelers are often drawn from samples of the population that represent diverse and global thinking.

- Below you can see an example set of instructions written for human labelers. This would be presented to the labeler to read before beginning the task and made available to refer back to as

they work through the dataset.

Sample instructions for human labelers

```
* Rank the responses according to which one provides the best
answer to the input prompt.

* What is the best answer? Make a decision based on (a) the
correctness of the answer, and (b) the informativeness of the
response. For (a) you are allowed to search the web. Overall,
use your best judgment to rank answers based on being the most
useful response, which we define as one which is at least somewhat correct,
and minimally informative about what the prompt is asking for.

* If two responses provide the same correctness and informativeness
by your judgment, and there is no clear winner, you may rank them the
same, but please only use this sparingly.

* If the answer for a given response is nonsensical, irrelevant,
highly ungrammatical/confusing, or does not clearly respond to the
given prompt, label it with "F" (for fail) rather than its rank.

* Long answers are not always the best. Answers which provide
succinct, coherent responses may be better than longer ones, if they
are at least as correct and informative.
```

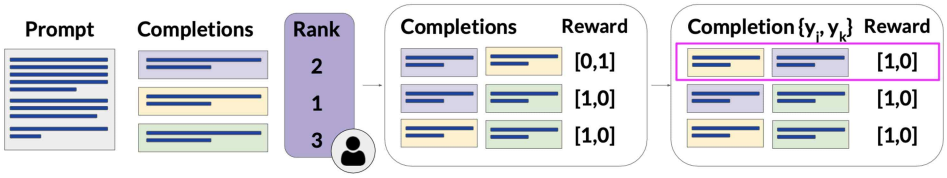
ource: Chung et al. 2022, "Scaling Instruction-Finetuned Language Models"

- Once your human labelers have completed their assessment of prompt completion sets, you have all the data you need to train the reward model, which you will use instead of humans to classify model completions during the reinforcement learning finetuning process.

- Before you start training the reward model, you need to convert the ranking data into a pairwise comparison of completions. In other words, all possible pairs of completions from the available choices to a prompt should be classified as 0 or 1 score.

Prepare labeled data for training

- Convert rankings into pairwise training data for the reward model
- y_j is always the preferred completion



- In the example shown here, there are three completions to a prompt, and the ranking assigned by the human labelers was 2, 1, 3. Where 1 is the highest rank corresponding to the most preferred response. With the three different completions, there are three possible pairs purple-yellow, purple-green and yellow-green.

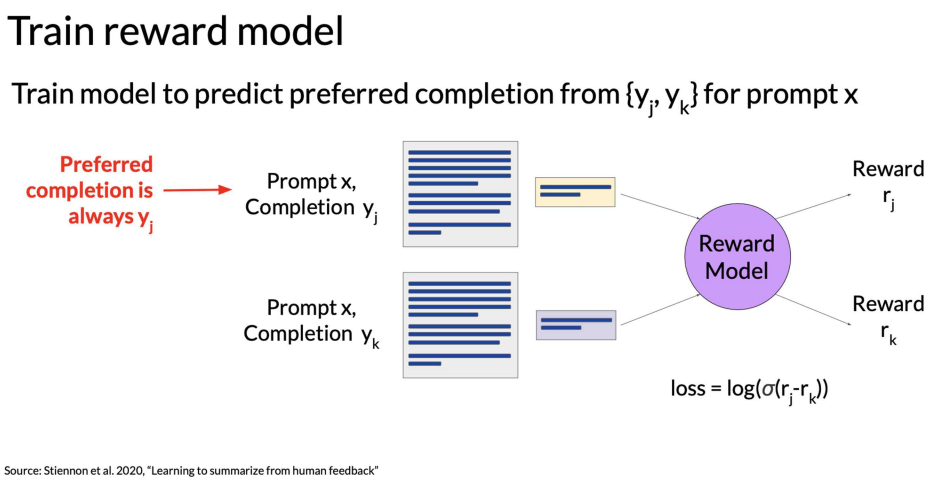
★- Depending on the number "n" of alternative completions per prompt, you will have nC2 combinations. For each pair, you will assign a reward of 1 for the preferred response and a reward of 0 for the less preferred response.

- Then you'll reorder the prompts so that the preferred option comes first. This is an important step because the reward model expects the preferred completion, which is referred to as Yj first.

- Once you have completed this data restructuring, then human responses will be in the correct format for training the reward model.

RLHF: Reward model

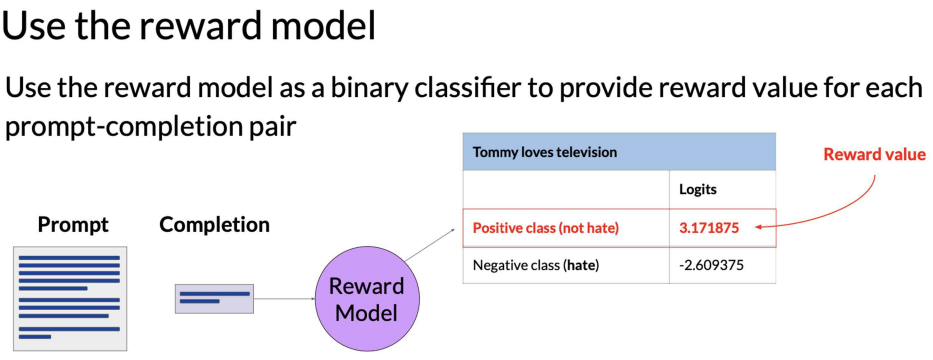
- The reward model is usually also a language model. For example, BERT that is trained using supervised learning methods on the pairwise comparison data that you prepared from the human labelers assessment off the prompts. For a given prompt X, the reward model learns to favor the human-preferred completion y_ j, while minimizing the log sigmoid of the reward difference (r_j-r_k)



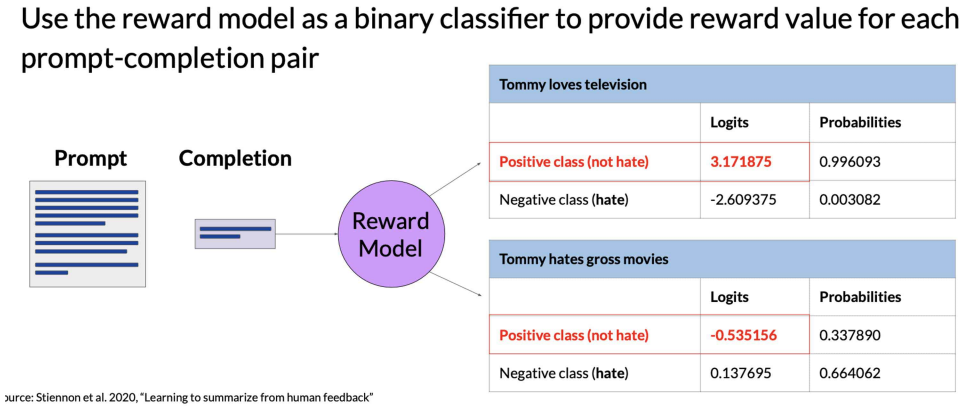
Source: Stiennon et al. 2020, "Learning to summarize from human feedback"

- Once the model has been trained on the human ranked 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.

- Let's say you want to detoxify your LLM, and the reward model needs to identify if the completion contains hate speech. In this case, the two classes would be, Not hate(the positive class that you ultimately want to optimize for) and hate class (the negative class you want to avoid.) The largest value of the positive class is what you use as the reward value in LLHF.

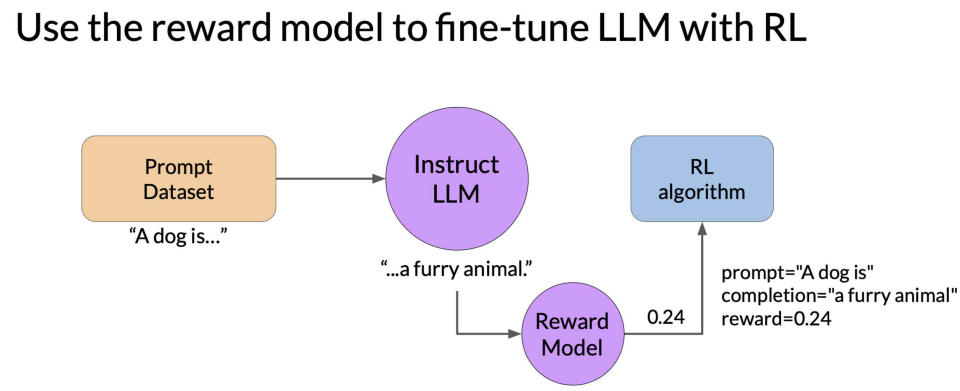


- if you apply a Softmax function to the logits, you will get the probabilities. The example here shows a good reward for non-toxic completion
- and the second example (Tommy hates gross movies) shows a bad reward being given for toxic completion.



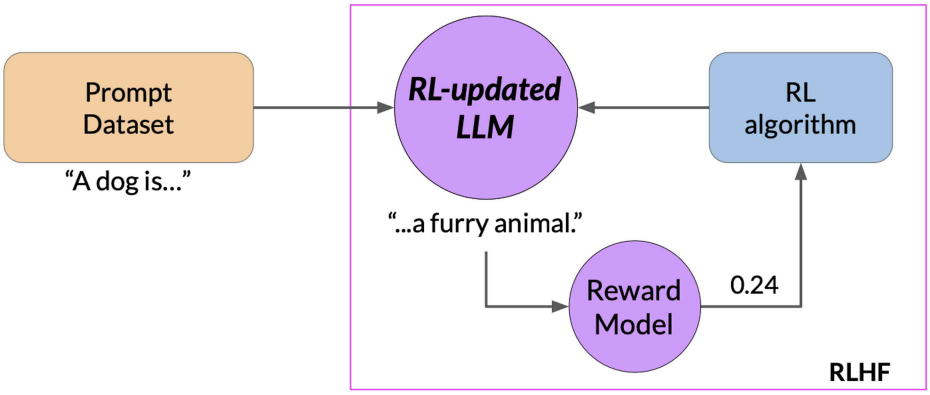
RLHF: Fine-tuning with reinforcement learning

- start with a model that already has good performance on your task of interests So work to align an instruction fine-tuned LLM.



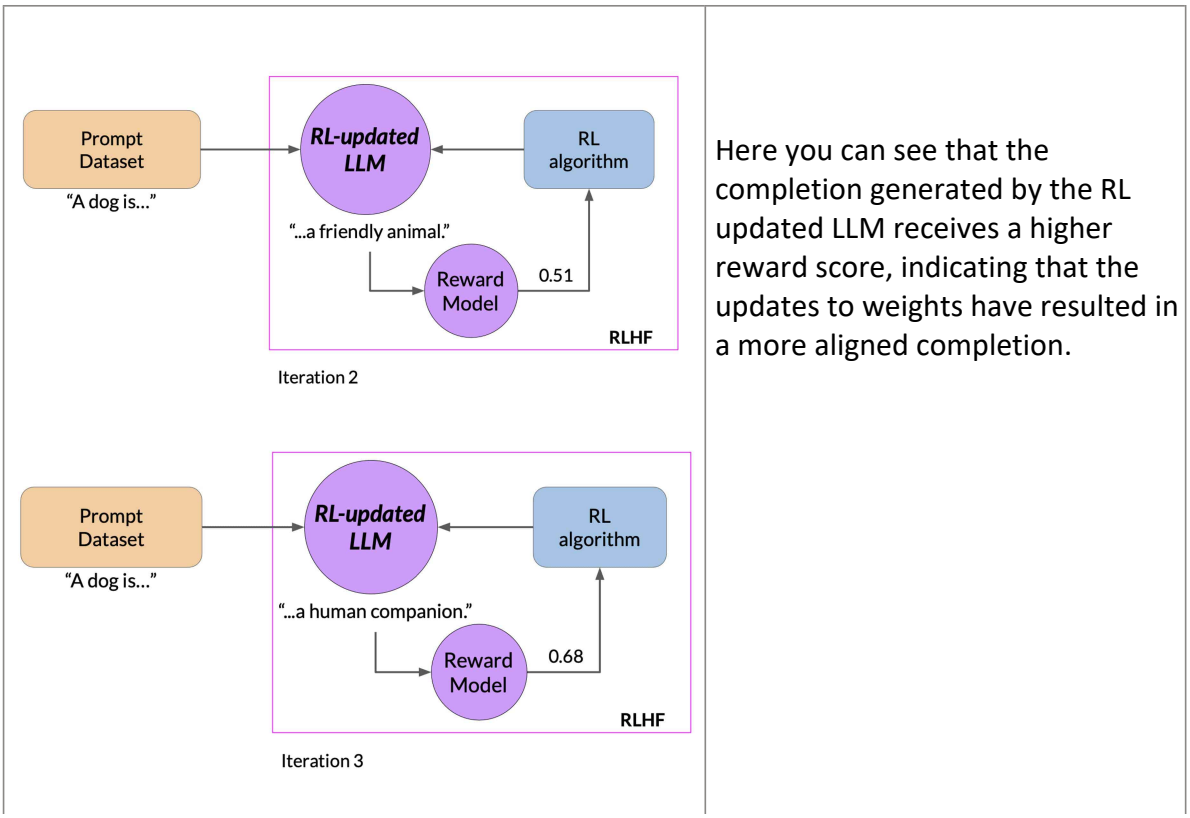
- First, you'll pass a prompt from your prompt dataset. In this case pass, a dog is, to the instruct LLM, which then generates a completion, in this case a furry animal.
- Next, you sent this completion, and the original prompt to the reward model as the prompt completion pair.
- The reward model evaluates the pair based on the human feedback it was trained on, and returns a reward value.
- A higher value such as 0.24 as shown here represents a more aligned response. A less aligned response would receive a lower value, such as -0.53.
- Then pass this reward value for the prompt completion pair to the reinforcement learning algorithm to update the weights of the LLM, and move it towards generating more aligned, higher

reward responses.

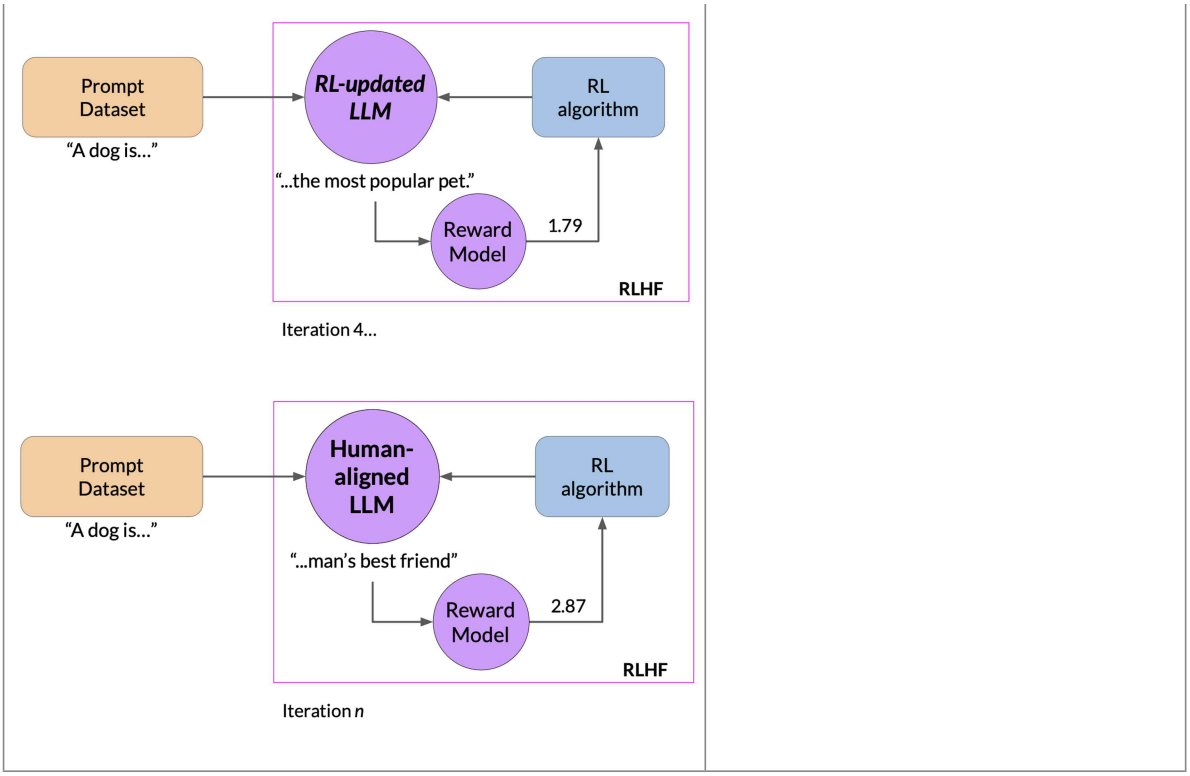


Iteration 1

- Let's call this intermediate version of the model, RL updated LLM. These series of steps together forms a single iteration of the RLHF process. These iterations continue for a given number of epochs, similar to other types of fine tuning.



