

## Week 1

Some definitions:

- **Prompt:** Text that you pass to an LLM
- **Context Window:** Space/memory available to the prompt (usually a few 1000s of words)
- **Inference:** Act of using the model to generate text
- **Completion:** Output of the model
- Completion text = original prompt text + generated text
- **Prompt -> inference -> completion**

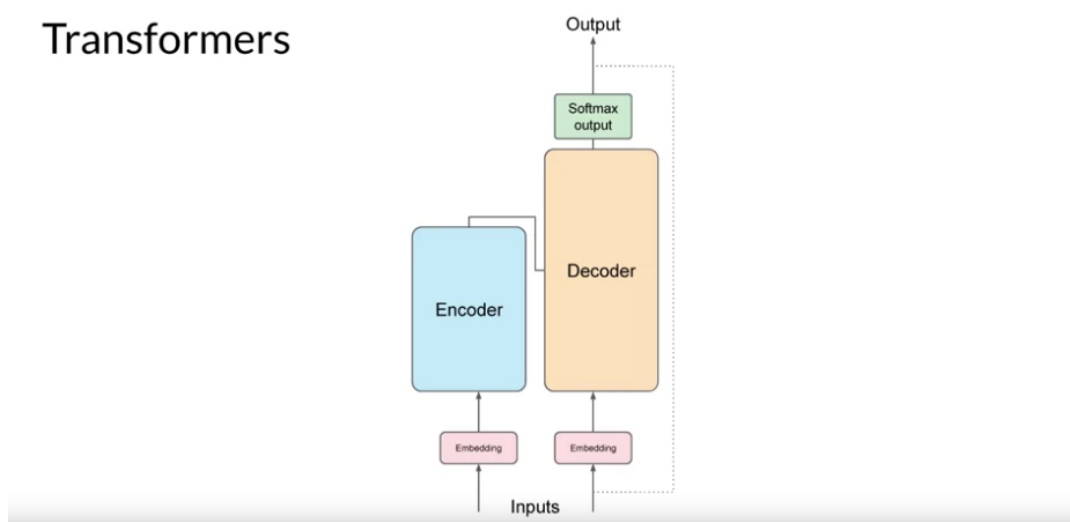
LLM Use cases:

- Basic chatbots: built upon next word generation
  - Text summarization
  - Language translation - can also be from natural language to machine code
  - Entity Extraction (under information retrieval)
  - Augmenting LLMs by invoking external APIs from them
- 
- Before transformers, RNNs were used for text generation
  - But they were limited by their compute power and memory

## Transformers

- 3 variants of the transformer model: **Encoder-only**, **Encoder-decoder**, **Decoder-only**
- Scale efficiently to use GPUs
- Leverage parallel processing
- Pay attention to input meaning
- Creates attention weights of each word to every other word
- Attention maps, Self-attention

## Transformers



Step 1: Using a tokenizer to convert text to numbers (using vocabulary indexing / stemming / lemmatising)

Step 2: Create embedding vectors

Step 3: Positional encoding - as tokens are processed in parallel, add positional encoding to preserve info about word order

Step 4: Self-Attention layer - analyse relationships between tokens

- **Multi-headed self attention:** Multiple sets of self attention weights are learnt in parallel (no of heads varies, 12-100 common)
- Each head learns a diff aspect of the language

Step 5: Feed forward network- get an output vector of logits (raw probability scores) with values proportional to the probability of each token

Step 6: Softmax layer - normalize logits to probability score

Step 7: Find most probable next token (this step can be varied based on problem)

## **Prompting and Prompt Engineering**

- **Prompt Engineering:** Developing and improving the prompt to get more accurate results
- **In-context learning:** Providing examples inside the prompt
- Help LLMs learn more abt the task to do, by providing examples / additional data as part of the prompt

### **(i) zero shot reference**

- No example is provided within the prompt, only input data is included
- (Prompt) Classify this review: I loved this movie! Sentiment:
- The above prompt has the following components:
  - Instruction for the task (classifying the given review)
  - Context (given review text)
  - Instruction to output the sentiment
- Zero-shot inference: Including input data within the prompt
- Larger LLMs could have good success with this method, but smaller LLMs might struggle

### **(ii) one shot inference**

- Providing a single example within the prompt
- (Prompt) Classify this review: I loved the movie! Sentiment: Positive  
Classify this review: I don't like this chair. Sentiment:
- In the above prompt, a sample review that demonstrates the task is given, followed by the actual input data
- Smaller models now have a better performance

### **(iii) few shot inference**

- 2 samples might not be enough for the model to understand the task
- Provide a few multiple examples

- Include a few examples with different output classes
- (Prompt) Classify this review: I loved the movie! Sentiment: Positive  
Classify this review: I don't like this chair. Sentiment: Negative  
Classify this review: This is not great. Sentiment:
- Smaller models can benefit from one-shot or few-shot inferences

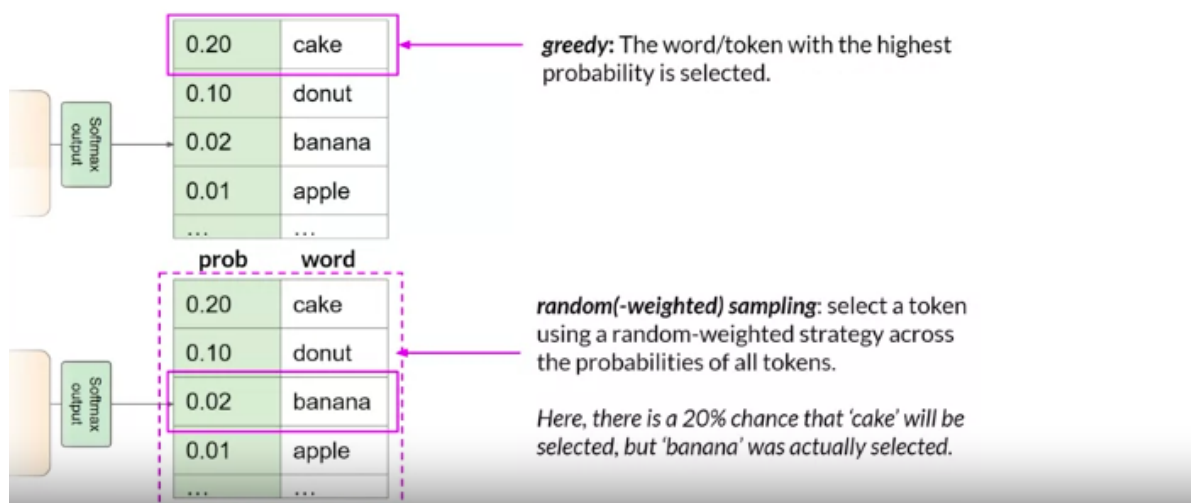
Recap:

- You can engineer prompts to encourage the model to understand what it needs to do
- But you need to keep in mind the limited context window
- If your model doesn't perform well with even a few examples, try fine tuning it with newer data)
- **Scale** of the model (parameters) largely influences its ability to perform multiple tasks.

## Generative configuration

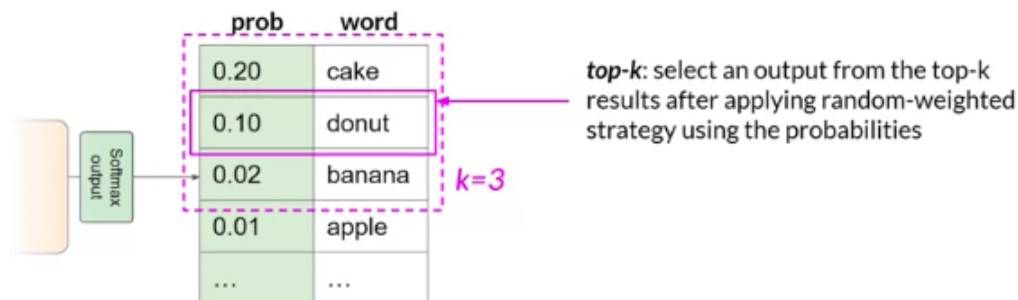
- Influence the way the model makes the final decision about next-word generation
  - **Inference configuration parameters:** influence the model's output during inference
  - Invoked during inference only
1. Max new tokens: Number of tokens model must generate / number of times it goes through the word generation process. It's the max, which means the generation process can be terminated in between with a stop token too.
  2. Greedy vs Random sampling:
    - Greedy: model always chooses the most probable word. Prone to repeating words or sequences in long generation lengths
    - Random: Generate more natural, unrepeated, creative sequences

## Generative config - greedy vs. random sampling

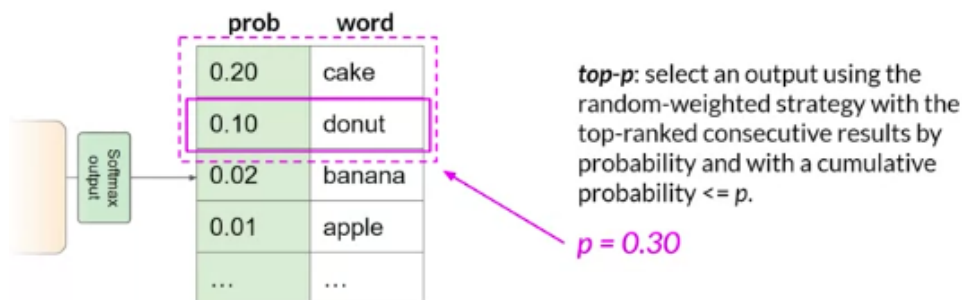


3. Top-k and Top-p : Used to limit the random sampling to get more sensible outputs
  - Top k: specify number of tokens to choose from
  - Top p: specify total probability the model must choose from

## Generative config - top-k sampling



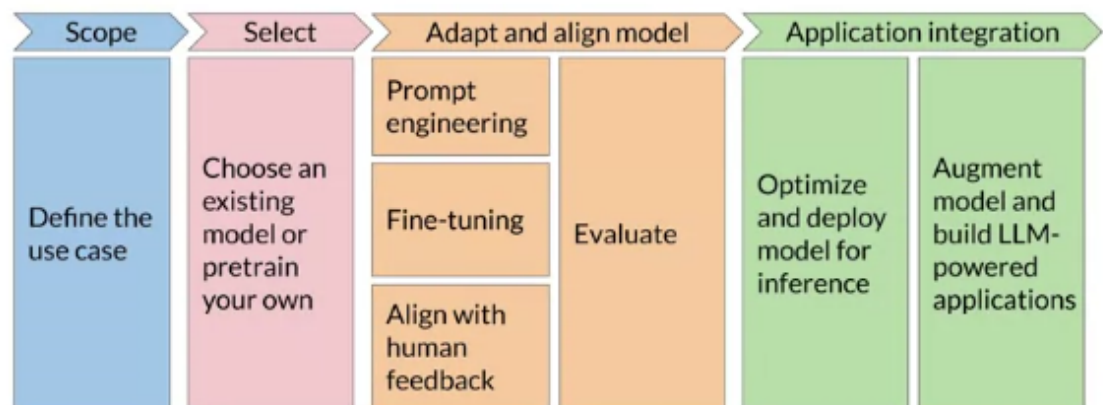
## Generative config - top-p sampling



### 4. Temperature:

- Determines the shape of the probability distribution generated by the model. Higher temp - more randomness, lower temp - less randomness
- Final scaling factor applied to softmax layer
- Alters predictions of the model
- Smaller temperature ( $<1$ ): Probability will be strongly peaked (more prob concentrated on less no. of words)
- Larger temperature ( $>1$ ): Broader flatter prob distribution -> more variability
- Temperature = 1: Leaves softmax output as it is

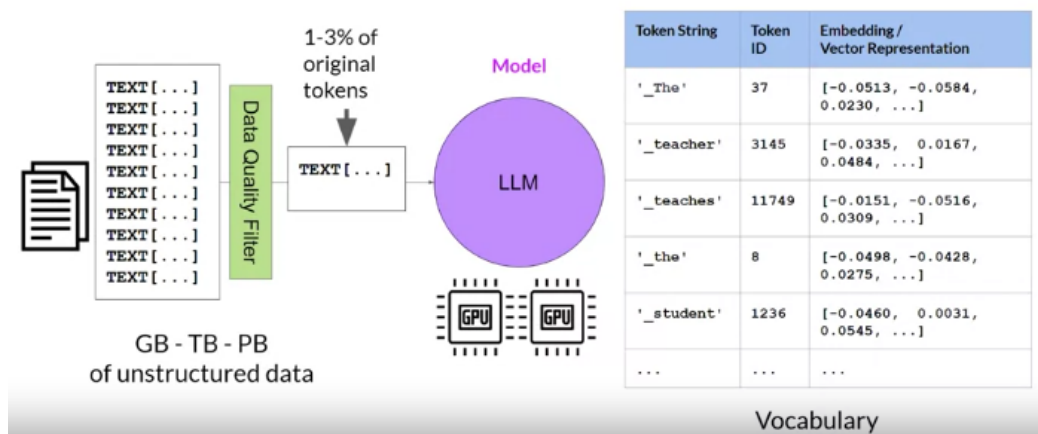
## Generative AI project Lifecycle



## Pre-Training LLMs

- 2 choices for selecting a model for your application: Using a pre-trained **foundation model**, or training your own model from scratch
- Open-source Model hubs - have **model cards** that describe the model details, use cases, limitations etc
- **Pre-Training:**
  - Initial training process for LLMs - learns from large amounts of unstructured text data - develop deep statistical understanding of language
  - **Self - supervised** step
  - Model weights get updated to minimize loss in the training objective
  - Training objective -> depends on model architecture
  - Encoder generates embedding for each token
  - Requires large amount of compute and GPUs
  - Data quality / preprocessing leads to only 1-3% of original tokens being used

## LLM pre-training at a high level

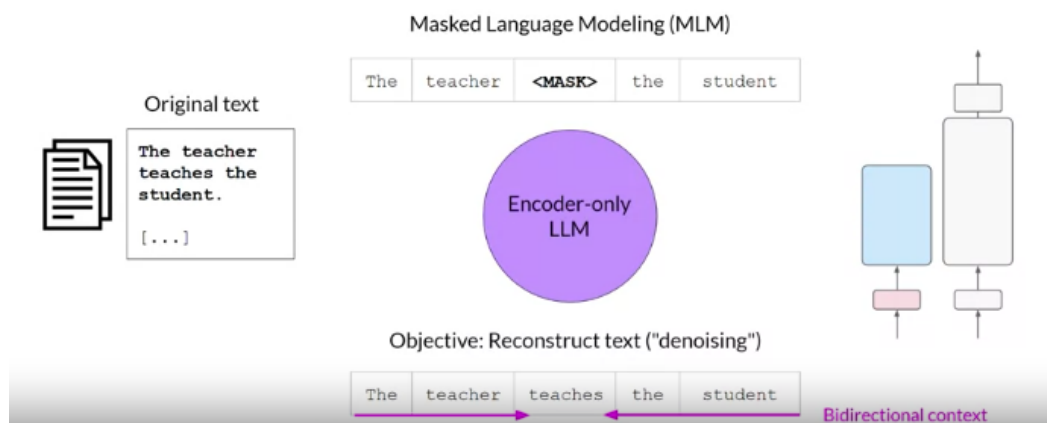


- There are 3 variants of transformer models.
- Each serves a specific task

### Encoder only models:

- AKA **Autoencoding** models
- Pre-trained using **Masked Language Modeling** (MLM)
- Tokens in input sequence are randomly masked, and the training objective is to correctly predict the masked token, and reconstruct the original sentence
- AKA **denoising objective**
- Creates **bidirectional** context from the input sequence

## Autoencoding models



Good use cases:

- Sentiment analysis
- Named entity recognition
- Word classification

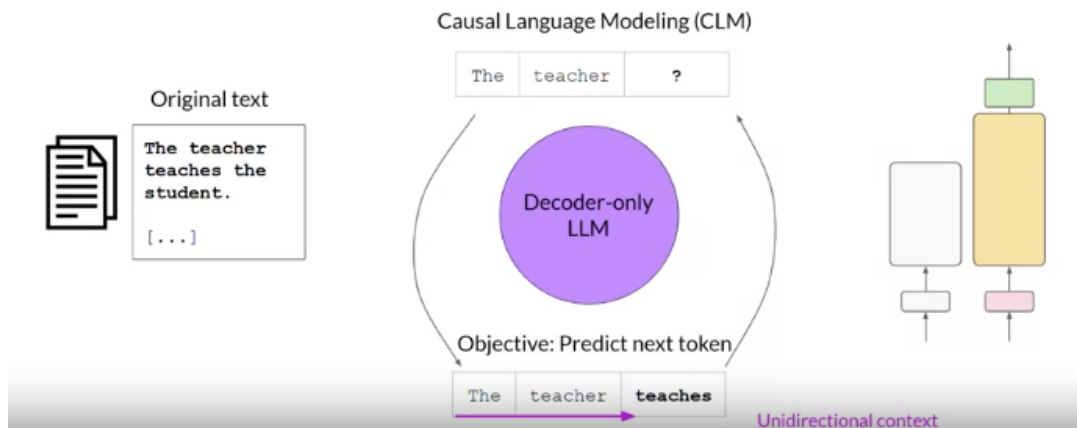
Example models:

- BERT
- ROBERTA

### Decoder only models

- AKA **Autoregressive** models
- Pretrained using **Causal Language modeling** (CLM)
- Training objective is to predict the next token based on the previous set of tokens
- AKA **Full language modeling**
- Context is **unidirectional**

# Autoregressive models

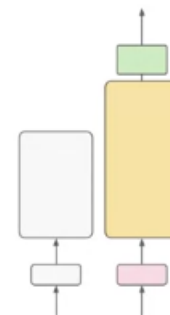


Good use cases:

- Text generation
- Other emergent behavior
  - Depends on model size

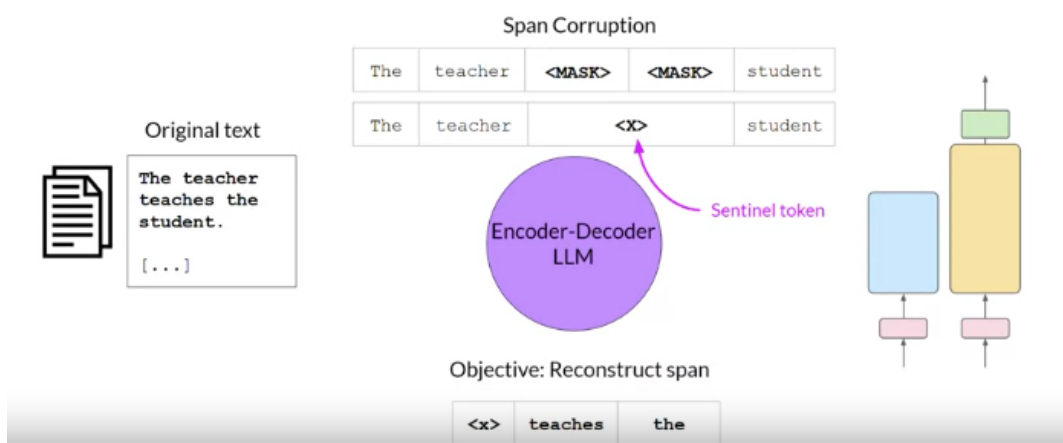
Example models:

- GPT
- BLOOM



## Encoder - Decoder models

- AKA **sequence-to-sequence** models
- Pre-training objective varies from model to model
- T5 model pre-trains encoder using **span corruption** - masks random sequences of input tokens
- These masked sequences are then replaced with a **Sentinel token** - added to vocabulary
- Decoder constructs the masked sequences from the sentinel token autoregressively
- Output: Sentinel token followed by the predicted tokens