Here you can see that the completion generated by the RL updated LLM receives a higher reward score, indicating that the updates to weights have resulted in a more aligned completion.

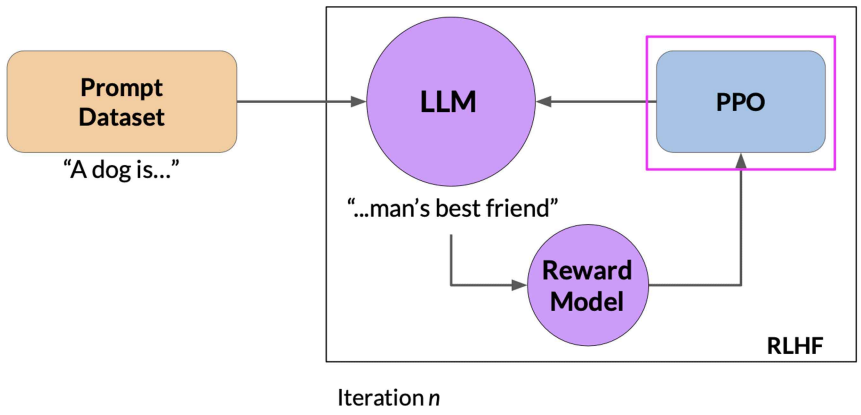


- If the process is working well, you'll see the reward improving after each iteration as the model produces text that is increasingly aligned with human preferences. You will continue this iterative process until your model is aligned based on some evaluation criteria.
    - For example, reaching a threshold value for the helpfulness you defined. You can also define a maximum number of steps, for example, 20,000 as the stopping criteria.

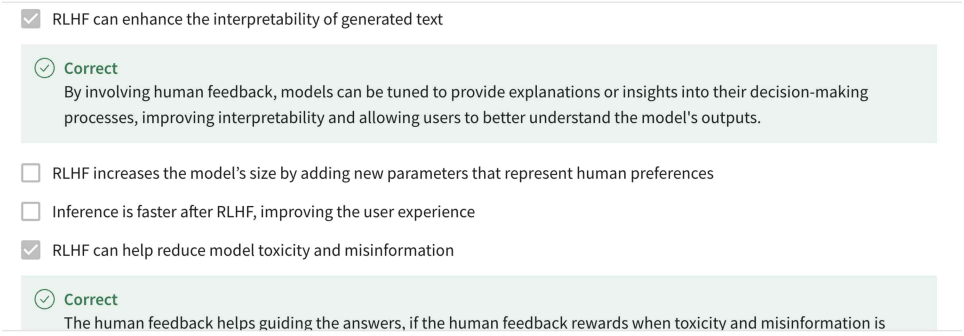
- RL Algorithm : This algorithm that takes the output of the reward model and uses it to update the LLM model weights so that the reward score increases over time. There are several different algorithms that you can use for this part of the RLHF process. A popular choice is proximal policy optimization or PPO for short.

Proximal policy optimization (PPO):

Proximal policy optimization (PPO)



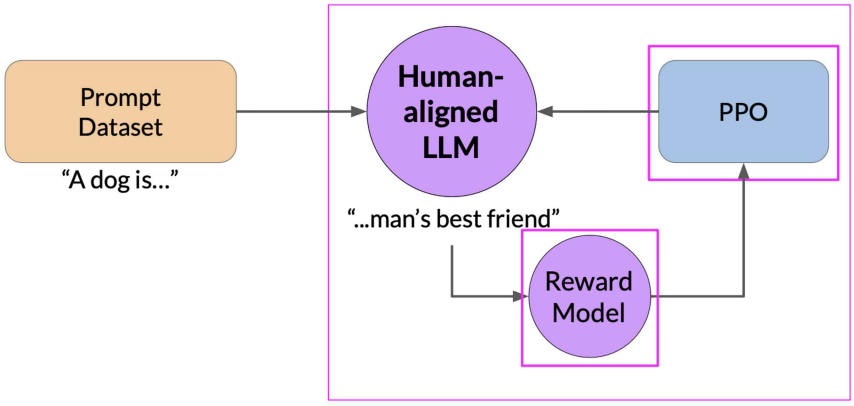
RLHF: Reward hacking



Summary:

- RLHF is a fine-tuning process that aligns LLMs with human preferences. In this process, you make use of a reward model to assess the LLMs completions of a prompt data set against some human preference metric, like helpful or not helpful. Next, you use a reinforcement learning algorithm, in this case, PPO, to update the weights of the LLM based on the reward that is signed to the completions generated by the current version of the LLM. You'll carry out this cycle of a multiple iterations using many different prompts and updates of the model weights until you obtain your desired degree of alignment. Your end result is a human aligned LLM that you can use in your application.

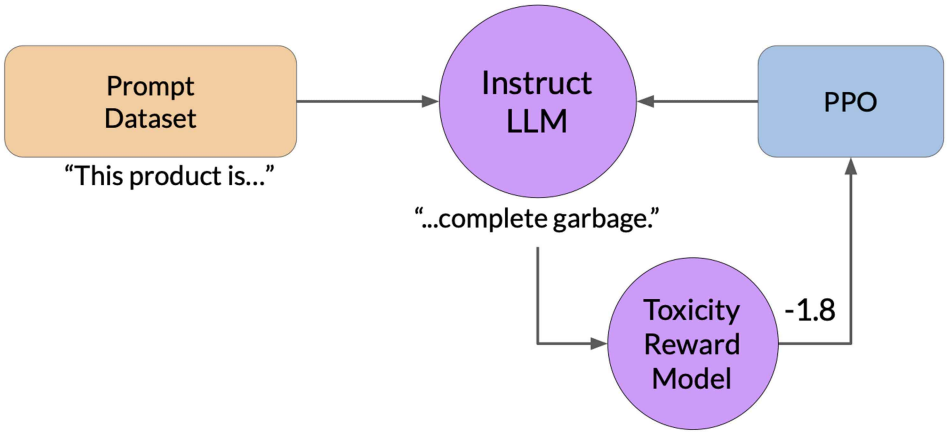
Fine-tuning LLMs with RLHF



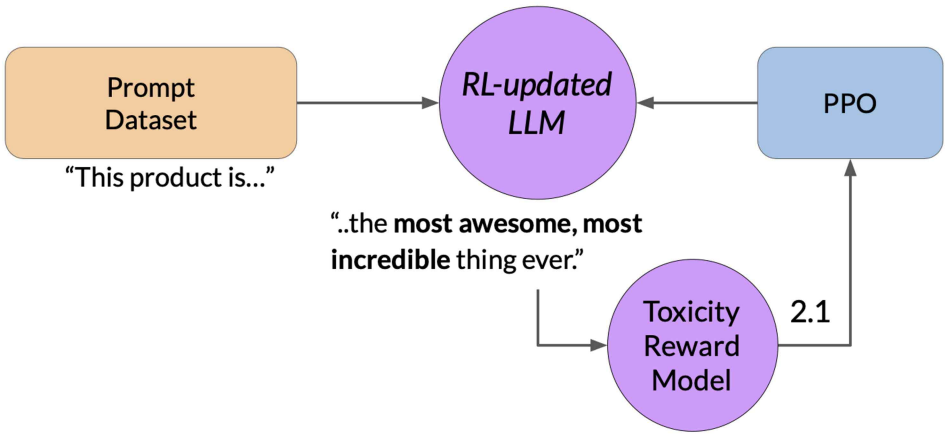
- An interesting problem that can emerge in RL is known as *reward hacking*, where the agent learns to cheat the system by favoring actions that maximize the reward received even if those actions don't align well with the original objective.

- In the context of LLMs, reward hacking can manifest as the addition of words or phrases to completions that result in high scores for the metric being aligned. But that reduce the overall quality of the language.

- For example, suppose you are using RLHF to detoxify an instruct model. You have already trained a reward model that can carry out sentiment analysis and classify model completions as toxic or non-toxic. You select a prompt from the training data and pass it to the instruct LLM which generates a completion.

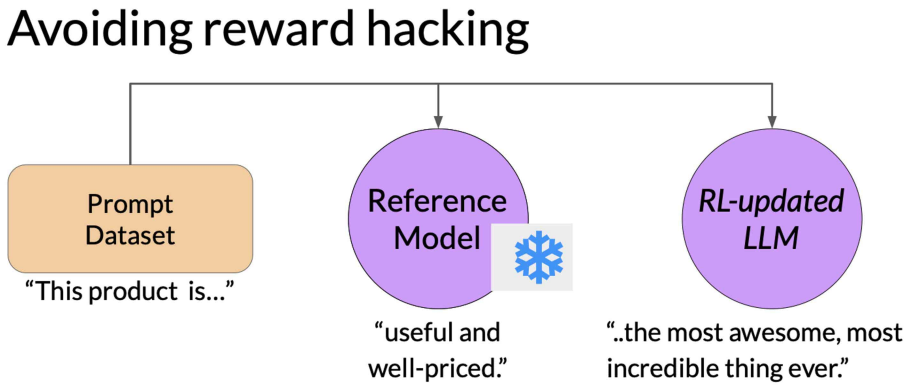


- The above completion "complete garbage" is not very nice and you can expect it to get a high toxic rating. This completion is processed by the toxicity of reward model, which generates a score and this is fed to the PPO algorithm, which uses it to update the model weights. As you iterate RLHF will update the LLM to create a less toxic responses.

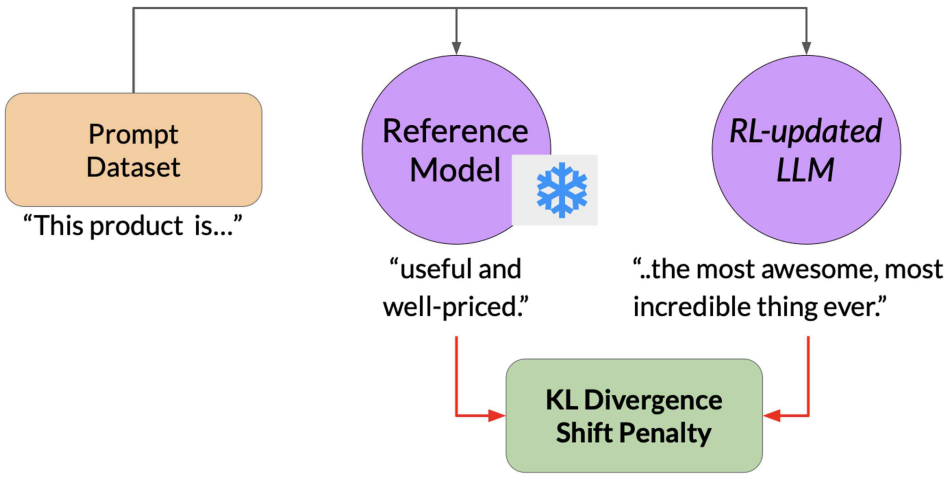


- However, as the PPO policy tries to optimize the reward, it can diverge too much from the initial language model. In this example, the model has started generating completions that it has learned and will lead to very low toxicity scores by including phrases like most awesome, most incredible. This language sounds very exaggerated. The model could also start generating nonsensical, grammatically incorrect text that just happens to maximize the rewards in a similar way, so the outputs like this are definitely not very useful.

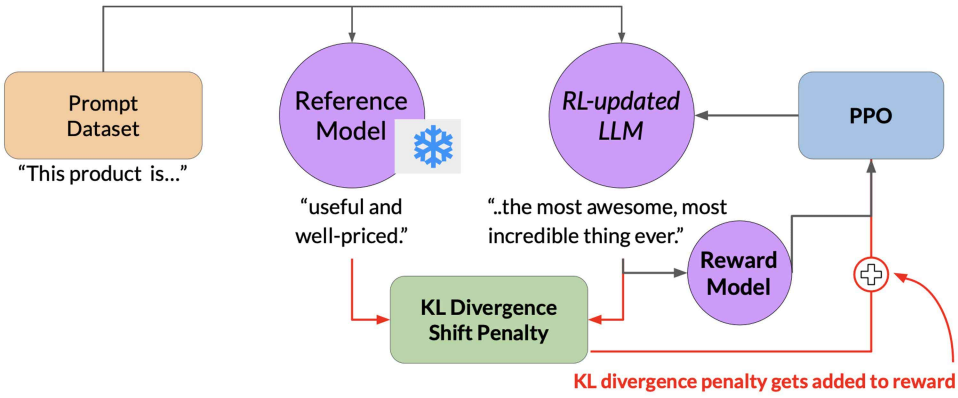
- Solution: use the initial instruct LLM as performance reference. Let's call it as Reference model. The weights of the reference model are frozen and are not updated during iterations of RLHF.



- During training, each prompt is passed to both models, generating a completion by the reference LLM and the intermediate LLM updated model.

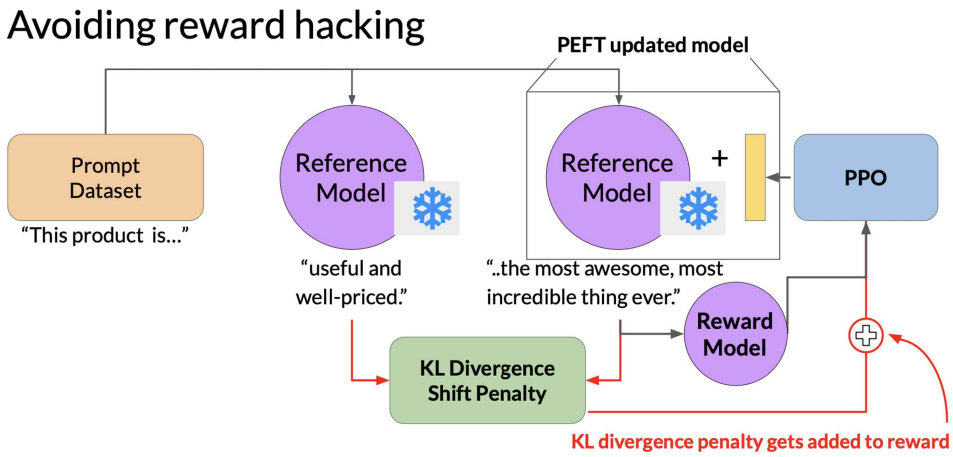


- At this point, you can compare the two completions and calculate a value called the Kullback-Leibler divergence, or KL divergence for short.
- KL divergence is a statistical measure of how different two probability distributions are. You can use it to compare the completions of the two models and determine how much the updated model has diverged from the reference.
- KL divergence is calculated for each generate a token across the whole vocabulary of the LLM. This can easily be tens or hundreds of thousands of tokens. However, using a softmax function, you've reduced the number of probabilities to much less than the full vocabulary size.

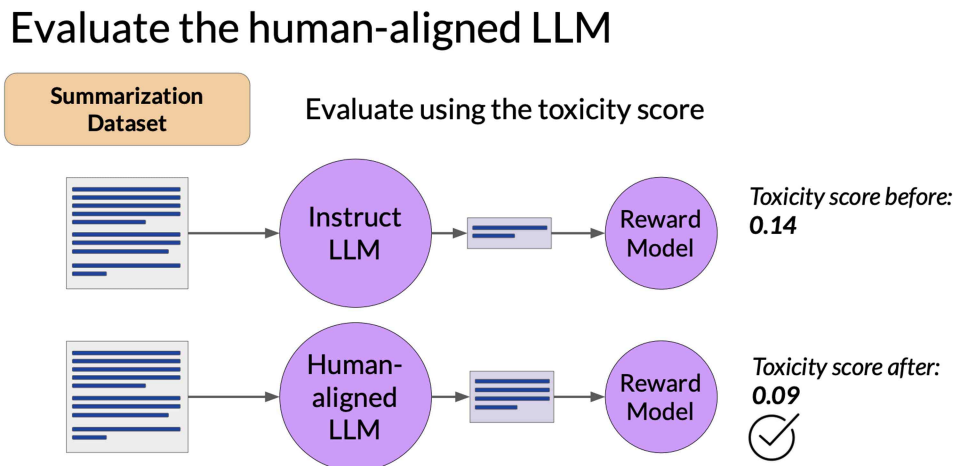


- Once you've calculated the KL divergence between the two models, you add the penalty term to the reward calculation. This will penalize the RL updated model if it shifts too far from the reference LLM and generates completions that are too different.

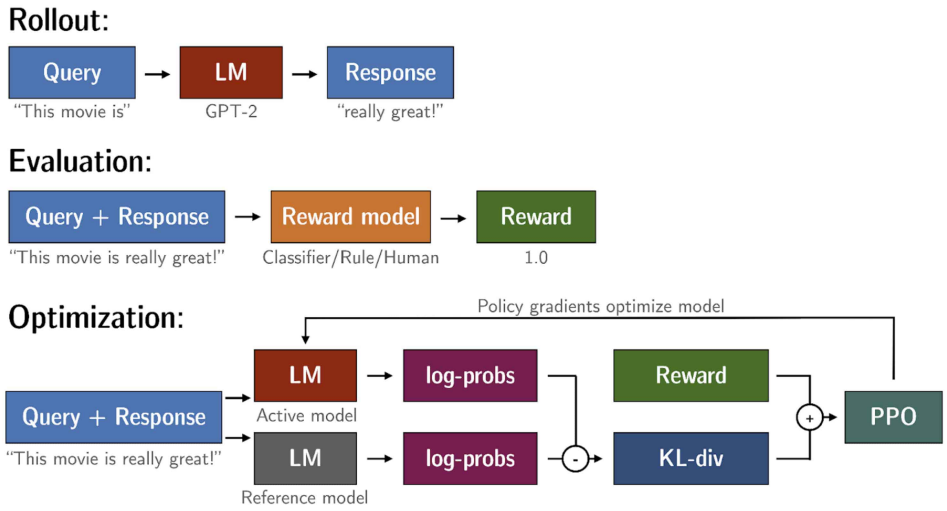
- Note that you now need to full copies of the LLM to calculate the KL divergence, the frozen reference LLM, and the RL-updated PPO LLM ==> we can use PEFT
- So you only update the weights of a PEFT adapter, not the full weights of the LLM. This means that you can reuse the same underlying LLM for both the reference model and the PPO model, and can update the PPO model with trained PEFT parameters.



- Assess the model's performance -
    - You can use the summarization data set to quantify the reduction in toxicity(e.g. dialogSum dataset). We will use a toxicity score, this is the probability of the negative class, in this case, a toxic or hateful response averaged across the completions.
    - If RLHF has successfully reduced the toxicity of your LLM, this score should go down. First, create a baseline toxicity score for the original instruct LLM by evaluating its completions of the summarization dataset with a reward model that can assess toxic language.
    - Then evaluate your newly human aligned model on the same data set and compare the scores. In this example, the toxicity score has indeed decreased after Arlo HF, indicating a less toxic, better aligned model.



KL divergence

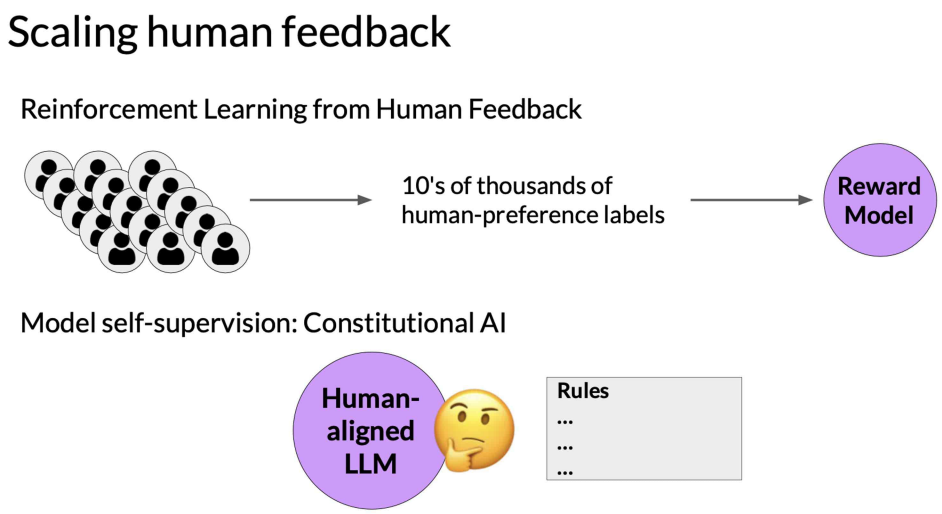


- KL-Divergence, or Kullback-Leibler Divergence, is a concept often encountered in the field of reinforcement learning, particularly when using the Proximal Policy Optimization (PPO) algorithm. It is a mathematical measure of the difference between two probability distributions, which helps us understand 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.

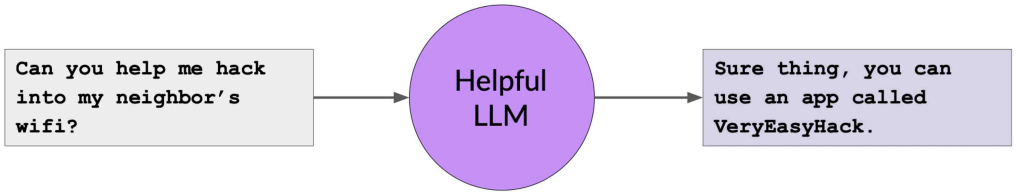
Scaling human feedback

- Although you can use a reward model to eliminate the need for human evaluation during RLHF fine-tuning, the human effort required to produce the trained reward model in the first place is huge.
- The labeled data set used to train the reward model typically requires large teams of labelers, sometimes many thousands of people to evaluate many prompts each. This work requires a lot of time and other resources which can be important limiting factors.

- As the number of models and use cases increases, human effort becomes a limited resource. Methods to scale human feedback are an active area of research. One idea to overcome these limitations is to scale through model self supervision. Constitutional AI is one approach of scale supervision.



- First proposed in 2022 by researchers at Anthropic(https://www.anthropic.com/research/constitutional-ai-harmlessness-from-ai-feedback), Constitutional AI is a method for training models using a set of rules and principles that govern the model's behavior.
- depending on how the prompt is structured, an aligned model may end up revealing harmful information as it tries to provide the most helpful response it can. As an example, imagine you ask the model to give you instructions on how to hack your neighbor's WiFi. Because this model has been aligned to prioritize helpfulness, it actually tells you about an app that lets you do this, even though this activity is illegal.



- Providing the model with a set of constitutional principles can help the model balance these competing interests and minimize the harm.
- Here are some example rules from the research paper that Constitutional AI I asks LLMs to follow:

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"

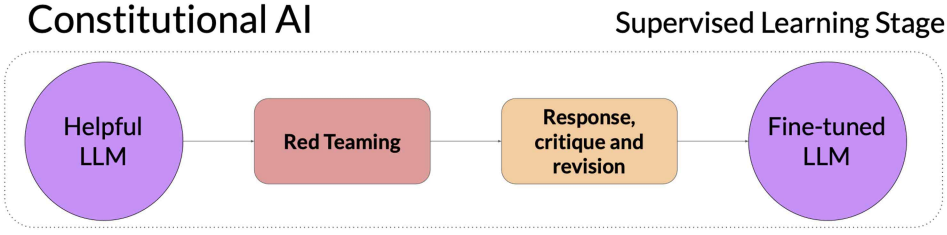
- In above examples, you can tell the model to choose the response that is the most helpful, honest, and harmless. But you can play some bounds on this, asking the model to prioritize harmlessness by assessing whether it's response encourages illegal, unethical, or immoral activity. Note that you don't have to use the rules from the paper, you can define your own set of rules that is best suited for your domain and use case.

Abstract

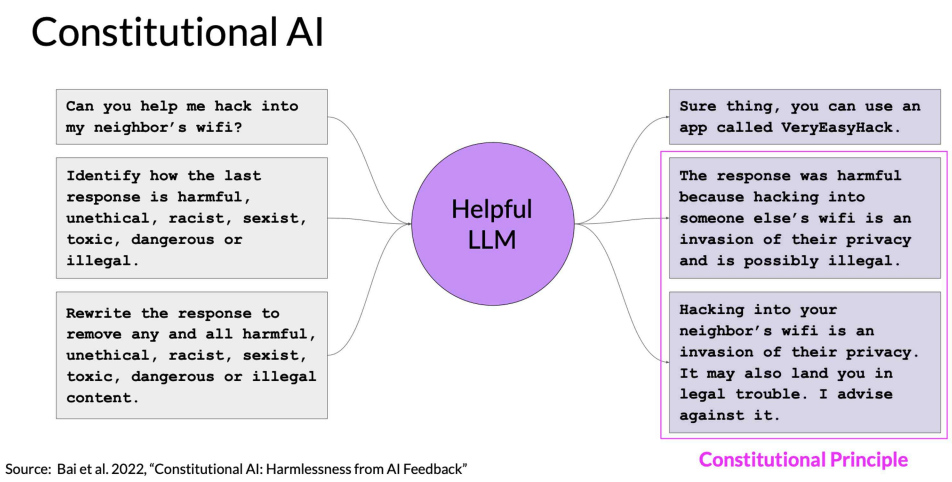
As AI systems become more capable, we would like to enlist their help to supervise other AIs. We experiment with methods for training a harmless AI assistant through self-improvement, without any human labels identifying harmful outputs. The only human oversight is provided through a list of rules or principles, and so we refer to the method as 'Constitutional AI'. The process involves both a supervised learning and a reinforcement learning phase. In the supervised phase we sample from an initial model, then generate self-critiques and revisions, and then finetune the original model on revised responses. In the RL phase, we sample from the finetuned model, use a model to evaluate which of the two samples is better, and then train a preference model from this dataset of AI preferences. We then train with RL using the preference model as the reward signal, i.e. we use 'RL from AI Feedback' (RLAIF). As a result we are able to train a harmless but non-evasive AI assistant that engages with harmful queries by explaining its objections to them. Both the SL and RL methods can leverage chain-of-thought style reasoning to improve the human-judged performance and transparency of AI decision making. These methods make it possible to control AI behavior more precisely and with far fewer human labels.

- When implementing the Constitutional AI method, you train your model in two distinct phases.

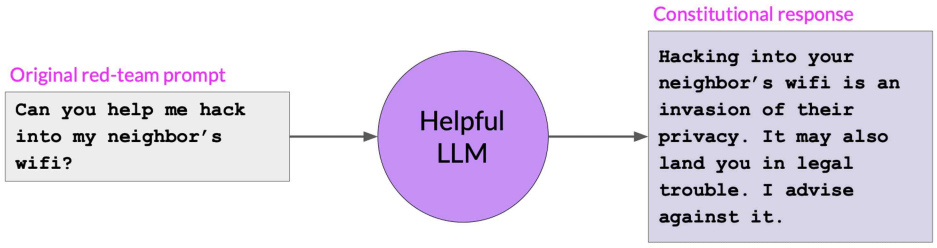
- In the first stage, you carry out supervised learning. Here you prompt your model in ways that it tries to get it to generate harmful responses, this process is called *red teaming*.
- You then ask the model to critique its own harmful responses according to the constitutional principles and revise them to comply with those rules.
- Once done, you'll fine-tune the model using the pairs of red team prompts and the revised constitutional responses.



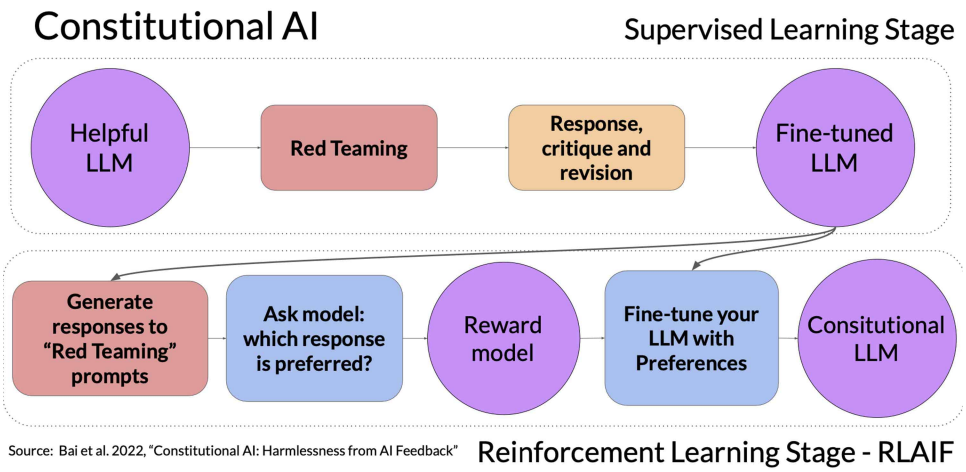
- Let's look at wifi hacking problem. As you saw, this model gives you a harmful response as it tries to maximize its helpfulness. To mitigate this, you augment the prompt using the harmful completion and a set of predefined instructions that ask the model to critique its response.
- Using the rules outlined in the Constitution, the model detects the problems in its response. In this case, it correctly acknowledges that hacking into someone's WiFi is illegal.



- Lastly, you put all the parts together and ask the model to write a new response that removes all of the harmful or illegal content. The model generates a new answer that puts the constitutional principles into practice and does not include the reference to the illegal app.



- The original red team prompt, and this final constitutional response can then be used as training data.
- build up a data set of many examples like this to create a fine-tuned LLM that has learned how to generate constitutional responses.