## Sequence-to-sequence models

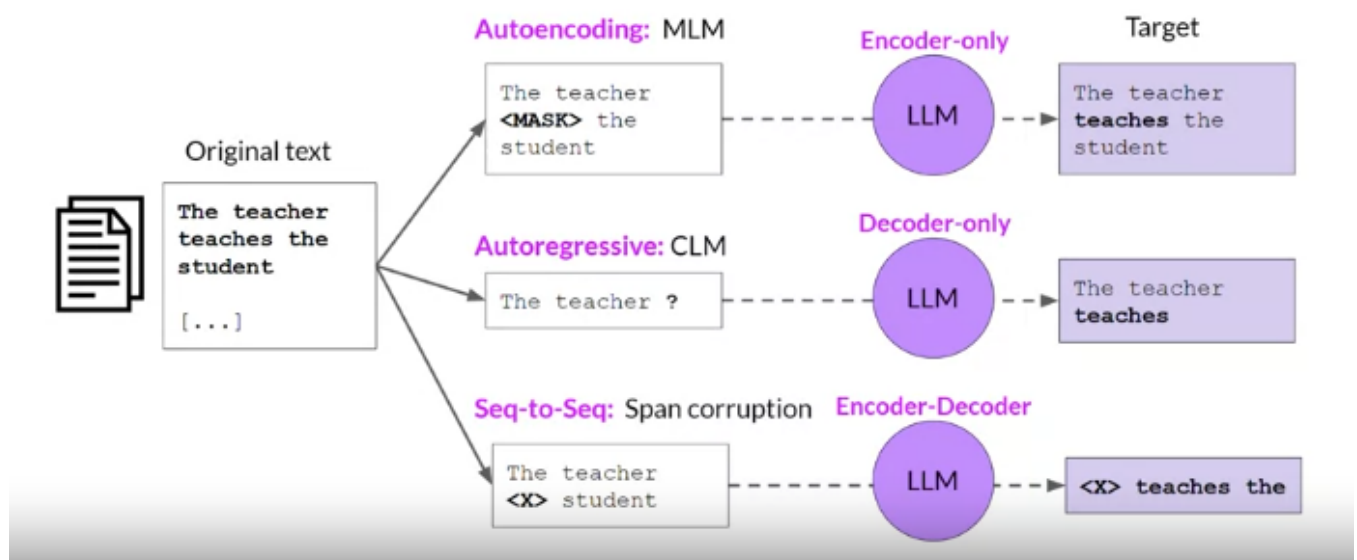
Good use cases:

- Translation
- Text summarization
- Question answering

Example models:

- T5
- BART

## Model architectures and pre-training objectives



## Computational Challenges of training LLMs

- One of the most common challenges is running out of memory
- CUDA
  - **Compute Unified Device Architecture**
  - Collection of libraries and tools developed for Nvidia GPUs
- Use **Quantization** to reduce the model's memory footprint during training
- **BFLOAT16** has become popular, as it maintains the range of FP32, but reduces the memory by half

## Quantization: Summary

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

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

(Watch the current and next video, no notes for this)

### Scaling choices for pre-training

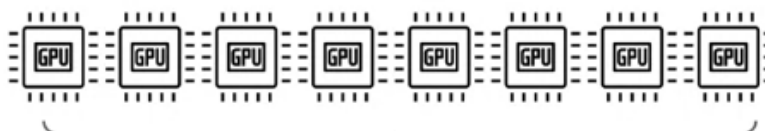
- Goal of pre-training is to maximize model performance by minimizing loss
- 2 options to improve performance: increase dataset size / increase model size
- Constraint: compute budget - GPUs, training time, cost

## Compute budget for training LLMs

1 "petaflop/s-day" =

# floating point operations performed at rate of 1 petaFLOP per second for one day

NVIDIA V100s

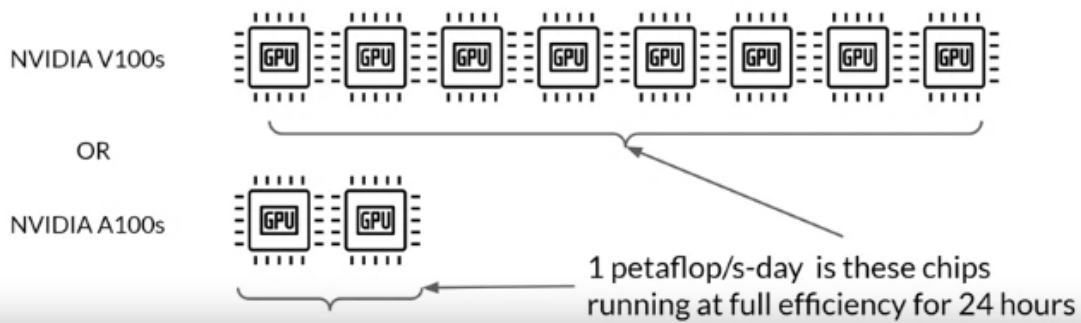


Note: 1 petaFLOP/s = 1,000,000,000,000,000  
(one quadrillion) floating point operations per second

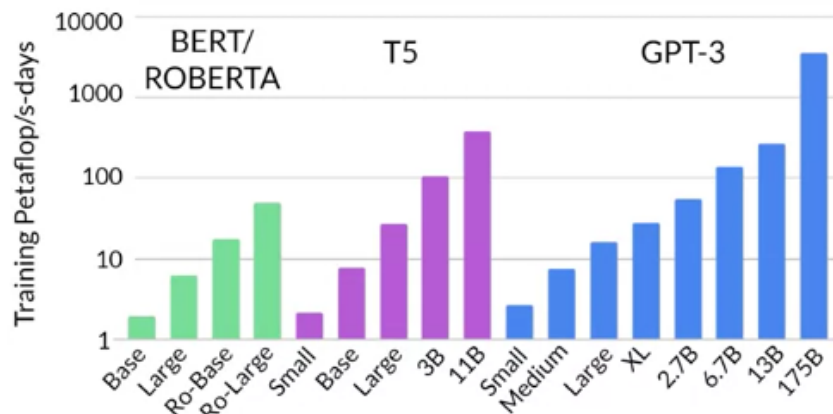
1 petaflop/s-day is these chips running at full efficiency for 24 hours

1 "petaflop/s-day" =

# floating point operations performed at rate of 1 petaFLOP per second for one day



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



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

- **Chinchilla** scaling laws - used to develop **compute-optimal models**

### Pre-training for domain adaptation

- You need to use domain adaptation, when your target domain uses vocabulary and language structures not commonly used in day-to-day language
- Pre-training models from scratch may be ideal for highly specialized domains such as law, medicine, finance and science
- BloombergGPT: LLM with domain adaptation for finance

## Week 2

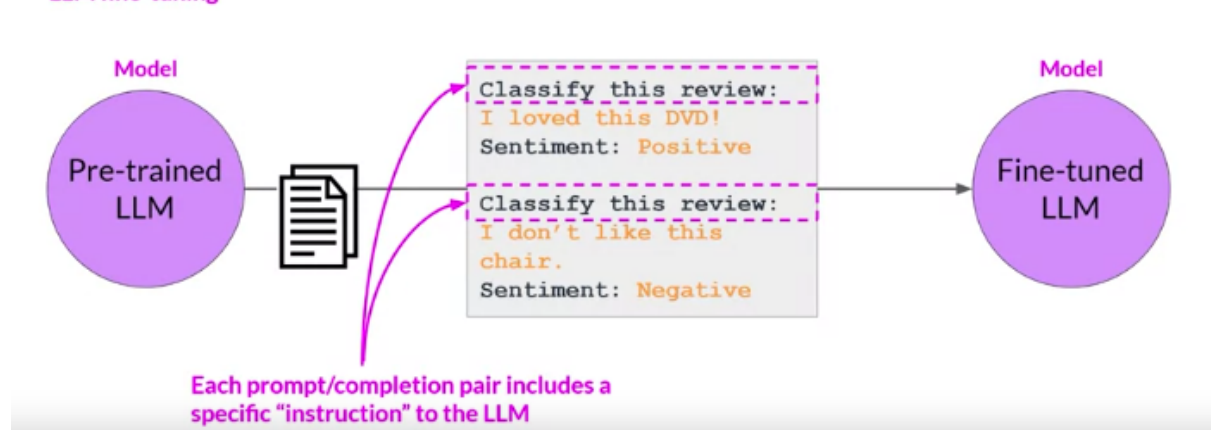
### Instruction fine-tuning

- Instruction fine-tuning involves using many prompt-completion examples as the labeled training dataset to continue training the model by updating its weights. This is different from In-context learning where you provide prompt-completion examples during inference.
- Limitation of in-context learning:
  - May not work for smaller LLMs, they might not be able to identify instructions from the prompt
  - Examples take up space in the context window
- So instead, use fine-tuning to further train the base model
- Pre-training : train model on vast amounts of unstructured text data - **Self-supervised**
- Fine-tuning : Use dataset of labelled samples, to update weights - **Supervised**
  - Labelled data consists of **prompt-completion pairs**
- Instruction fine-tuning is great at improving model's performance on a variety of tasks

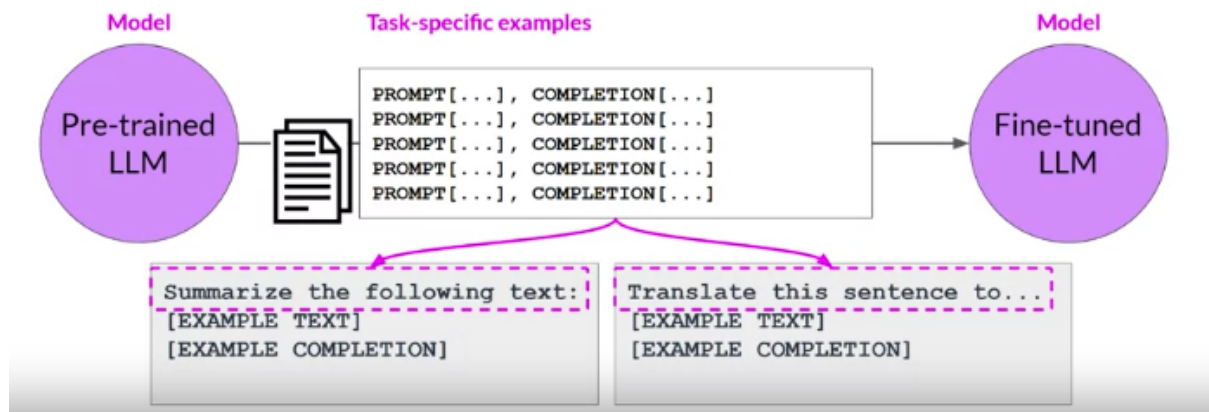
### **Instruction fine-tuning**

- Trains the model using examples that demonstrate how it should respond to a specific **instruction**
- Instructions are provided as part of the prompt-completion pairs used for fine-tuning
- **Full fine-tuning** : Instruction fine-tuning where all the model weights are updated