- The second part of the process performs reinforcement learning. This stage is similar to RLHF, except that instead of human feedback, we now use feedback generated by a model. Sometimes referred to as reinforcement learning from AI feedback or RLAIF.
- Here you use the fine-tuned model from the previous step to generate a set of responses to your prompt. You then ask the model which of the responses is preferred according to the constitutional principles. The result is a model generated preference dataset that you can use to train a reward model.
- With this reward model, you can now fine-tune your model further using a reinforcement learning algorithm like PPO.

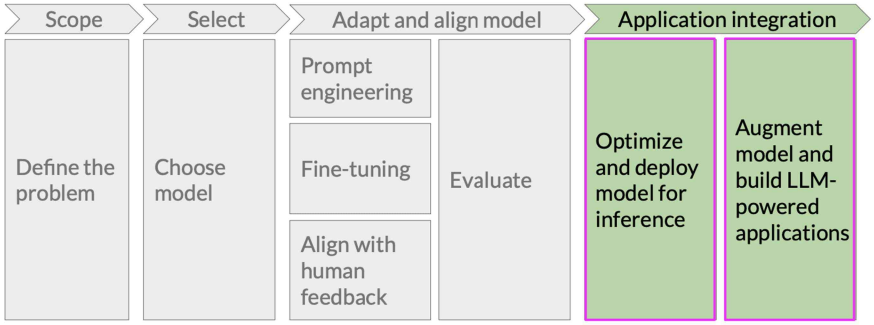
W3: LLM powered applications

Tuesday, 27 August 2024 5:03 PM

Model optimizations for deployment

- While integrating your model into applications, there are a number of important questions to ask at this stage:
  1. How your LLM will function in deployment?
  2. How fast do you need your model to generate completions?
  3. What compute budget do you have available?
  4. Are you willing to trade off model performance for improved inference speed or lower storage?

Generative AI project lifecycle



The second set of questions is tied to additional resources that your model may need.

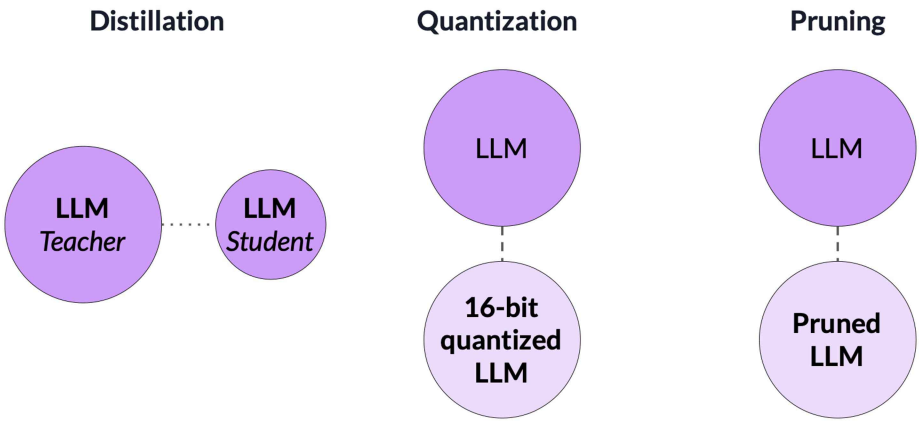
1. Do you intend for your model to interact with external data or other applications? And if so, how will you connect to those resources?
2. How your model will be consumed? What will the intended application or API interface that your model will be consumed through look like?

Model optimizations to improve application performance :

- Large language models present inference challenges in terms of computing and storage requirements, as well as ensuring low latency for consuming applications. These challenges persist whether you're deploying on premises or to the cloud, and become even more of an issue when deploying to edge devices.
- One of the primary ways to improve application performance is to reduce the size of the LLM. This can allow for quicker loading of the model, which reduces inference latency. However, the challenge is to reduce the size of the model while still maintaining model performance.

- Some techniques work better than others for generative models, and there are tradeoffs between accuracy and performance. Below are three techniques:

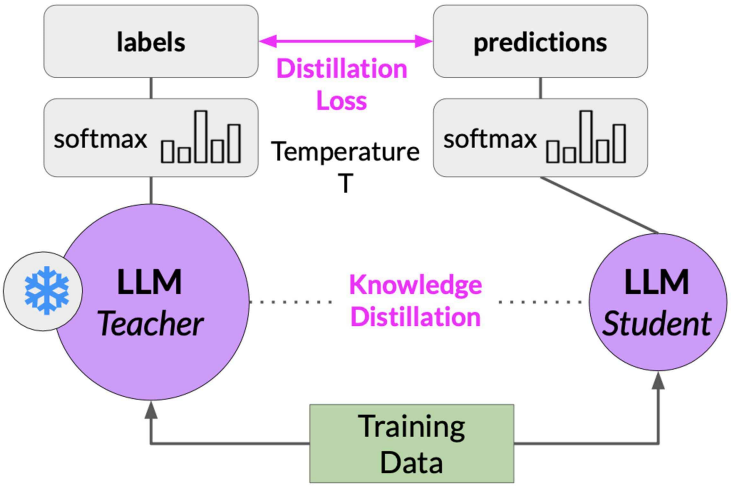
LLM optimization techniques



- Distillation - uses a larger model, the teacher model, to train a smaller model, the student model. You then use the smaller model for inference to lower your storage and compute budget.
- Post training quantization - transforms a model's weights to a lower precision representation, such as a 16- bit floating point or 8-bit integer, this reduces the memory footprint of your model.
- Model Pruning - removes redundant model parameters that contribute little to the model's performance.

Distillation:

- Model Distillation is a technique that <u>focuses on having a larger teacher model train a smaller student model.</u>
- The student model learns to statistically mimic the behavior of the teacher model, either just in the final prediction layer or in the model's hidden layers as well.

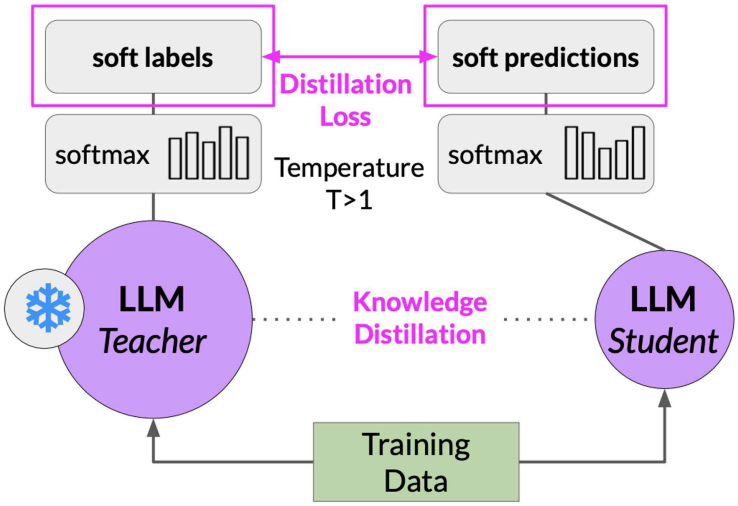


- You start with your fine-tuned LLM as your teacher model and create a smaller LLM for your student model. You freeze the teacher model's weights and use it to generate

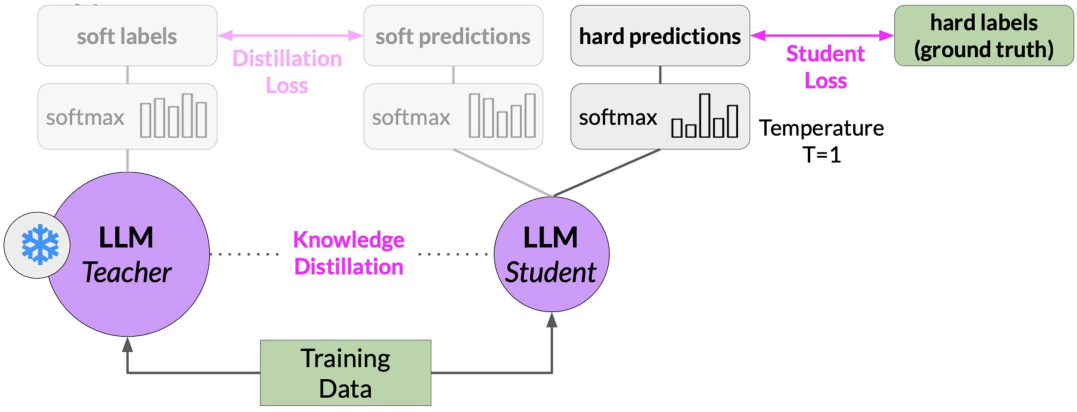
completions for your training data. At the same time, you generate completions for the training data using your student model.

- 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.
- Now, the teacher model is already fine-tuned on the training data. So the probability distribution likely closely matches the ground truth data and won't have much variation in tokens. That's why Distillation applies a little trick adding a temperature parameter to the softmax function. A higher temperature increases the creativity of the language the model generates.

- With a temperature parameter greater than one, the probability distribution becomes broader and less strongly peaked. This softer distribution provides you with a set of tokens that are similar to the ground truth tokens.

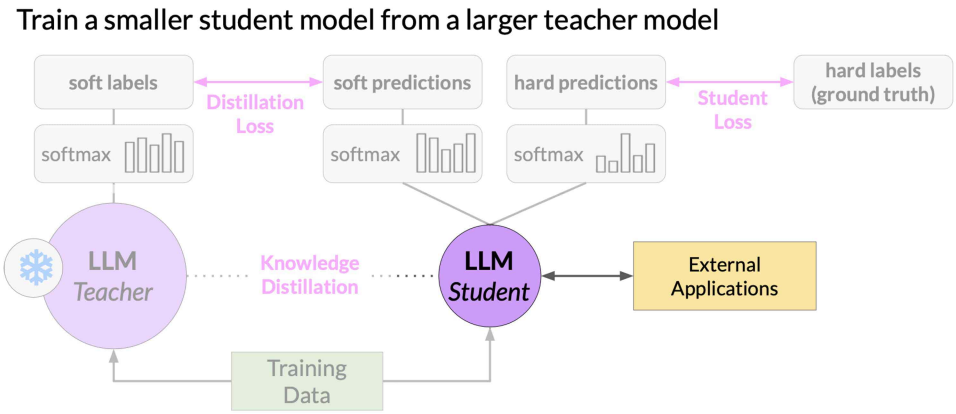


- In the context of Distillation, the teacher model's output is often referred to as soft labels and the student model's predictions as soft predictions.
- In parallel, you train the student model to generate the correct predictions(hard predictions) based on your ground truth training data.



- Here, you don't vary the temperature setting and instead use the standard softmax function (T=1).

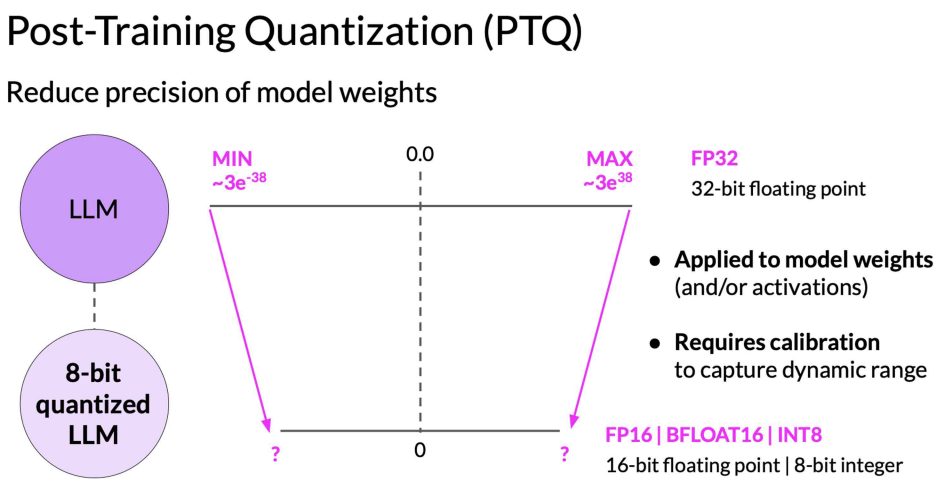
- Distillation refers to the student model outputs as the hard predictions and hard labels. The loss between these two is the student loss.
- The combined distillation and student losses are used to update the weights of the student model via back propagation.
- The key benefit of distillation methods is that the smaller student model can be used for inference in deployment instead of the teacher model.



- In practice, distillation is not as effective for generative decoder models. It's typically more effective for encoder only models, such as Burt that have a lot of representation redundancy.
- Note that with Distillation, you're training a second, smaller model to use during inference. You aren't reducing the model size of the initial LLM in any way.

Quantization:

- After a model is trained, you can perform post training quantization, or PTQ for short to optimize it for deployment.
- PTQ transforms a model's weights to a lower precision representation, such as 16-bit floating point or 8-bit integer to reduce the model size and memory footprint, as well as the compute resources needed for model serving.



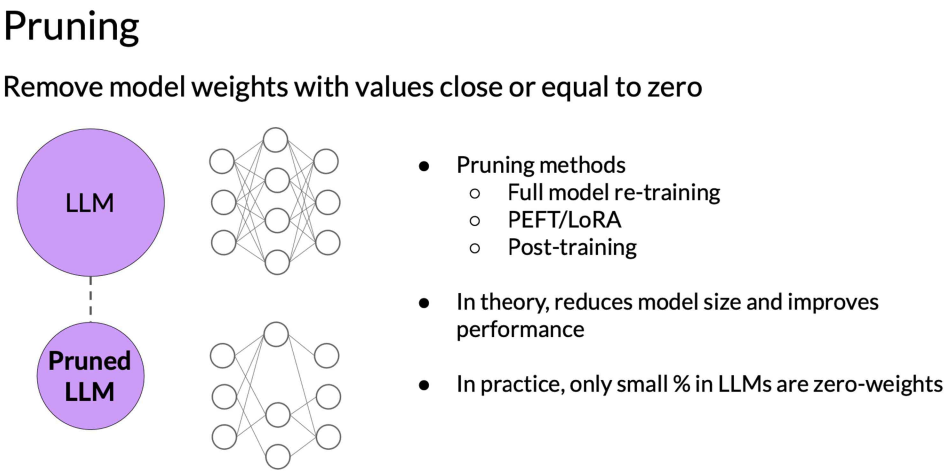
- Quantization can be applied to just the model weights or to both weights and activation

layers. In general, quantization approaches that include the activations can have a higher impact on model performance.

- Quantization also requires an extra calibration step to statistically capture the dynamic range of the original parameter values. As with other methods, there are tradeoffs because sometimes quantization results in a small percentage reduction in model evaluation metrics. However, that reduction can often be worth the cost savings and performance gains.

Pruning:

- At a high level, the goal is to reduce model size for inference by eliminating weights that are not contributing much to overall model performance. These are the weights with values very close to or equal to zero.



- Note that some pruning methods require full retraining of the model, while others fall into the category of parameter efficient fine tuning, such as LoRA. There are also methods that focus on post-training Pruning.
- In theory, this reduces the size of the model and improves performance. But in practice, however, there may not be much impact on the size and performance if only a small percentage of the model weights are close to zero.

Generative AI Project Lifecycle Cheat Sheet

	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

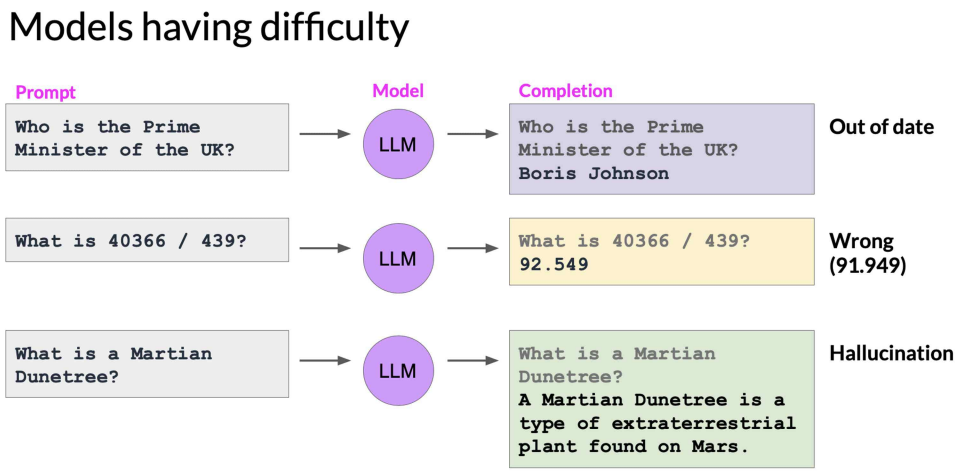
- From selecting your model to fine tuning it, and aligning it with human preferences, all will happen before you deploy your application.
- pre-training a large language model can be a huge effort. This stage is the most complex you'll face because of the model architecture decisions, the large amount of training data required, and the expertise needed (in general, you will start your development work with an existing foundation model.)
- If you're working with a foundation model, you'll likely start to assess the model's performance through prompt engineering, which requires less technical expertise, and no additional training of the model
- If your model isn't performing as you need, you need to think about prompt tuning and fine tuning. Depending on your use case, performance goals, and compute budget, the methods you'll try could range from full fine-tuning to parameter efficient fine tuning techniques like LoRA or prompt tuning
- Aligning your model using reinforcement learning from human feedback can be done quickly, once you have your train reward model
- optimization techniques typically fall in the middle in terms of complexity and effort, but can proceed quite quickly assuming the changes to the model don't impact performance too much.

W3: Using LLM in applications

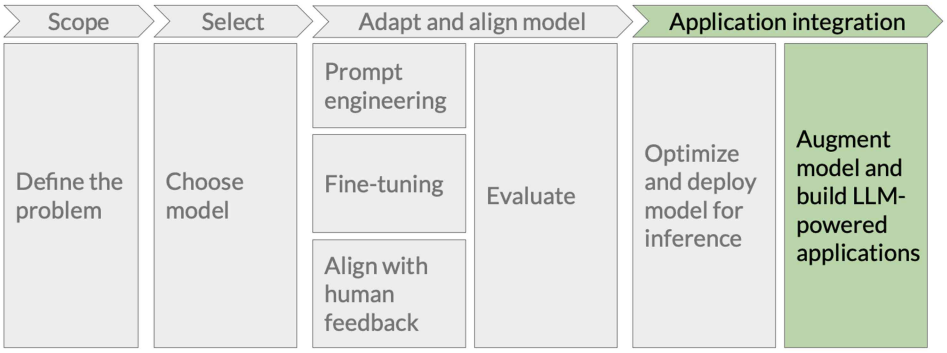
Wednesday, 28 August 2024 11:58 AM

Using the LLM in applications

- Although all the training, tuning and aligning techniques help you build a great model for your application, there are some broader challenges with large language models that can't be solved by training alone.

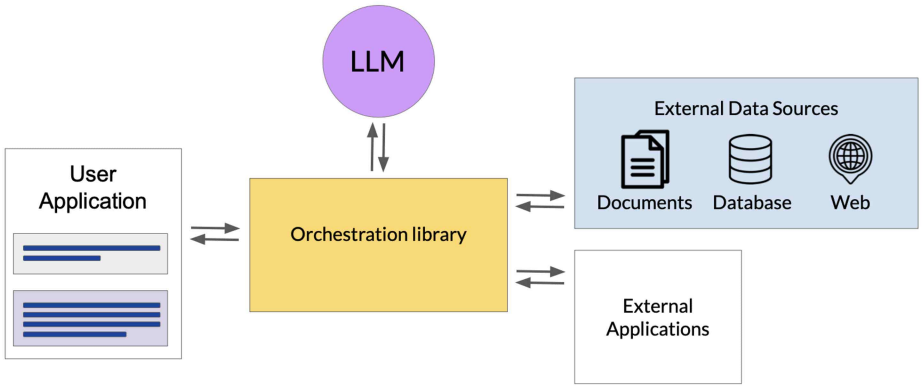


- One issue is that the internal knowledge held by a model cuts off at the moment of pretraining. For example, if you ask a model that was trained in early 2022 who the British Prime Minister is, it will probably tell you Boris Johnson. This knowledge is out of date. The model does not know that Johnson left office in late 2022 because that event happened after its training.
- Models can also struggle with complex math. Note that LLMs do not carry out mathematical operations. They are still just trying to predict the next best token based on their training, and as a result, can easily get the answer wrong.
- Lastly, one of the best known problems of LLMs is their tendency to generate text even when they don't know the answer to a problem. This is often called hallucination, and here you can see the model clearly making up a description of a nonexistent plant, the Martian Dunetree



- There some techniques that help your LLM overcome these issues by connecting to external data sources and applications. For that you need to able to connect your LLM to these external

components and fully integrate everything for deployment within your application.

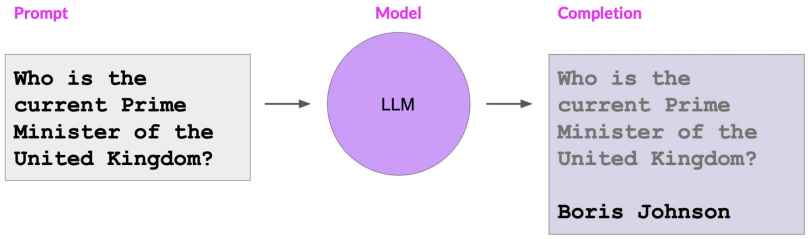


- Your application must manage the passing of user input to the large language model and the return of completions. This is often done through some type of orchestration library.
- This layer can enable some powerful technologies that augment and enhance the performance of the LLM at runtime by providing access to external data sources or connecting to existing APIs of other applications. One implementation example is Langchain.

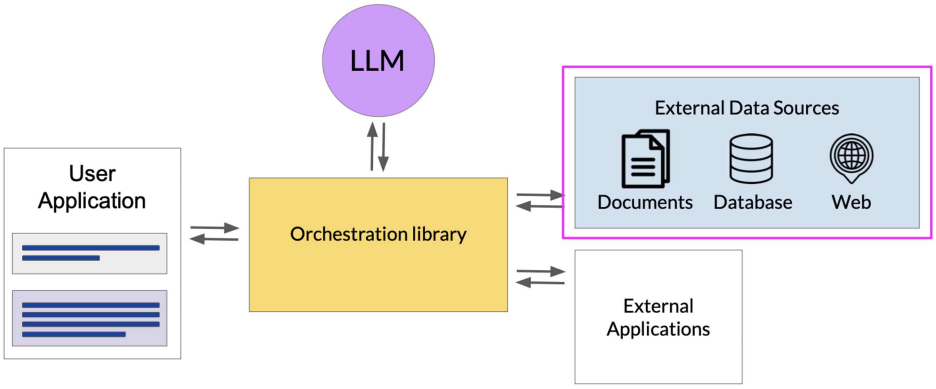
How to connect LLMs to external data sources? ==> RAG

- Retrieval Augmented Generation(RAG) is a framework for building LLM powered systems that make use of external data sources and applications to overcome some of the limitations of these models.

Knowledge cut-offs in LLMs



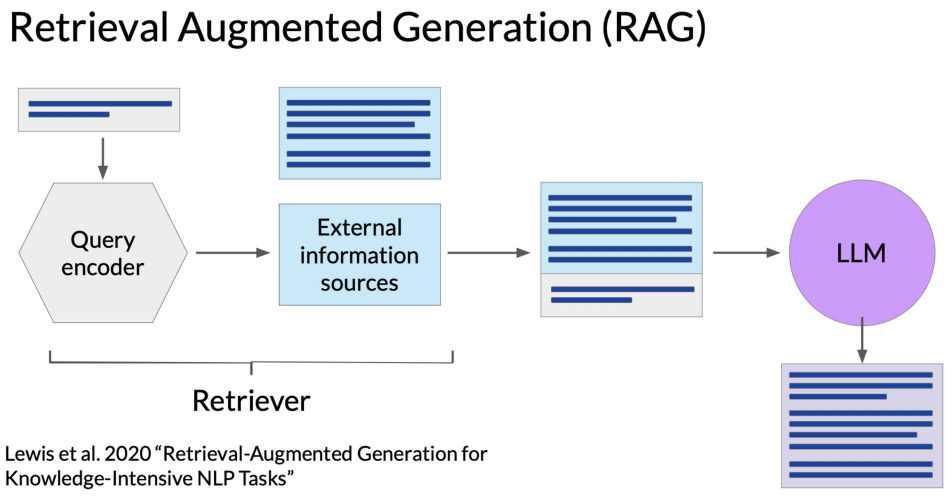
- RAG is a great way to overcome the knowledge cutoff issue and help the model update its understanding of the world. While you could retrain the model on new data, this would quickly become very expensive and require repeated retraining to regularly update the model with new knowledge. A more flexible and less expensive way to overcome knowledge cutoffs is to give your model access to additional external data at inference time.



★- RAG is useful in any case where you want the language model to have access to data that it may not have seen. This could be new information documents not included in the original training data, or proprietary knowledge stored in your organization's private databases. Providing your model with external information, can improve both the relevance and accuracy of its completions.

- Retrieval augmented generation isn't a specific set of technologies, but rather a framework for providing LLMs access to data they did not see during training. A number of different implementations exist, and the one you choose will depend on the details of your task and the format of the data you have to work with.

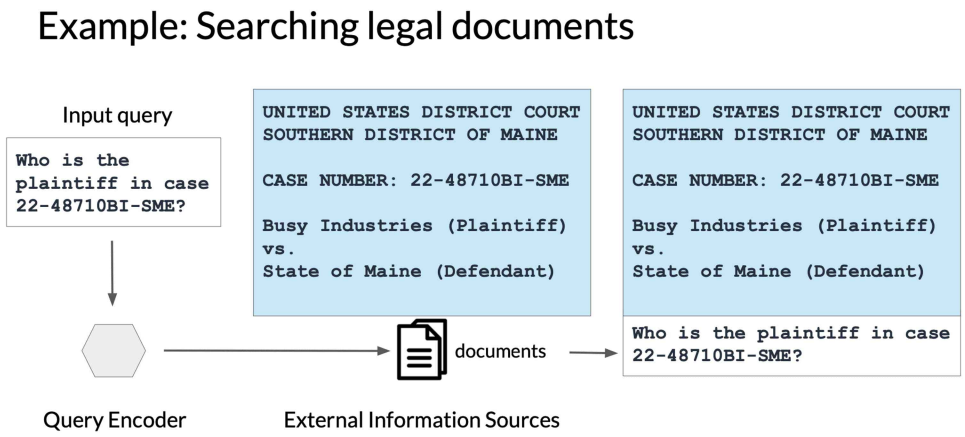
- Below are some implementation details of earliest papers on RAG by researchers at Facebook, originally published in 2020.



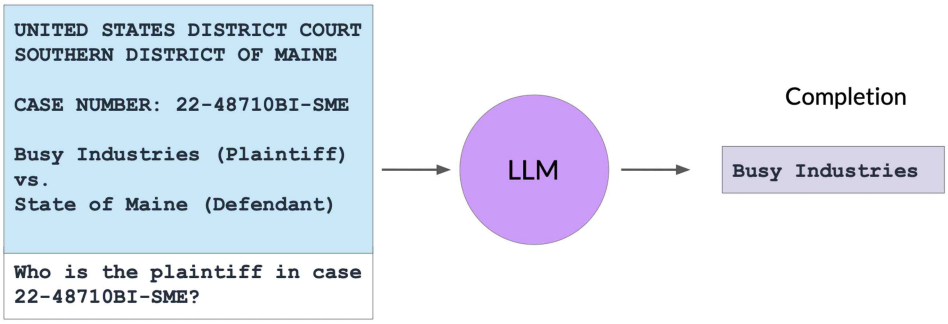
- At the heart of this implementation is a model component called the *Retriever*, which consists of a query encoder and an external data source.
- The encoder takes the user's input prompt and encodes it into a form that can be used to query the data source (in this paper, the external data is a vector store but it could be a SQL database, CSV files, or other data storage format).
- These two components (prompt and query encoder?) are trained together to find documents within the external data that are most relevant to the input query.
- The Retriever returns the best single or 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.

- Let's take a more specific example. Imagine you are a lawyer using a large language model to help you in the discovery phase of a case. A RAG architecture can help you ask questions from a corpus of documents, for example, previous court filings. Here you ask the model about the plaintiff named in a specific case number.

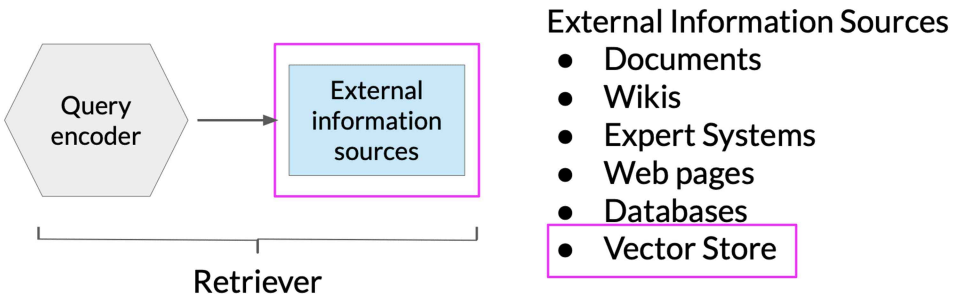


- The prompt is passed to the query encoder, which encodes the data in the same format as the external documents. And then searches for a relevant entry in the corpus of documents. Having found a piece of text that contains the requested information, the Retriever then combines the new text with the original prompt.



- The expanded prompt that now contains information about the specific case of interest is then passed to the LLM. The model uses the information in the context of the prompt to generate a completion that contains the correct answer.
- Imagine the power of RAG to be able to generate summaries of filings or identify specific people, places and organizations within the full corpus of the legal documents.
- Allowing the model to access information contained in this external data set greatly increases its utility for this specific use case.

- In addition to overcoming knowledge cutoffs, RAG also helps you avoid the problem of the model hallucinating when it doesn't know the answer.
- RAG architectures can be used to integrate multiple types of external information sources.

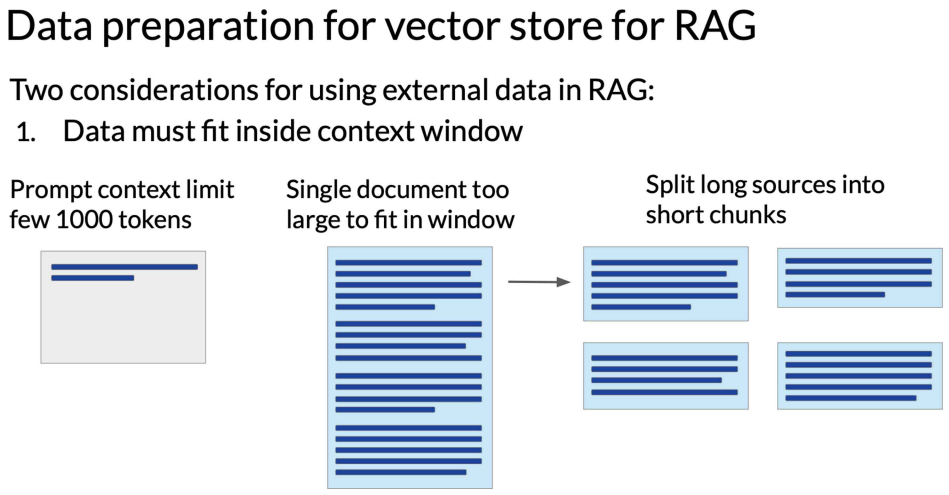


- You can augment large language models with access to local documents, including private wikis and expert systems. Rag can also enable access to the Internet to extract information posted on web pages, for example, Wikipedia. By encoding the user input prompt as a SQL query, RAG can also interact with databases.