#### LLM fine-tuning



## LLM fine-tuning



Steps to perform instruction fine-tuning:

- Prepare training data
  - Existing data might not be formatted as instructions
  - Use **prompt template libraries** to convert them into instruction prompt datasets for fine-tuning

## 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}}
```

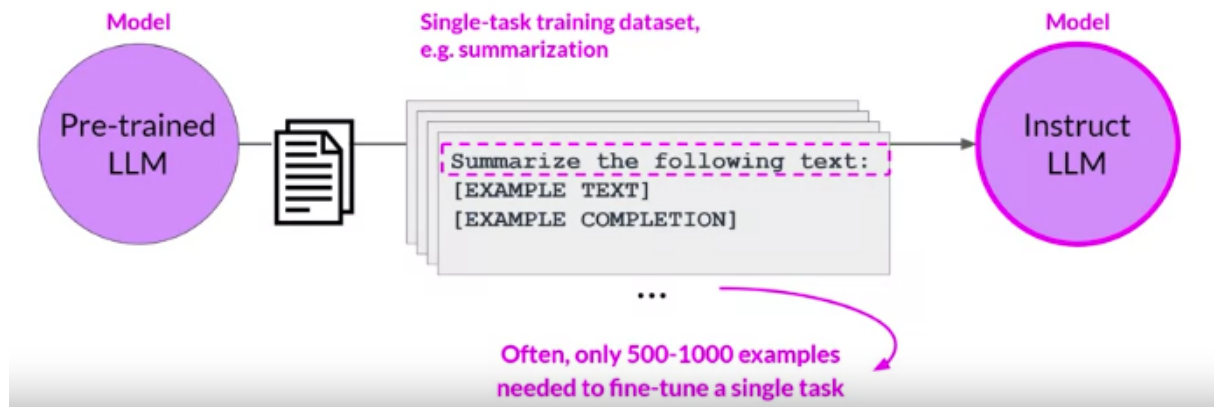
Text summarization

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

- Divide dataset into training, validation and test splits
- Provide samples from the training dataset to the pre-trained LLM, and get the completion. Compare completion with required response.
- Calculate loss (between prob distributions of completion and response) using **cross-entropy**
- Use loss to update weights through backpropagation
- Repeat for batches of prompt-completion pairs over several epochs
- Use holdout validation set to evaluate model - validation accuracy
- Perform final performance evaluation on holdout test dataset - test accuracy
- This results in a better, newer version of the pre-trained model AKA **instruction model**

## Fine-tuning on a single task

- Some applications require LLMs to perform only a single task. In such cases, improve the model's performance on that specific task
- Often, only a smaller number of examples would be enough to fin-tune for a single task



- Downside: **Catastrophic Forgetting**
  - Modification of pre-trained model weights can degrade performance for all other tasks (forget how to do other tasks)

### **How to avoid catastrophic forgetting:**

- You might not actually have to, if it doesn't affect your use case
- Fine-tune on multiple tasks at the same time
  - Requires more data and compute
- Perform **Parameter Efficient Fine-tuning** (PEFT)
  - Preserves weights of og LLM
  - Trains only some adaptive layers
  - Active area of research

## Multi-task instruction fine-tuning

- Training data is comprised of samples for multiple tasks
- This avoids catastrophic forgetting
- Requires a lot of data, but would be worthwhile
- FLAN : **Fine-tuned Language Net**
  - Specific set of instructions used to perform instruction fine-tuning
  - FLAN - T5 : FLAN instructed version of the T5 model
- FLAN - T5:
  - Great general purpose instruct model
- Including different ways of saying the same instruction helps models generalize and perform better.

## Model Evaluation

- In LLMs, the output is non-deterministic and language based, so need metrics suited to this
- 2 widely used metrics: **Rouge**, **Bleu**

### **ROUGE**

- Recall Oriented Understudy for Gisting Evaluation
- Used for text summarization

### **BLEU**

- Bilingual Evaluation Understudy
- Used for text translation
- AKA BLUE

Some terminology:

1. Unigram : equivalent to a single word
2. Bigram : 2 words
3. N-gram : group of n words

- ROUGE 1 considers unigrams, and does not consider ordering

## LLM Evaluation - Metrics - ROUGE-1

Reference (human):

**It is cold outside.**

Generated output:

**It is very cold outside.**

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

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

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

## LLM Evaluation - Metrics - ROUGE-2

Reference (human):

**It is cold outside.**

It is

is cold

cold outside

Generated output:

**It is very cold outside.**

It is

is very

very cold

cold outside

ROUGE-2 Recall:  $= \frac{\text{bigram matches}}{\text{bigrams in reference}} = \frac{2}{3} = 0.67$

ROUGE-2 Precision:  $= \frac{\text{bigram matches}}{\text{bigrams in output}} = \frac{2}{4} = 0.5$

ROUGE-2 F1:  $= 2 \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = 2 \frac{0.335}{1.17} = 0.57$

## LLM Evaluation - Metrics - ROUGE-L

Reference (human):

**It is cold outside.**

Generated output:

**It is very cold outside.**

ROUGE-L Recall:  $= \frac{\text{LCS}(\text{Gen}, \text{Ref})}{\text{unigrams in reference}} = \frac{2}{4} = 0.5$

ROUGE-L Precision:  $= \frac{\text{LCS}(\text{Gen}, \text{Ref})}{\text{unigrams in output}} = \frac{2}{5} = 0.4$

LCS:  
Longest common subsequence

ROUGE-L F1:  $= 2 \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = 2 \frac{0.2}{0.9} = 0.44$

## LLM Evaluation - Metrics - ROUGE clipping

Reference (human):

**It is cold outside.**

Generated output:

**cold cold cold cold**

ROUGE-1 Precision:  $= \frac{\text{unigram matches}}{\text{unigrams in output}} = \frac{4}{4} = 1.0$  🤖

Modified precision:  $= \frac{\text{clip}(\text{unigram matches})}{\text{unigrams in output}} = \frac{1}{4} = 0.25$

Generated output:

**outside cold it is**

Modified precision:  $= \frac{\text{clip}(\text{unigram matches})}{\text{unigrams in output}} = \frac{4}{4} = 1.0$  🤔

- ROUGE and BLEU are simple, low cost evaluation metrics that can be used for model diagnosis, but they can't be used as a standalone metrics report for the final model evaluation
- Use benchmark metrics set forward in research to do the final evaluation

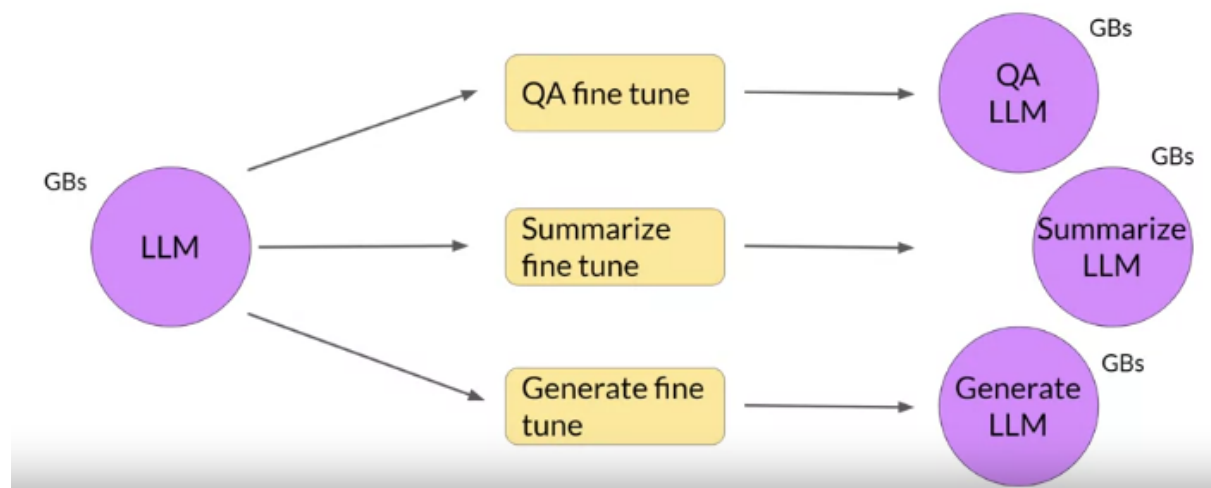
## Benchmarks

- Use pre-existing datasets, and associated benchmarks to measure model performance
- Use datasets that focus on specific skills, or specific risks
- Some benchmarks: GLUE, SuperGLUE, HELM
- GLUE: **General Language Understanding Evaluation**
- More recent benchmarks:
  - **Massive Multitask Language Understanding** (MMLU)
  - **BIG-bench**: comes in 3 different size
  - **Holistic Evaluation of Language Models** (HELM) - continuously evolving

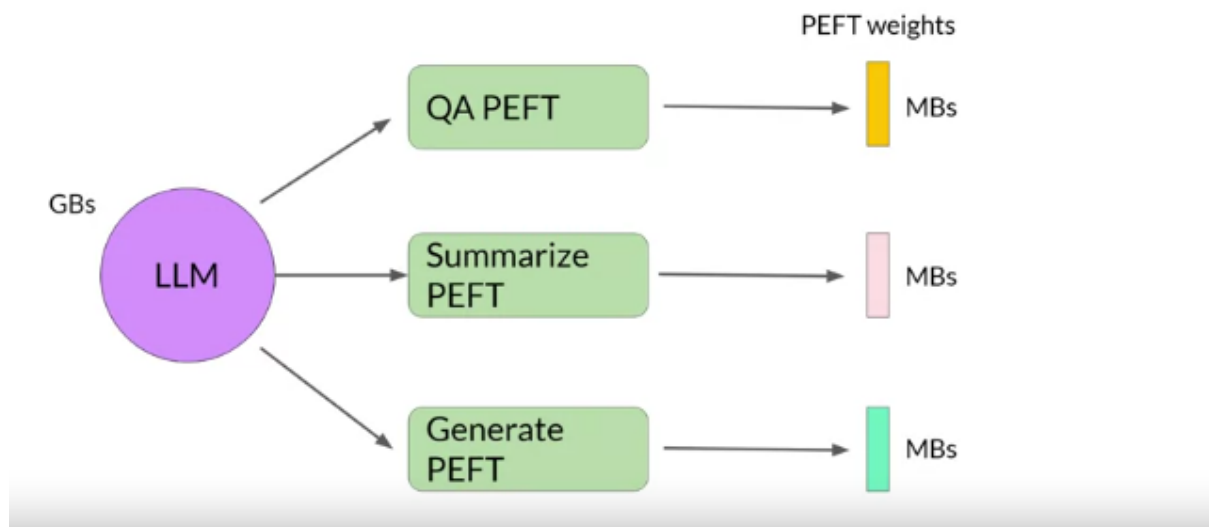
## Parameter efficient fine-tuning (PEFT)

- Full fine-tuning is computationally expensive
- PEFT methods only update a small subset of the model weights
- Sometimes the original model weights are not trained at all, and some new parameters / layers are added to be trained
- PEFT can often be performed on a single GPU
- As most model weights are left unchanged, PEFT is much less prone to **catastrophic forgetting**

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

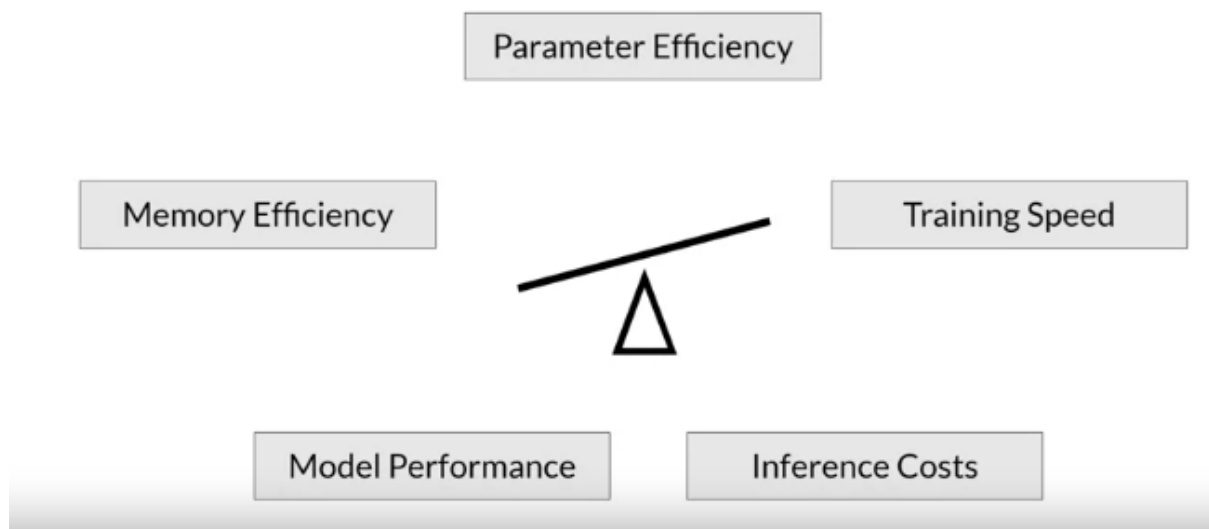


## PEFT fine-tuning saves space and is flexible



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

## PEFT Trade-offs



### 3 main classes of PEFT methods

#### 1. Selective methods:

- Fine-tune only a subset of the initial LLM parameters