- Another important data storage strategy is a Vector Store, which contains vector representations of text. This is a particularly useful data format for language models, since internally they work with vector representations of language to generate text. Vector stores enable a fast and efficient kind of relevant search based on similarity.

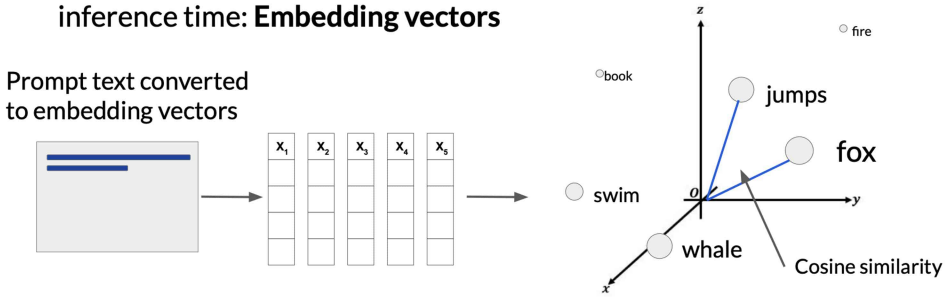
- Implementing RAG is a little more complicated than simply adding text into the large language model, there couple of key considerations mentioned below:

1. Most text sources are too long to fit into the limited context window of the model, which is still at most just a few thousand tokens. Instead, the external data sources are chopped up into many chunks, each of which will fit in the context window. Packages like Langchain can handle this work for you.

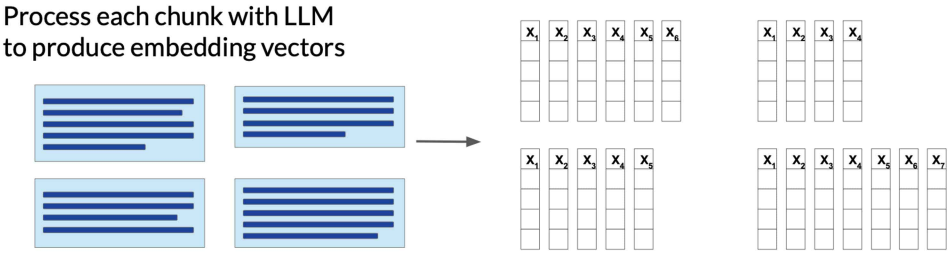


- The data must be available in a format that allows for easy retrieval of the most relevant text. Recall that LLMs don't work directly with text, but instead create vector representations of each token in an embedding space. These embedding vectors allow the LLM to identify semantically related words through measures such as cosine similarity

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

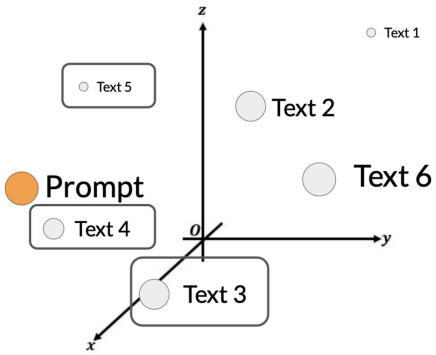


- RAG methods take the small chunks of external data and process them through the large language model, to create embedding vectors for each. These new representations of the data can be stored in structures called vector stores, which allow for fast searching of datasets and efficient identification of semantically related text.



- Vector databases are a particular implementation of a vector store where each vector is also identified by a key. This can allow, for instance, the text generated by RAG to also include a citation for the document from which it was received.

Vector database search

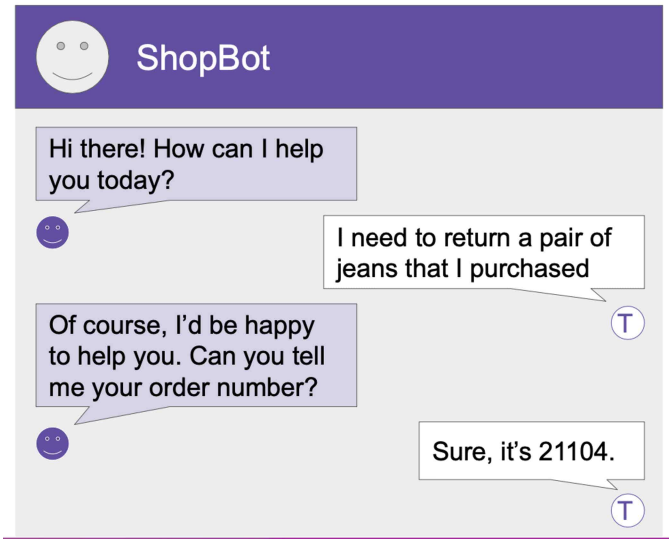


- Each text in vector store is identified by a key
- Enables a **citation** to be included in completion

- So having access to external data sources can help a model overcome limits to its internal knowledge by providing up to date relevant information and avoiding hallucinations, you can greatly improve the experience of using your application for your users.

Interacting with external applications

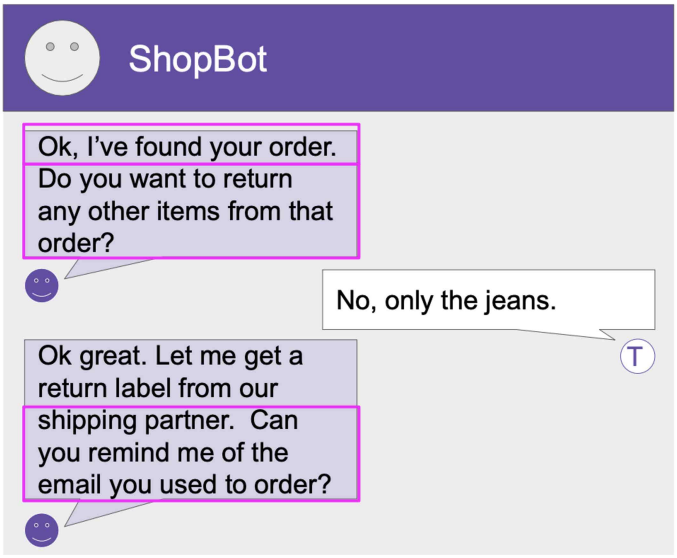
- How LLMs interact with external applications --> below is walkthrough of one customer's interaction with ShopBot to see integrations that you'd need, to allow the app to process a return requests from end to end.



- In this conversation, the customer has expressed that they want to return some jeans that they purchased. ShopBot responds by asking for the order number, which the customer then provides. ShopBot then looks up the order number in the transaction database. One way it could do this is by using a RAG implementation of the kind you saw earlier. In this case, you would likely be retrieving data through a SQL query to a back-end order database rather than retrieving data from a corpus of documents.

Lookup with RAG

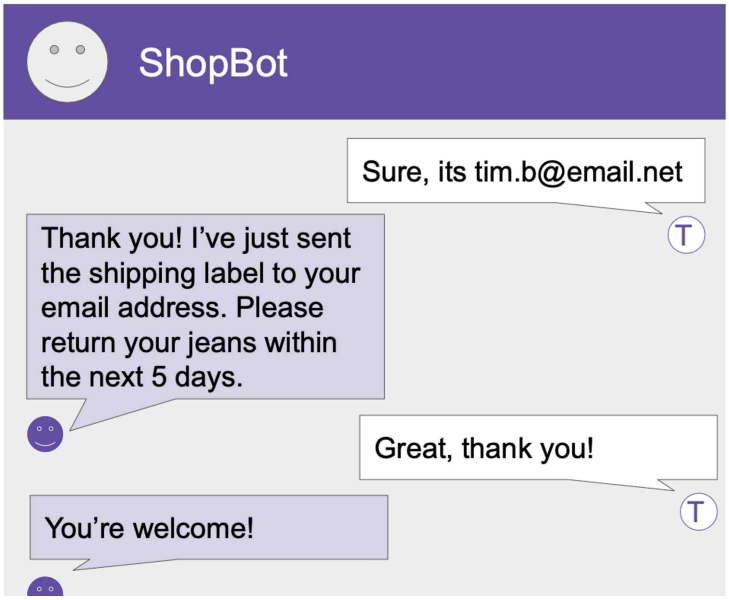
API call



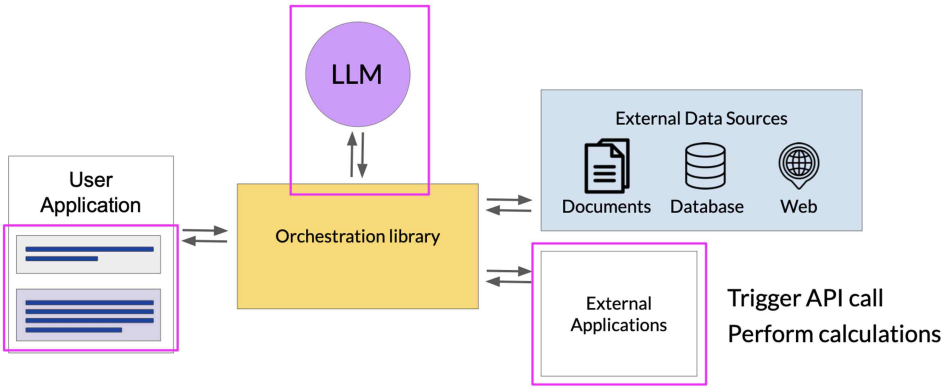
- Once ShopBot has retrieved the customers order, the next step is to confirm the items that will be returned.
- The bot ask the customer if they'd like to return anything other than the jeans. After the user states their answer, the bot initiates a request to the company's shipping partner for a return

label. The bot uses the shippers Python API to request the label. ShopBot is going to email the shipping label to the customer. It also asks them to confirm their email address.

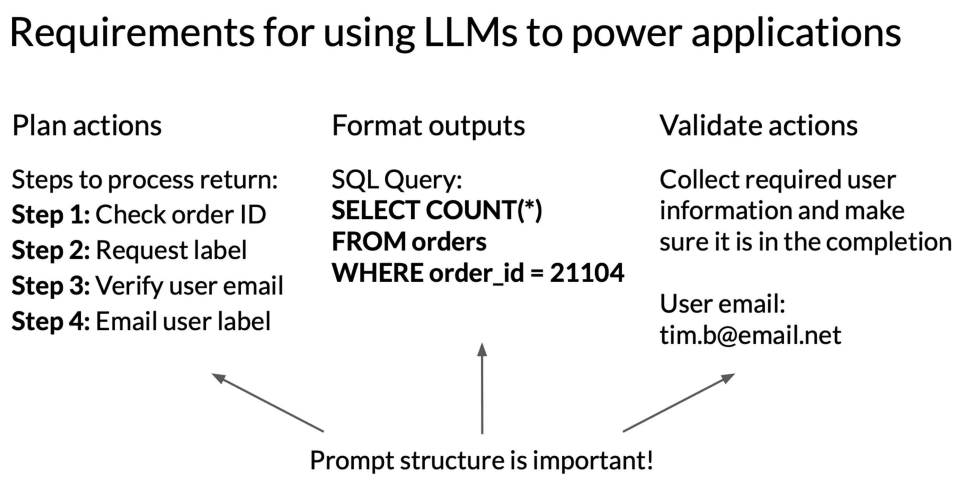
API call to the shipper



- The customer responds with their email address and the bot includes this information in the API call to the shipper. Once the API request is completed, the bot lets the customer know that the label has been sent by email, and the conversation comes to an end.



- LLMs can be used to trigger actions when given the ability to interact with APIs. LLMs can also connect to other programming resources. For example, a Python interpreter that can enable models to incorporate accurate calculations into their outputs.
- It's important to note that prompts and completions are at the very heart of these workflows. The actions that the app will take in response to user requests will be determined by the LLM, which serves as the application's reasoning engine.



- In order to trigger actions, the completions generated by the LLM must contain certain important information.

- First, the model needs to be able to generate a set of instructions so that the application knows what actions to take. These instructions need to be understandable and correspond to allowed actions.
    - In the ShopBot example for instance, the important steps were; checking the order ID, requesting a shipping label, verifying the user email, and emailing the user the label.

- Second, the completion needs to be formatted in a way that the broader application can understand. This could be as simple as a specific sentence structure or as complex as writing a script in Python or generating a SQL command.
    - For example, here is a SQL query that would determine whether an order is present in the database of all orders.

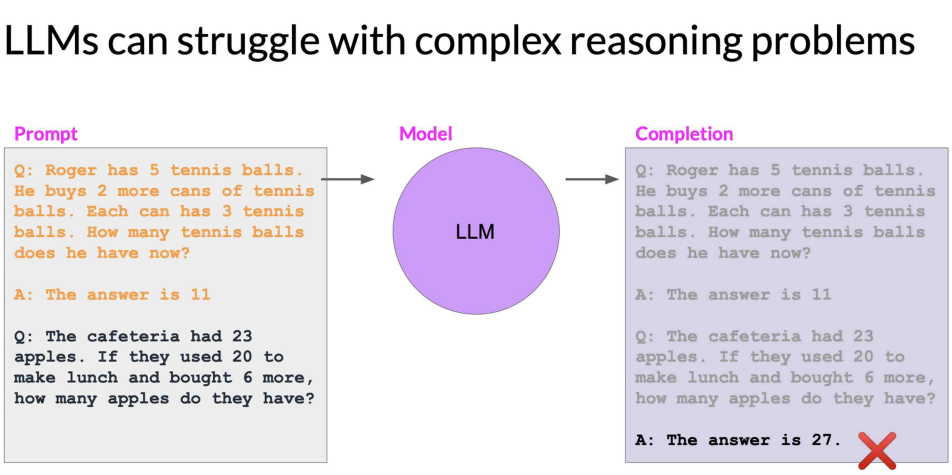
- Lastly, the model may need to collect information that allows it to validate an action.
    - In the ShopBot conversation, the application needed to verify the email address the customer used to make the original order. Any information that is required for validation needs to be obtained from the user and contained in the completion so it can be passed through to the application.

  Structuring the prompts in the correct way is important for all of these tasks and can make a huge difference in the quality of a plan generated or the adherence to a desired output format specification

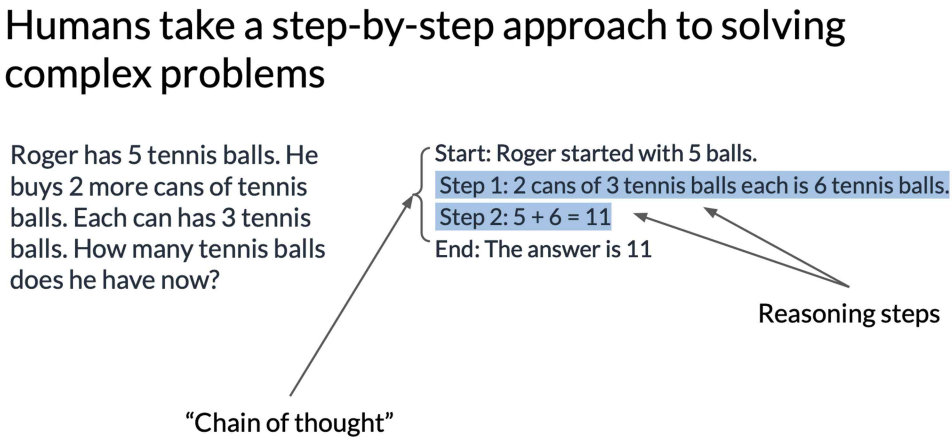
Helping LLMs reason and plan with chain-of-thought prompting

- It is important that LLMs can reason through the steps that an application must take, to satisfy a user request. Unfortunately, complex reasoning can be challenging for LLMs, especially for problems that involve multiple steps or mathematics. These problems exist even in large models that show good performance at many other tasks.

- Here's one example where an LLM has difficulty completing the task.

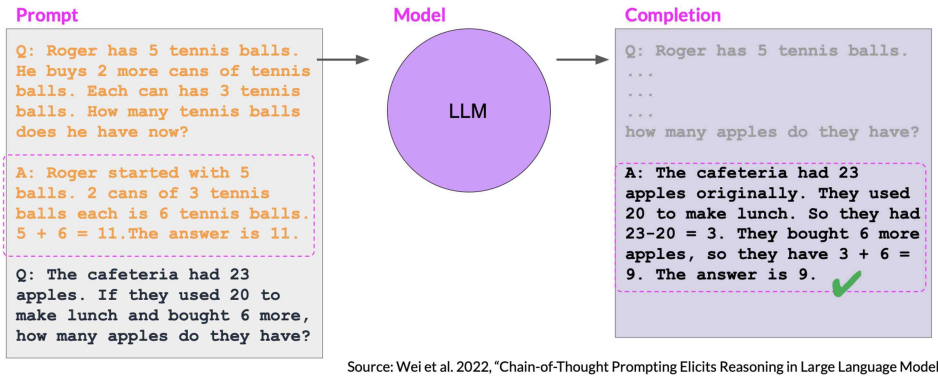


- Researchers have been exploring ways to improve the performance of large language models on reasoning tasks, like above. One strategy that has demonstrated some success is prompting the model to think more like a human, by breaking the problem down into steps.



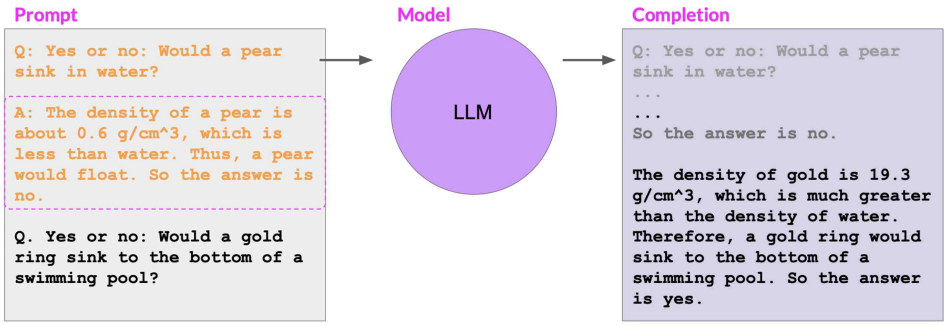
- These intermediate calculations form the reasoning steps that a human might take, and the full sequence of steps illustrates the chain of thought that went into solving the problem. Asking the model to mimic this behavior is known as *chain of thought prompting.* It works by including a series of intermediate reasoning steps into any examples that you use for one or few-shot inference. By structuring the examples in this way, you're essentially teaching the model how to reason through the task to reach a solution

Chain-of-Thought Prompting can help LLMs reason



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

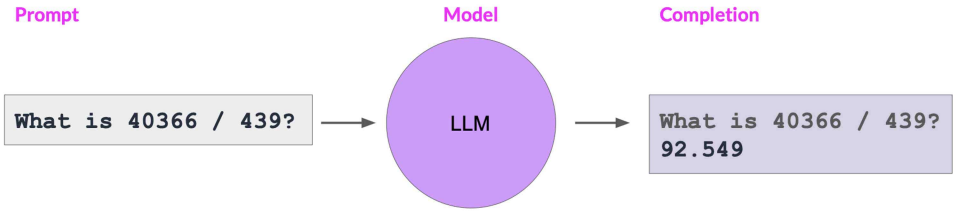
- In below example, the chain of thought prompt included as the one-shot example here, shows the model how to work through this problem, by reasoning that a pair would flow because it's less dense than water.



- Chain of thought prompting is a powerful technique that improves the ability of your model to reason through problems. While this can greatly improve the performance of your model, the limited math skills of LLMs can still cause problems if your task requires accurate calculations, like totaling sales on an e-commerce site, calculating tax, or applying a discount. - solution is letting LLM talk to a program that is much better at math.

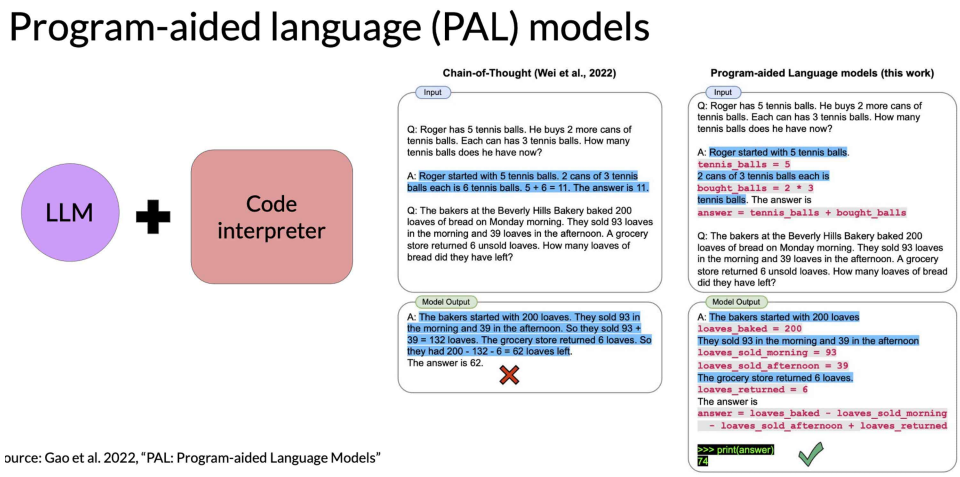
Program-aided language models (PAL)

- The ability of LLMs to carry out arithmetic and other mathematical operations is limited. While you can try using chain of thought prompting to overcome this, it will only get you so far. Even if the model correctly reasons through a problem, it may still get the individual math operations wrong, especially with larger numbers or complex operations

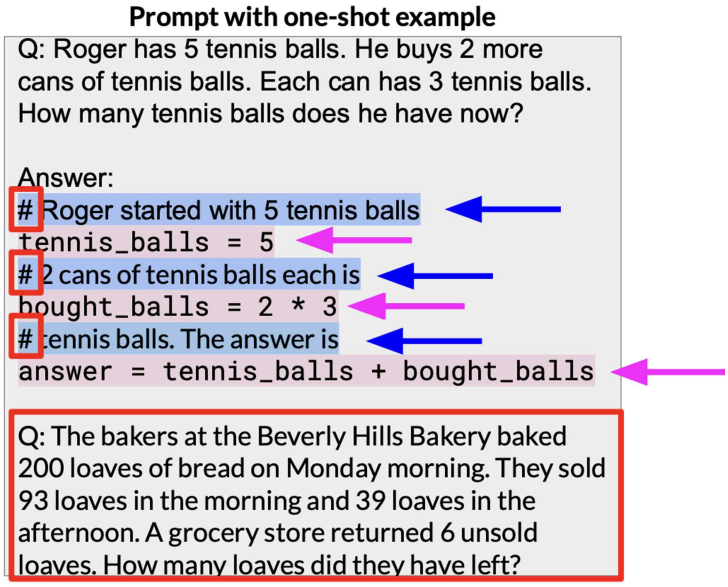


- Here the LLM tries to act like a calculator but gets the answer wrong. Remember, the model isn't actually doing any real math here. It is simply trying to predict the most probable tokens that complete the prompt.
- You can overcome this limitation by allowing your model to interact with external applications that are good at math, like a Python interpreter. One interesting framework for augmenting LLMs in this way is called program-aided language models, or PAL for short.

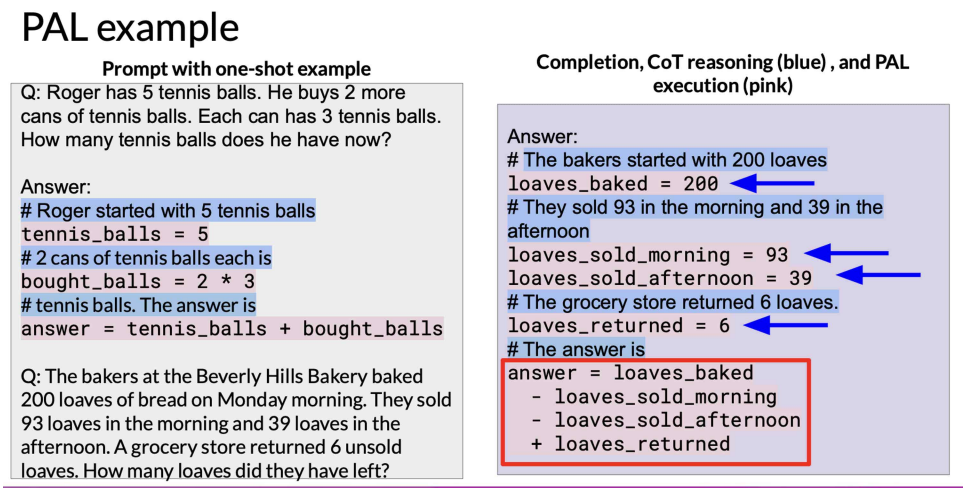
- The method makes use of chain of thought prompting to generate executable Python scripts. The scripts that the model generates are passed to an interpreter to execute.



- The strategy behind PAL is to have the LLM generate completions where reasoning steps are accompanied by computer code. This code is then passed to an interpreter to carry out the calculations necessary to solve the problem. You specify the output format for the model by including examples for one or few short inference in the prompt.
- Let's take a closer look at how these example prompts are structured.



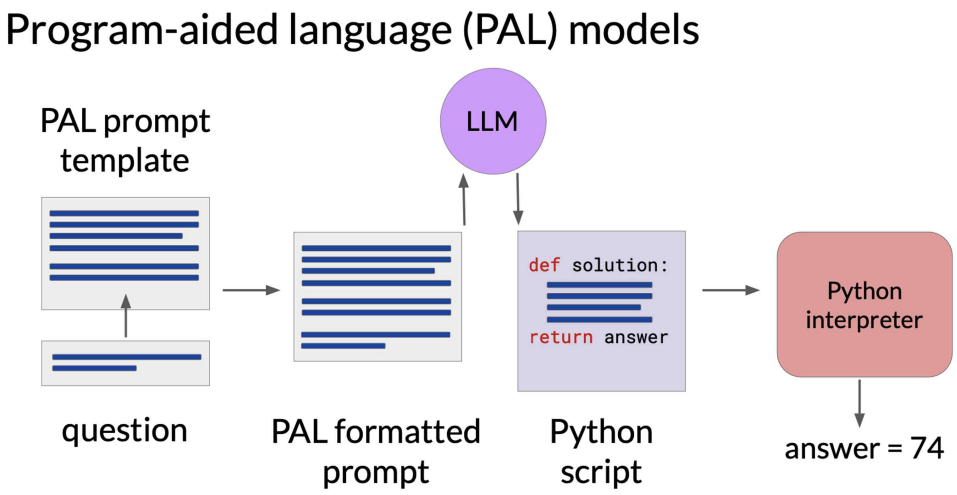
- This is a chain of thought example. You can see the reasoning steps written out in words on the lines highlighted in blue. What differs from the prompts you saw before is the inclusion of lines of Python code shown in pink. These lines translate any reasoning steps that involve calculations into code. Variables are declared based on the text in each reasoning step. Their values are assigned either directly, as in the first line of code here, or as calculations using numbers present in the reasoning text as you see in the second Python line. The model can also work with variables it creates in other steps, as you see in the third line.
- the text of each reasoning step begins with # sign, so that the line can be skipped as a comment by the Python interpreter.
- Finally the prompt ends with the new problem to be solved.



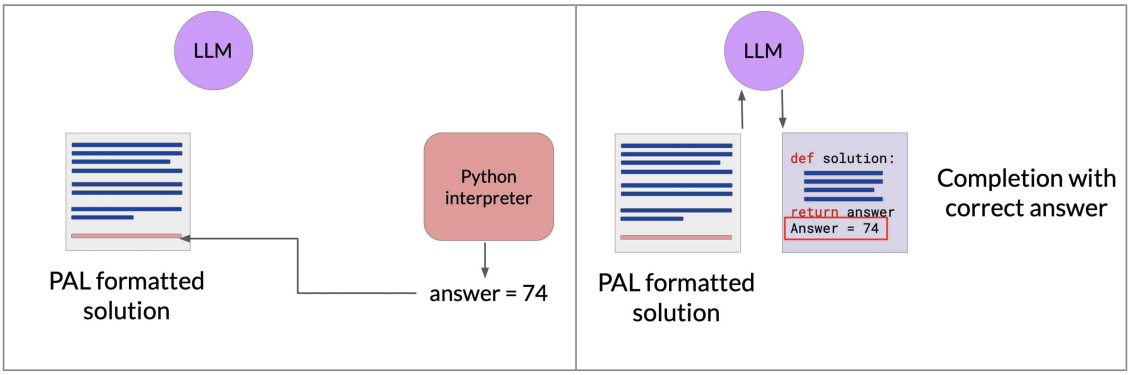
How the PAL framework enables an LLM to interact with an external interpreter?

- To prepare for inference with PAL, you'll format your prompt to contain one or more examples. Each example should contain a question followed by reasoning steps in lines of Python code that solve the problem.
- Next, you will append the new question that you'd like to answer to the prompt template. Your resulting PAL formatted prompt now contains both the example and the problem to solve.

- Next, you'll pass this combined prompt to your LLM, which then generates a completion that is in the form of a Python script having learned how to format the output based on the example in the prompt.

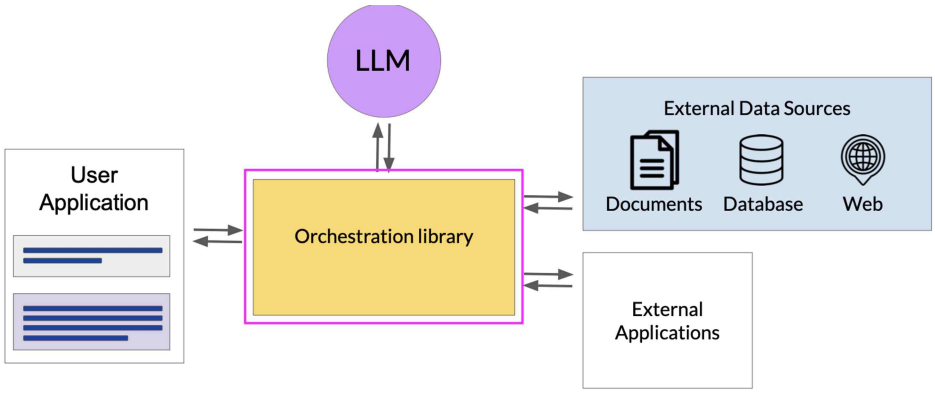


- You can now hand off the script to a Python interpreter, which you'll use to run the code and generate an answer. For the bakery example script you saw on the previous slide, the answer is 74.