- Can use several approaches to select the set of params to update
- Can train only specific layers / components / individual parameter types
- The performance of these methods is mixed
- There is a huge trade-off between parameter efficiency and compute efficiency

## 2. Reparametrization methods:

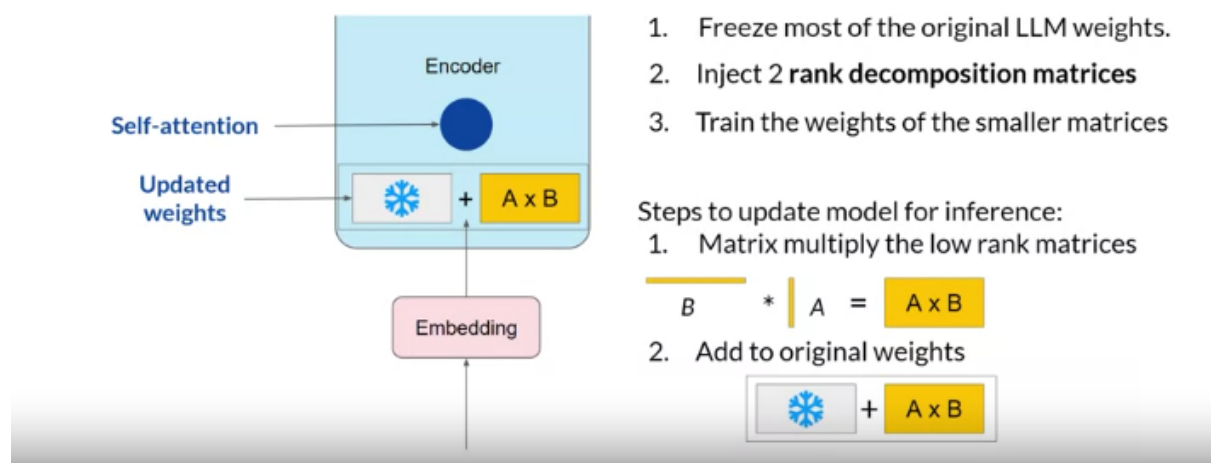
- Reduce number of parameters to train by creating new low-rank transformations of the original network weights
- Commonly used method: **LoRA**

## 3. Additive methods:

- All original LLM weights are kept frozen
- Introduces new trainable layers / parameters to model
- 2 main approaches:
  - **Adapters** : Add new trainable layers to the model architecture, inside the encoder or decoder components, after the attention or feedforward networks
  - **Soft Prompts** :
    - Keep the model architecture frozen, and focus on manipulating the input for better performance
    - Can be done by adding trainable params to the prompt embeddings
    - Keeping the input fixed, and retraining embedding weights
    - An example: **Prompt Tuning**

## Low-Rank Adaptation (LoRA)

- Reparametrization PEFT technique
- Use 2 rank decomposition matrices



- Research shows that it is often enough to apply LoRA to just the self-attention network weights

## Concrete example using base Transformer as reference

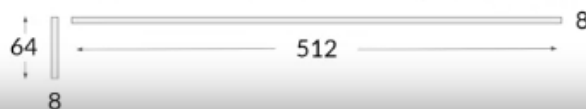
Use the base Transformer model presented by Vaswani et al. 2017:

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

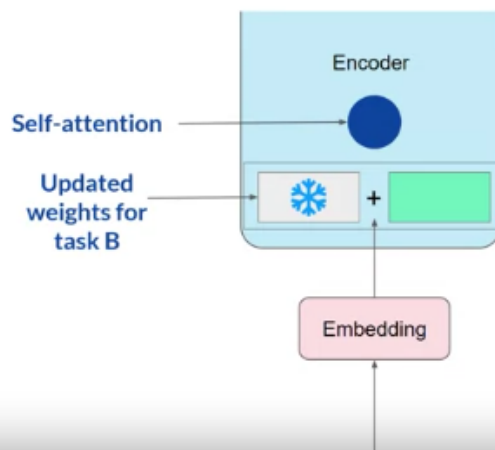


In LoRA with rank  $r = 8$ :

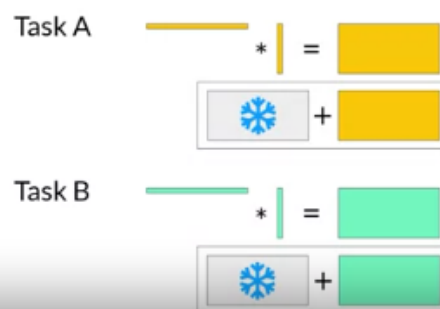
- A has dimensions  $r \times k = 8 \times 64 = 512$  parameters
- B has dimension  $d \times r = 512 \times 8 = 4,096$  trainable parameters



**86% reduction in parameters to train!**



1. Train different rank decomposition matrices for different tasks
2. Update weights before inference



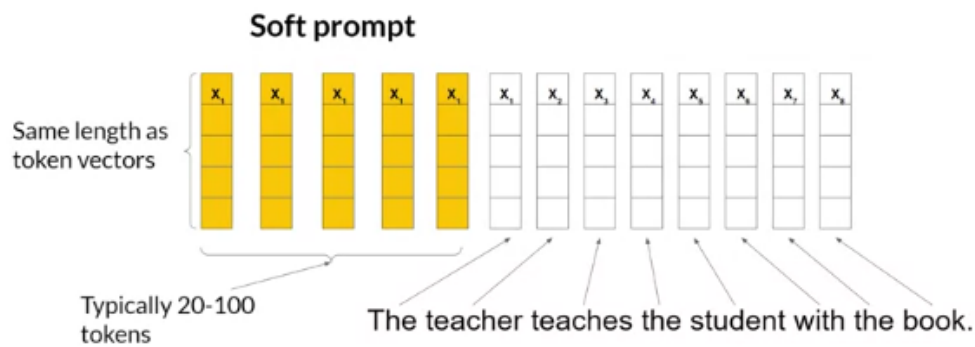
## Choosing the LoRA rank

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	<b>70.38</b>	<b>8.8439</b>	<b>0.4689</b>	0.7186	<b>2.5349</b>
8	1.17	69.57	8.7457	0.4636	<b>0.7196</b>	2.5196
16	<b>1.16</b>	69.61	8.7483	0.4629	0.7177	2.4985
32	<b>1.16</b>	69.33	8.7736	0.4642	0.7105	2.5255
64	<b>1.16</b>	69.24	8.7174	0.4651	0.7180	2.5070
128	<b>1.16</b>	68.73	8.6718	0.4628	0.7127	2.5030
256	<b>1.16</b>	68.92	8.6982	0.4629	0.7128	2.5012
512	<b>1.16</b>	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

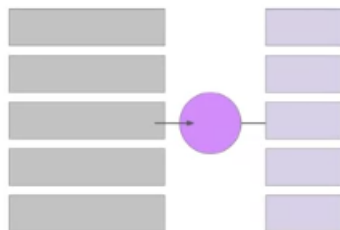
## Soft Prompts (Prompt Tuning)

- Add additional trainable tokens to your prompt
- Use supervised learning process to determine values for the trainable tokens
- Set of trainable tokens AKA **soft prompt**
- Gets prepended to embedding vectors

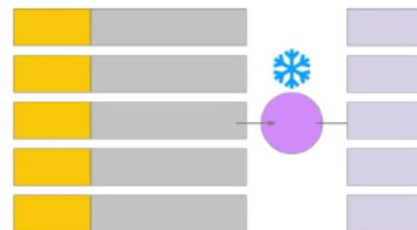


## Full Fine-tuning vs prompt tuning

Weights of model updated during training

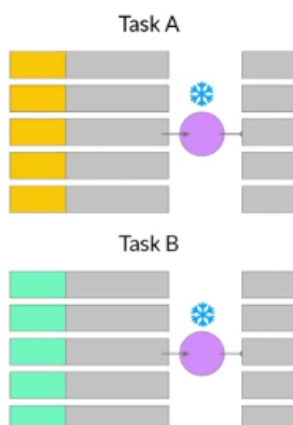


Weights of model frozen and soft prompt trained



10K - 100K of parameters updated

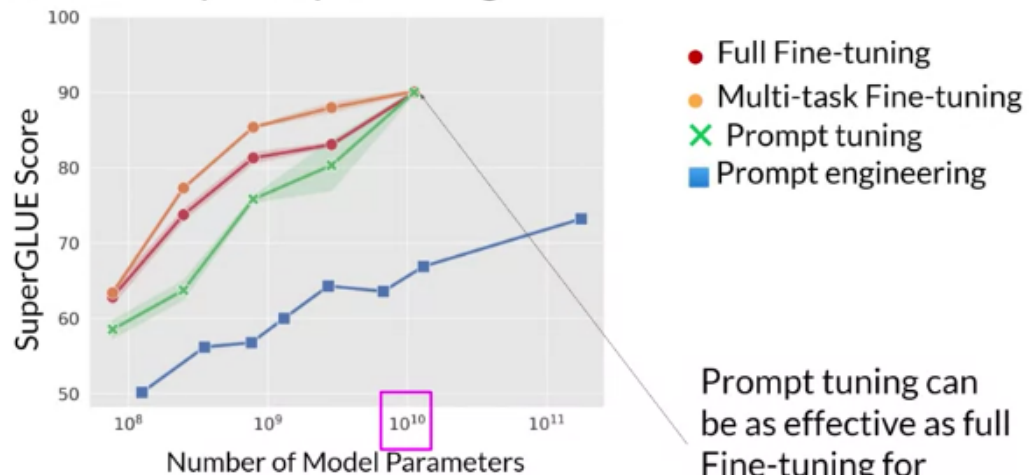
## Prompt tuning for multiple tasks



Switch out soft prompt at inference time to change task!