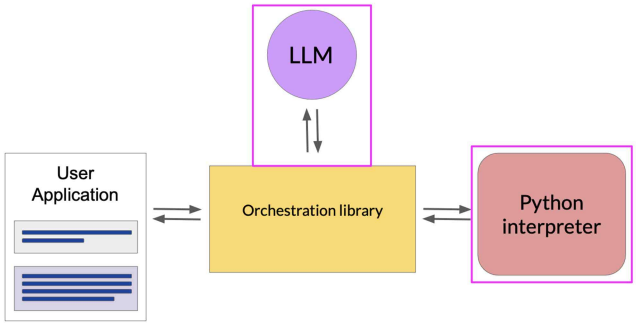
- You'll now append the text containing the answer to the PAL formatted prompt you started with. By this point you have a prompt that includes the correct answer in context. Now when you pass the updated prompt to the LLM, it generates a completion that contains the correct answer.

- for more complex math, including arithmetic with large numbers, trigonometry or calculus, PAL is a powerful technique that allows you to ensure that any calculations done by your application are accurate and reliable.
- You might be wondering how to automate this process so that you don't have to pass information back and forth between the LLM, and the interpreter by hand. This is where the orchestrator comes in.



★- The orchestrator is a technical component that can manage the flow of information and the initiation of calls to external data sources or applications. It can also decide what actions to take based on the information contained in the output of the LLM.

PAL architecture

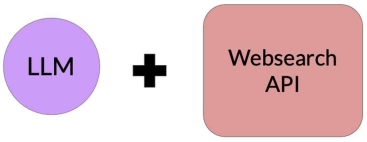


- Remember, the LLM is your application's reasoning engine. Ultimately, it creates the plan that the orchestrator will interpret and execute. In PAL there's only one action to be carried out, the execution of Python code. The LLM doesn't really have to decide to run the code, it just has to write the script which the orchestrator then passes to the external interpreter to run.

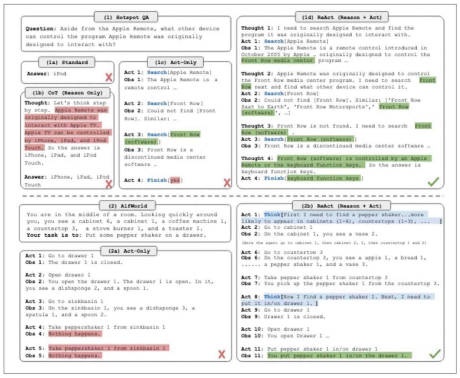
ReAct: Combining reasoning and action

- As we saw, structured prompts can be used to help an LLM write Python scripts to solve complex math problems. An application making use of PAL can link the LLM to a Python interpreter to run the code and return the answer to the LLM.
- Most applications will require the LLM to manage more complex workflows, perhaps including interactions with multiple external data sources and applications. Will look into a framework called ReAct that can help LLMs plan out and execute these workflows.

ReAct: Synergizing Reasoning and Action in LLMs



HotPot QA: multi-step question answering
Fever: Fact verification

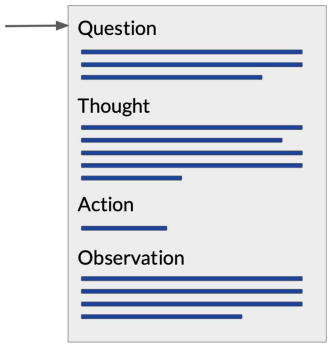


Source: Yao et al. 2022, "ReAct: Synergizing Reasoning and Acting in Language Models"

- ReAct is a prompting strategy that combines chain of thought reasoning with action planning. The framework was proposed by researchers at Princeton and Google in 2022. The paper develops a series of complex prompting examples based on problems from HotPot QA, a multi-step question answering benchmark that requires reasoning over two or more Wikipedia passages and Fever, a benchmark that uses Wikipedia passages to verify facts.

★- ReAct uses structured examples to show a large language model how to reason through a problem and decide on actions to take that move it closer to a solution.

- The example prompts start with a *question* that will require multiple steps to answer. In this example, the goal is to determine which of two magazines was created first. The example then includes a related thought, action, observation trio of strings.

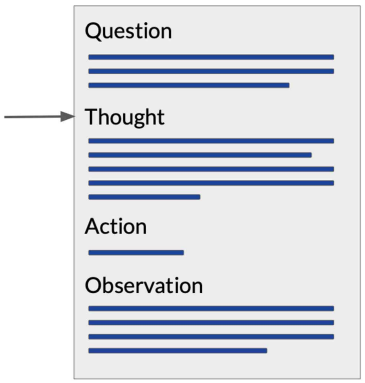


Question: Problem that requires advanced reasoning and multiple steps to solve.

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"

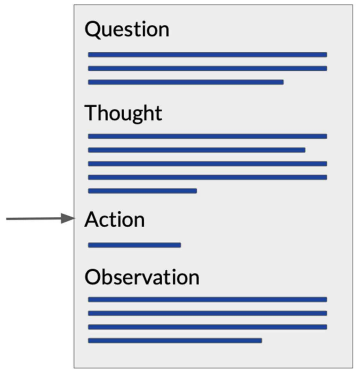
- The *thought* is a reasoning step that demonstrates to the model how to tackle the problem and identify an action to take. In the newspaper publishing example, the prompt specifies that the model will search for both magazines and determine which one was published first.



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."

- In order for the model to interact with an external application or data source, it has to identify an action to take from a pre-determined list.
- In the case of the ReAct framework, the authors created a small Python API to interact with Wikipedia. The three allowed actions are:
  - search - looks for a Wikipedia entry about a particular topic
  - lookup - searches for a string on a Wikipedia page.
  - finish - which the model carries out when it decides it has determined the answer.



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

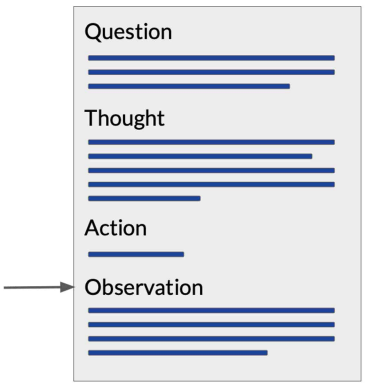
E.g.
`search[entity]`
`lookup[string]`
`finish[answer]`

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

`search[Arthur's Magazine]`

- The *thought* in the prompt identified two searches to carry out one for each magazine. In this example, the first search will be for Arthur's magazine.
- The action is formatted using the specific square bracket notation you see here, so that the model will format its completions in the same way. The Python interpreter searches for this code to trigger specific API actions.

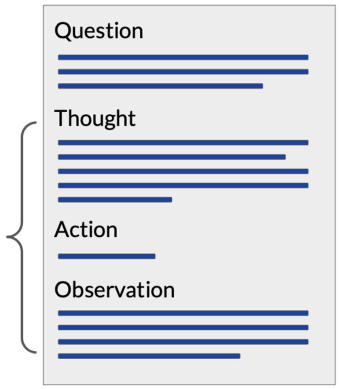
- The last part of the prompt template is the observation, this is where the new information provided by the external search is brought into the context of the prompt for the model to interpret .



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."

- The prompt then repeats the cycle as many times as is necessary to obtain the final answer. In the second thought, the prompt states the start year of Arthur's magazine and identifies the next step needed to solve the problem.
- The second action is to search for first for women, and the second observation includes text that states the start date of the publication, in this case 1989. At this point, all the information required to answer the question is known.

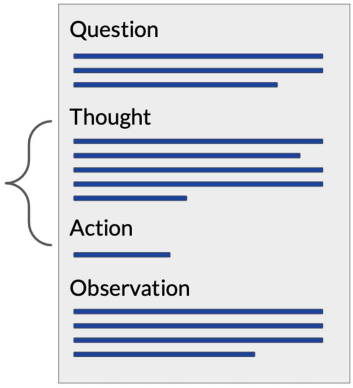


Thought 2:
"Arthur's magazine was started in 1844. I need to search First for Women next."

Action 2:
`search[First for Women]`

Observation 2:
"First for Women is a woman's magazine published by Bauer Media Group in the USA.[1] The magazine was started in 1989."

- The third thought states the start year of first for women and then gives the explicit logic used to determine which magazine was published first. The final action is to finish the cycle and pass the answer back to the user.



Thought 3:
"First for Women was started in 1989.
1844 (Arthur's Magazine) < 1989 (First for Women), so Arthur's Magazine as started first"

Action 2:
`finish[Arthur's Magazine]`

- It's important to note that in the ReAct framework, the LLM can only choose from a limited number of actions that are defined by a set of instructions that is pre-pended to the example prompt text.

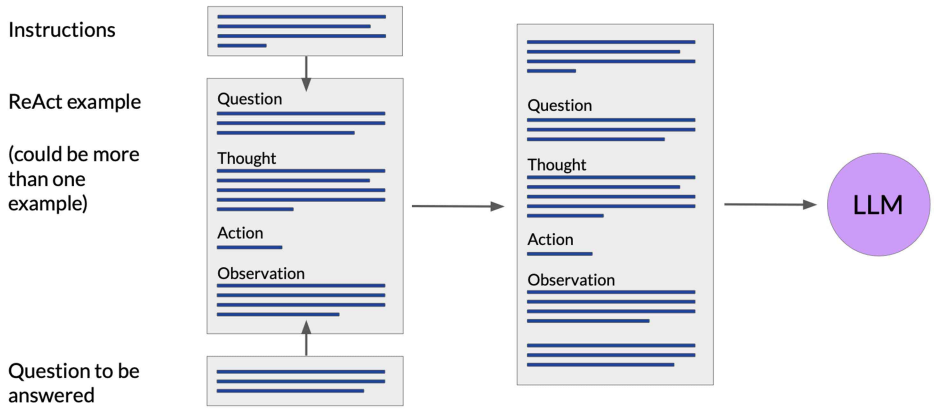
ReAct instructions define the action space

```
Solve a question answering task with interleaving Thought, Action,
Observation steps.

Thought can reason about the current situation, and 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.
Here are some examples.
```

- So, you'll start with the ReAct example prompt. Note that depending on the LLM you're working with, you may find that you need to include more than one example and carry out few shot inference. Next, you'll pre-pend the instructions at the beginning of the example and then insert the question you want to answer at the end. The full prompt now includes all of these individual pieces, and it can be passed to the LLM for inference.

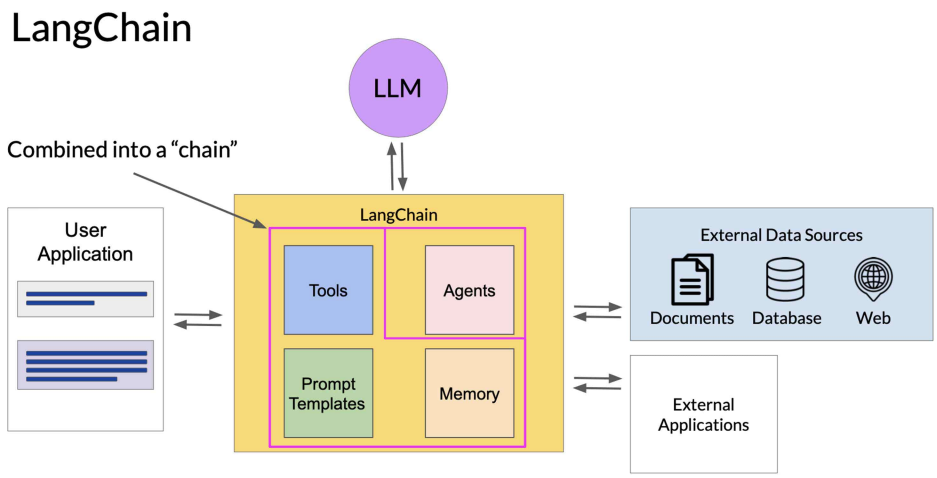
Building up the ReAct prompt



LangChain:

- LangChain framework 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.
  - The framework also includes pre-built tools that enable you to carry out a wide variety of tasks, including calls to external datasets and various APIs.
- Connecting a selection of these individual components together results in a chain. The creators of LangChain have developed a set of predefined chains that have been optimized for different

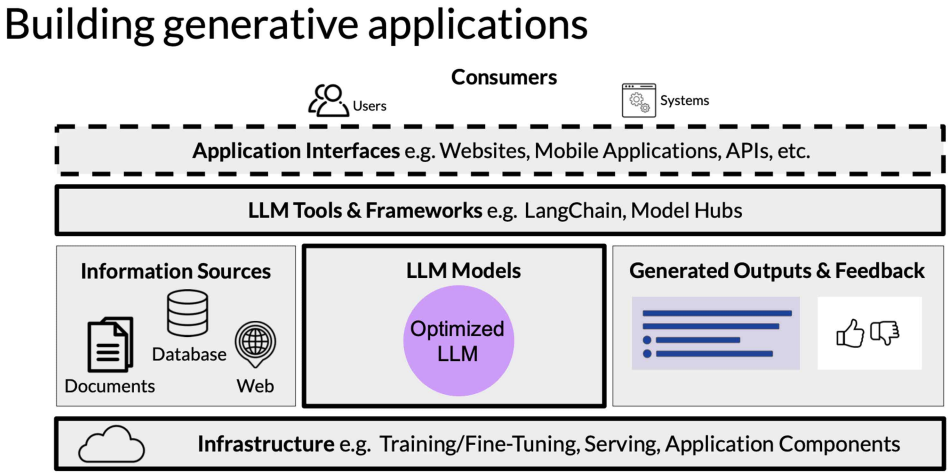
use cases, and you can use these off the shelf to quickly get your app up and running.



- LangChain defines another construct, known as an agent, that you can 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.

LLM application architectures

- For building LLM powered applications, you'll require several key components to create end-to-end solutions for your applications.



- Infrastructure:
  - This layer provides the compute, storage, and network to serve up your LLMs, as well as to host your application components.
  - You can make use of your on-premises infrastructure for this or have it provided for you via on-demand and pay-as-you-go Cloud services.

- LLM Models:

    - Large language models you want to use in your application. These could include foundation models, as well as the models you have adapted to your specific task.
    - The models are deployed on the appropriate infrastructure for your inference needs taking into account whether you need real-time or near-real-time interaction with the model.
- Information Sources/External data sources:
    - retrieve information from external sources, such as those discussed in the retrieval augmented generation section.
- Generated outputs and Feedback:
    - Your application will return the completions from your large language model to the user or consuming application.
    - Depending on your use case, you may need to implement a mechanism to 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 Frameworks:
    - you may need to use additional tools and frameworks for large language models that help you easily implement some of the techniques discussed.
    - As an example, you can use langChains built-in libraries to implement techniques like PAL, ReAct or chain-of-thought prompting.
    - You may also utilize model hubs which allow you to centrally manage and share models for use in applications
- Application Interfaces:
    - In the final layer, you typically have some type of user interface that the application will be consumed through, such as a website or a rest API. This layer is where you'll also include the security components required for interacting with your application.

- At a high level, this architecture stack represents the various components to consider as part of your generative AI applications. Your users, whether they are human end-users or other systems that access your application through its APIs, will interact with this entire stack.