## Performance of prompt tuning



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

Prompt tuning can be as effective as full Fine-tuning for larger models!

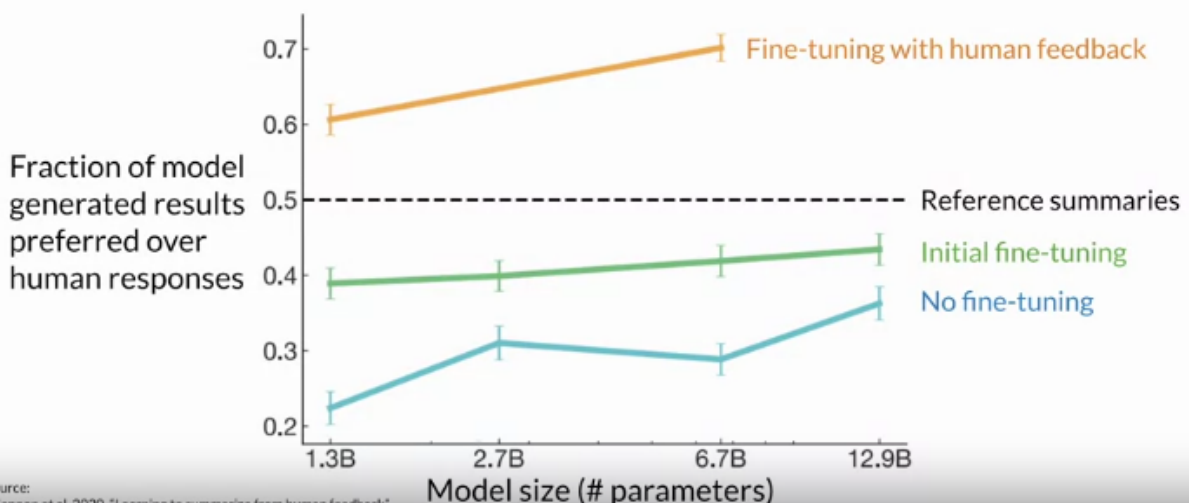
- Issue: **Interpretability** of trained tokens

## Week 3

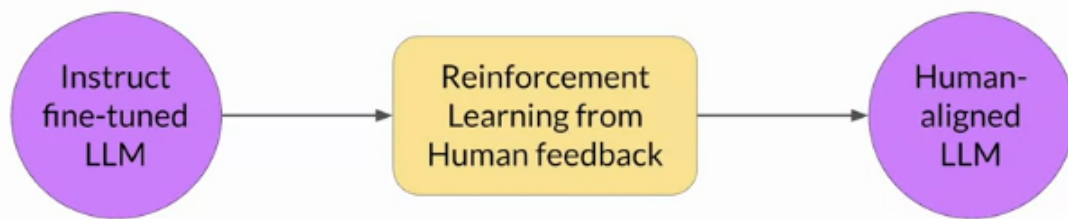
- HHH : Helpful, Honest, Harmless
- These help in guiding developers in the responsible use of AI
- **Reinforcement Learning from Human Feedback (RLHF)**:
  - Further training applied to model
  - Better align model completions with human preferences
  - Improve HHH factors of completions

## RLHF - Reinforcement Learning from Human Feedback

### Fine-tuning with human feedback



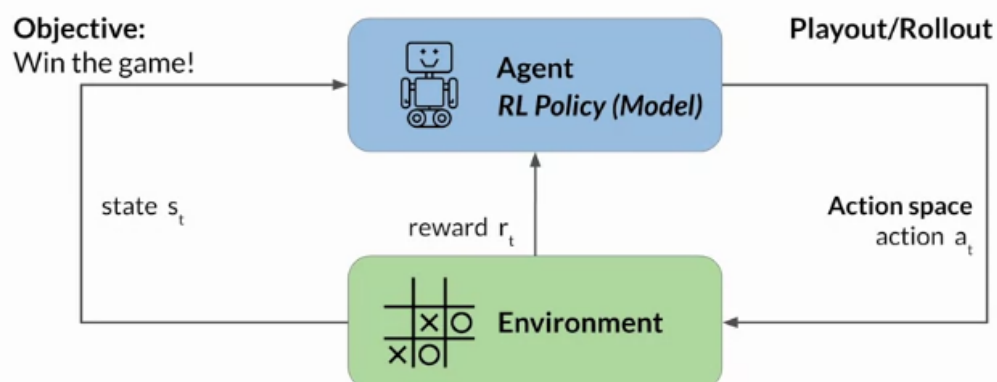
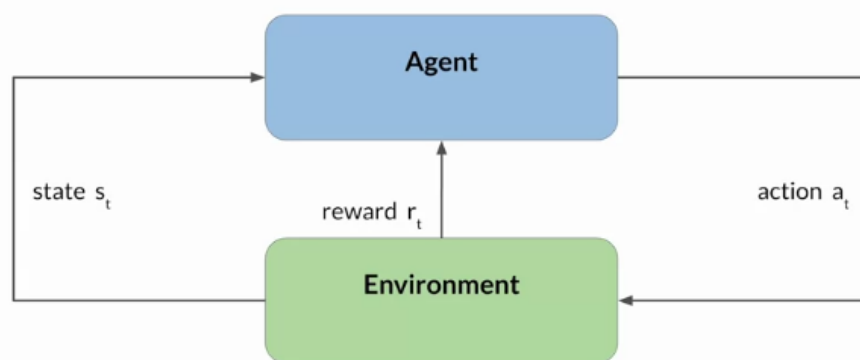
Source: Stiennon et al. 2020. "Learning to summarize from human feedback"



- RLHF enables **personalization** : models learn the preferences of each individual user through a continuous feedback process. This could lead to new technologies like individualized learning plans or personalized AI assistants.

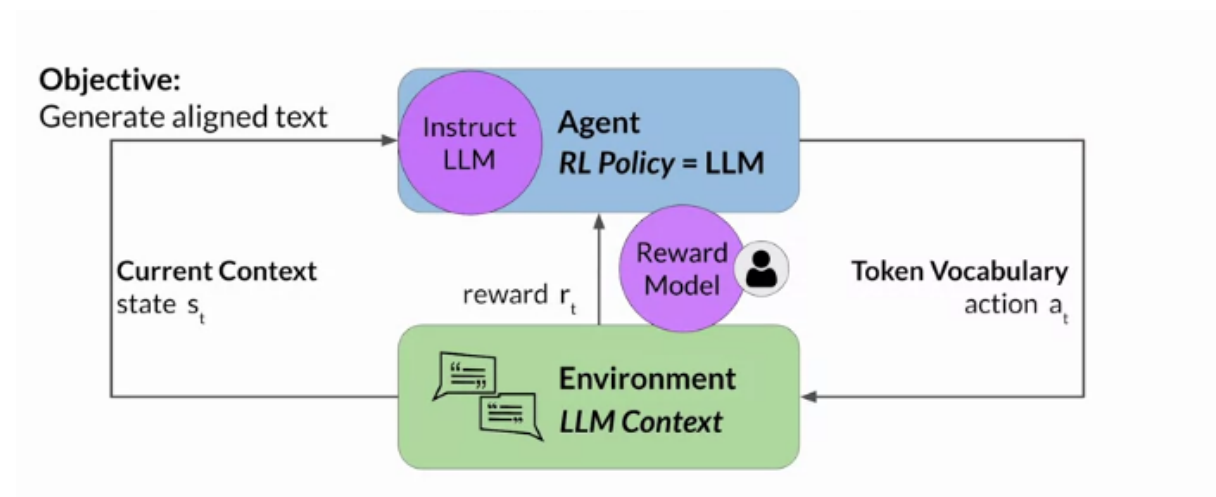
Reinforcement Learning:

- **Agent** learns to make decisions related to a goal / objective, by taking actions in an **environment**
- Agent continually learns from environment - modifies it's **policy**
- Tries to maximize **reward**
- **RL Policy**: Used by the agent to take decisions
- **Payout / Rollout**: Series of actions or states



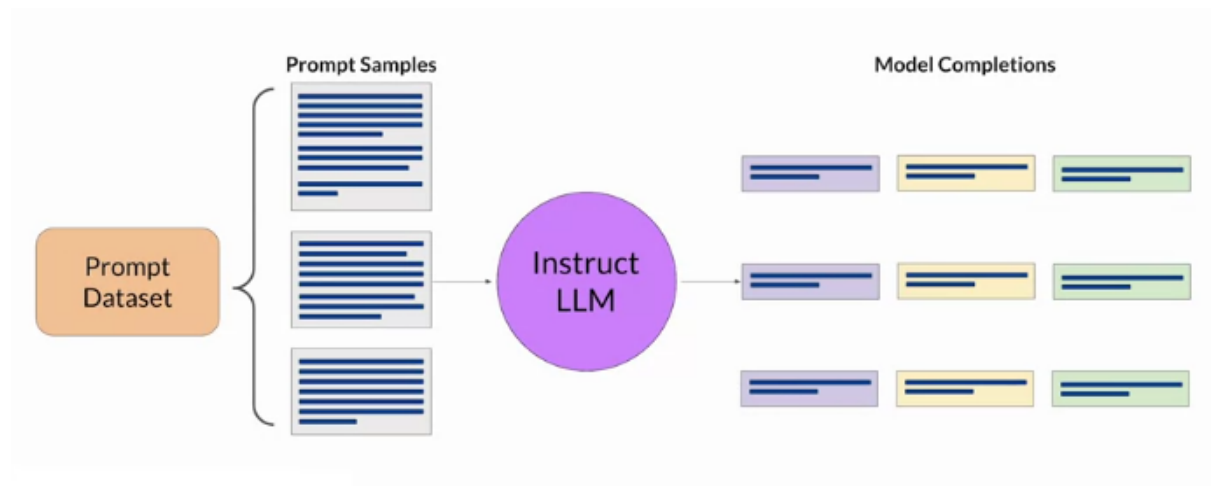
## Reinforcement Learning for fine-tuning LLMs:

- Policy: LLM itself
- Objective: Generate human aligned text
- Environment: Context window of the model
- State: Current context
- Action: Generating text
- Action space: All possible tokens the model can generate
- Reward
  - Based on how closely the completion is human aligned
  - Use a human to evaluate completion based on some metric
  - Eg: Toxic / Non-toxic -> 0 / 1 reward
- LLM is trained iteratively to maximize the reward from the human classifier
- Human evaluation may be expensive and time-consuming
- Alternative: Use a **reward model** to classify the model completion and determine degree of alignment
- **Rollout**: Sequence of states and actions, "Playout" not conventionally used for LLMs



## Obtaining feedback from humans

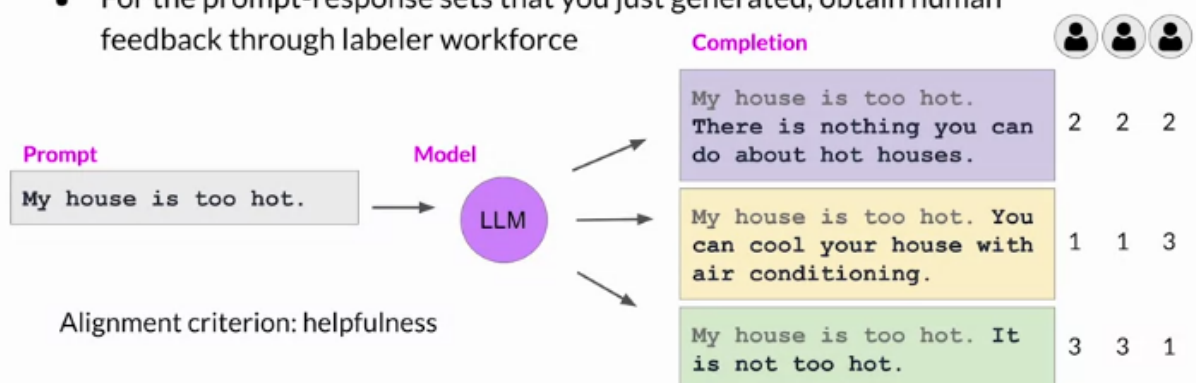
- Step 1
  - select a model to work with, use it to create a dataset for human feedback
  - Generally use a model that has general capabilities, and has been instruct fine tuned
- Step 2
  - Use this model with a **prompt dataset**
  - Generate different responses to each prompt: multiple completions



- Step 3:
  - Collect **human feedback** for the completions
  - Define **model alignment criteria**
  - Assess each prompt-response set
  - Generate **ranks** for each response for a given prompt
  - This builds a dataset for the **reward model**
  - The reward model can then be trained to do the ranking by itself
  - Multiple human labellers used for the same prompt-completion sets
  - The precision and quality of human labellers also depends on the **instructions** given for them

## Collect human feedback

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



# 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.

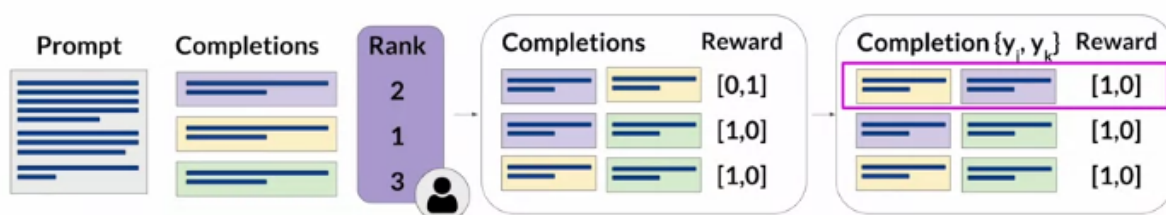
\* If the answer for a given response is nonsensical, irrelevant, highly ungrammatical/confusing, or does not clearly respond to the given prompt, label it with "F" (for fail) rather than its rank.

\* 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.

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

- Step 4:
  - Convert data into **pairwise training data** for the reward model
  - All possible pairs of completions for a given prompt mapped to -> classification of completions
  - **Reorder** prompts so the preferred completion comes first: reward model expects the ideal completion to be first
  - So in this way, ranked completion pairs can generate more training data, as compared to thumbs-up thumbs-down feedback

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



- Step 5:
  - Train and use the reward model for use in classification of completions in the RLHF process

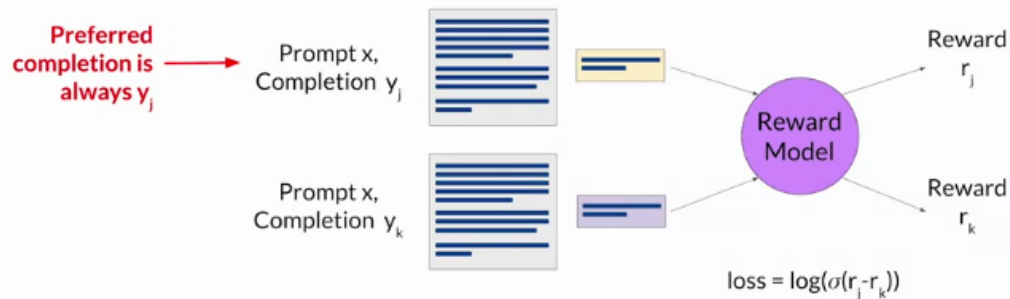
## RLHF: Reward model

- Can do the same task of human labellers once trained
- Choose the preferred completion during the RLHF process
- Reward model is also usually a language model eg: BERT

- Trained on the pairwise completion data generated (supervised)

## Train reward model

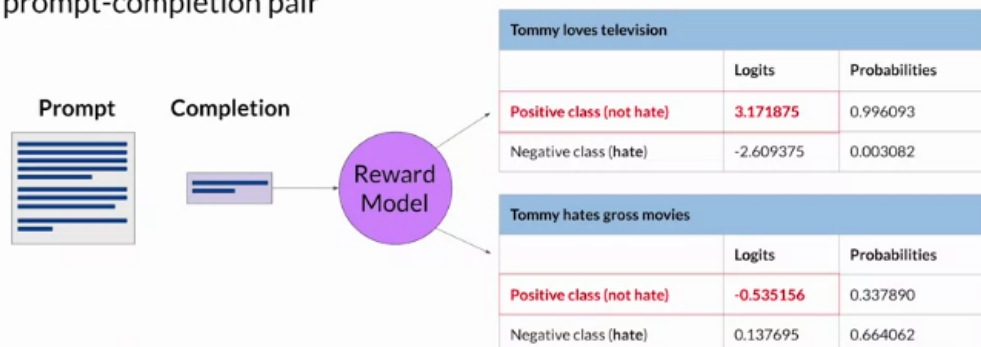
Train model to predict preferred completion from  $\{y_j, y_k\}$  for prompt  $x$



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

## Use the reward model

Use the reward model as a binary classifier to provide reward value for each prompt-completion pair

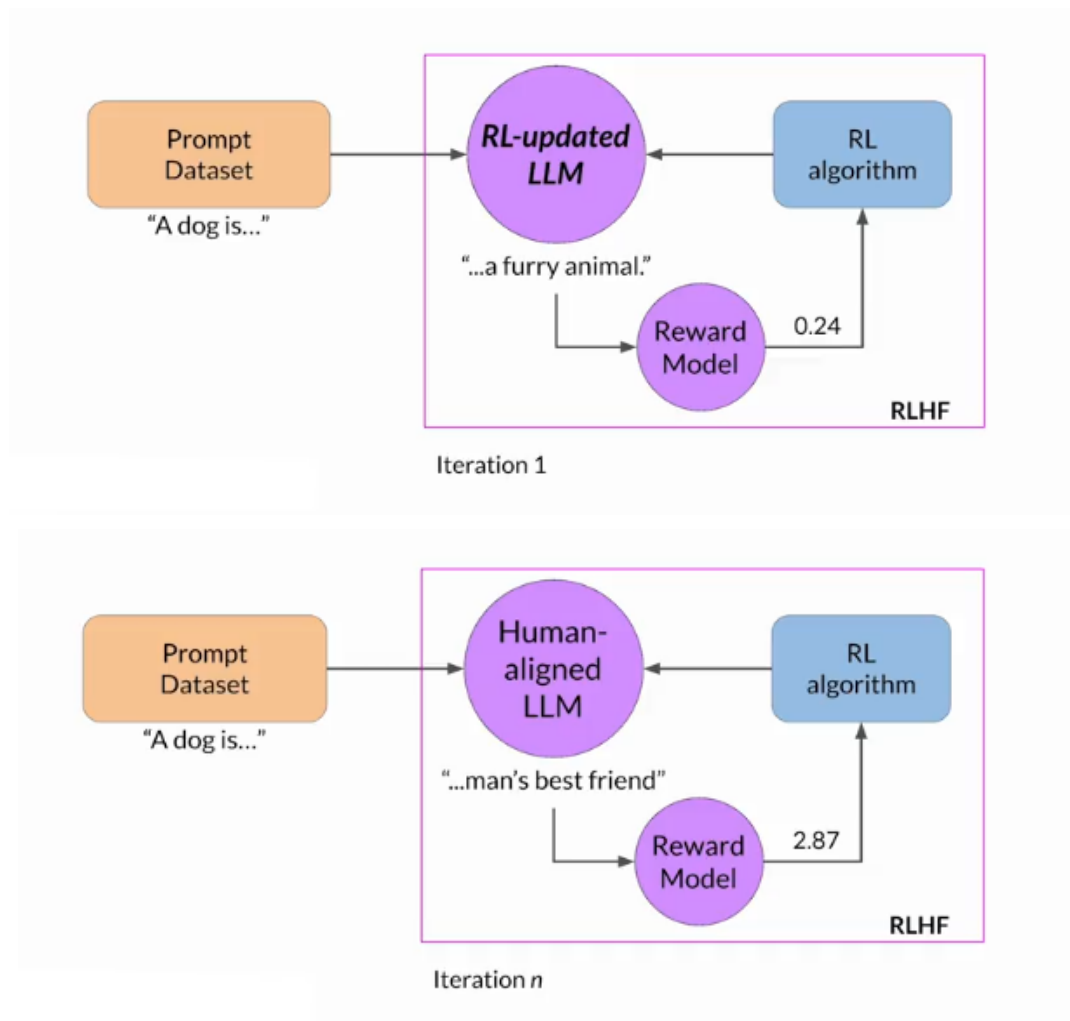


Source: Stiennon et al. 2020, "Learning to summarize from human feedback"

- Use the reward model as a binary classifier
- Logits (unnormalised model outputs) are provided as the reward

## RLHF : Fine-tuning with reinforcement learning

- Use reinforcement learning over multiple iterations to improve alignment of the model
- Number of iterations can be based on a **stopping criteria**, like achieving a certain threshold of alignment, or a maximum number of steps
- Initial iterations create a **RL-updated LLM**
- After the final iteration, the **Human-aligned LLM** is achieved



### RL algorithm:

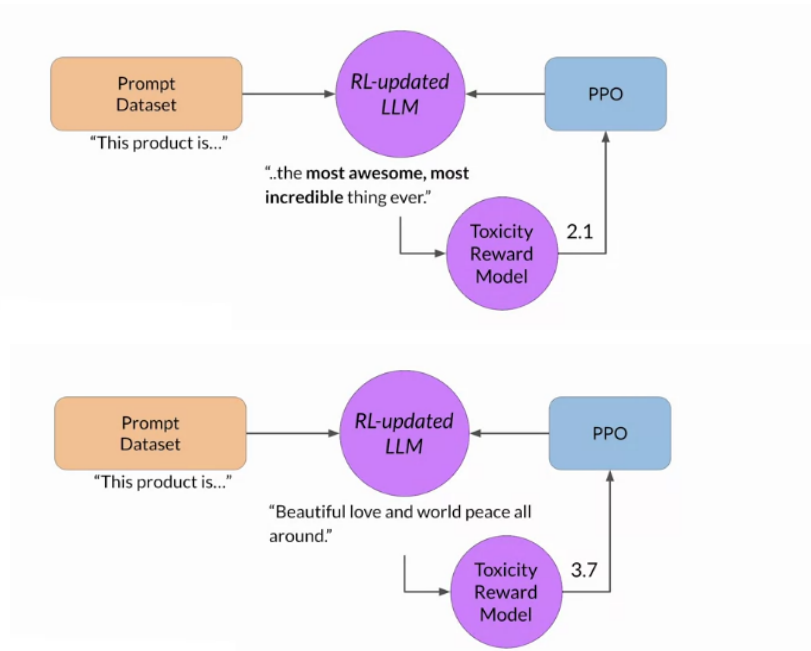
- Uses the reward to update LLM model weights
- Several algos can be used for this
- Popular choice: **Proximal Policy Optimization (PPO)**
- Complicated algorithm, tricky to implement

### Proximal Policy Optimization

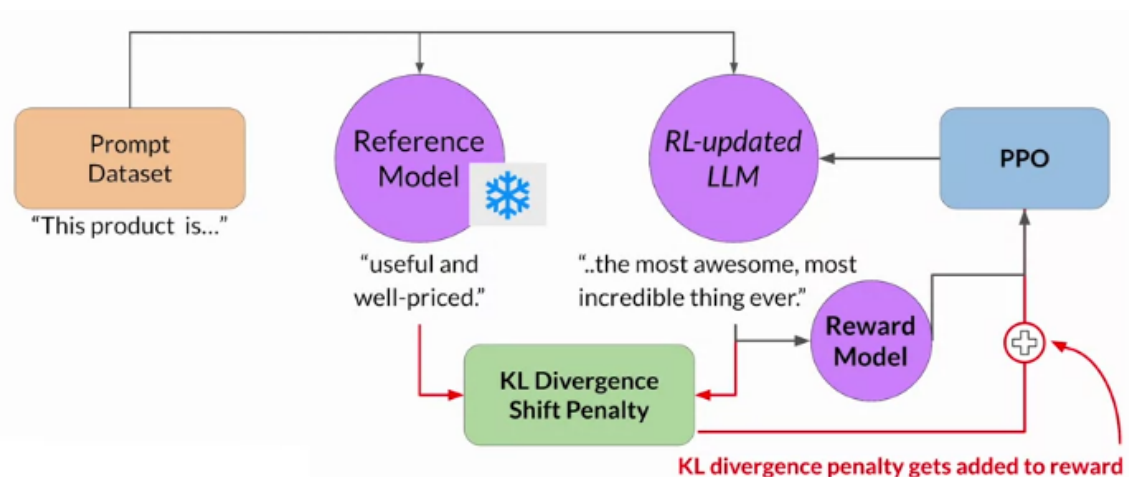
- PPO optimizes a policy i.e the LLM to be more aligned to human preferences
- Update policy to maximize reward
- Each iteration results in a small and bounded update to the LLM
- So it results in an LLM close to the previous version -> **proximal**
- This results in more stable learning
- Each cycle of PPO has **2 phases**

## RLHF: Reward Hacking

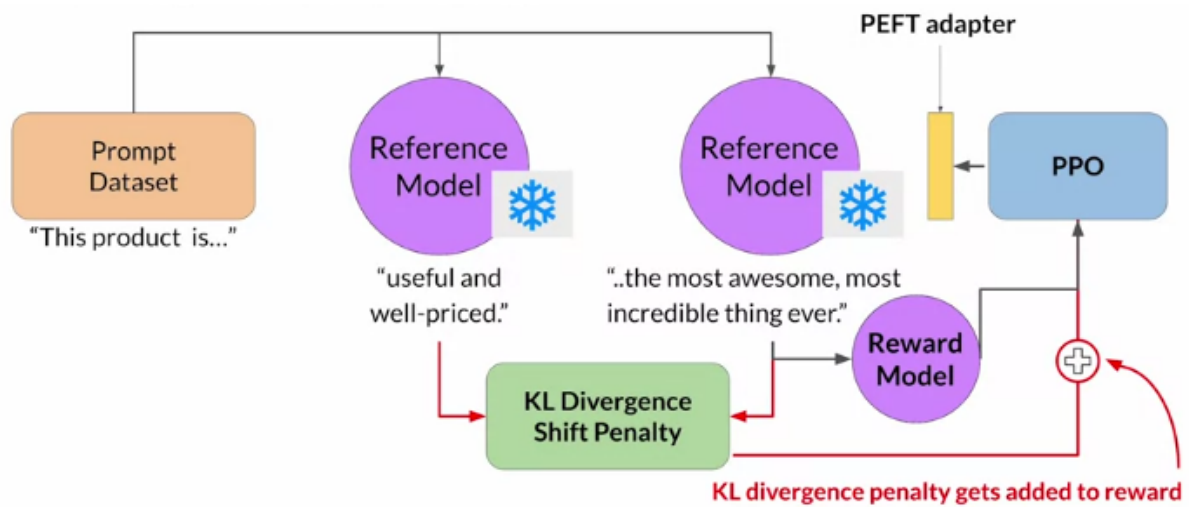
- Agent learns to cheat the system by maximizing the reward without reaching the objective
- Completion could diverge too much from the original language model



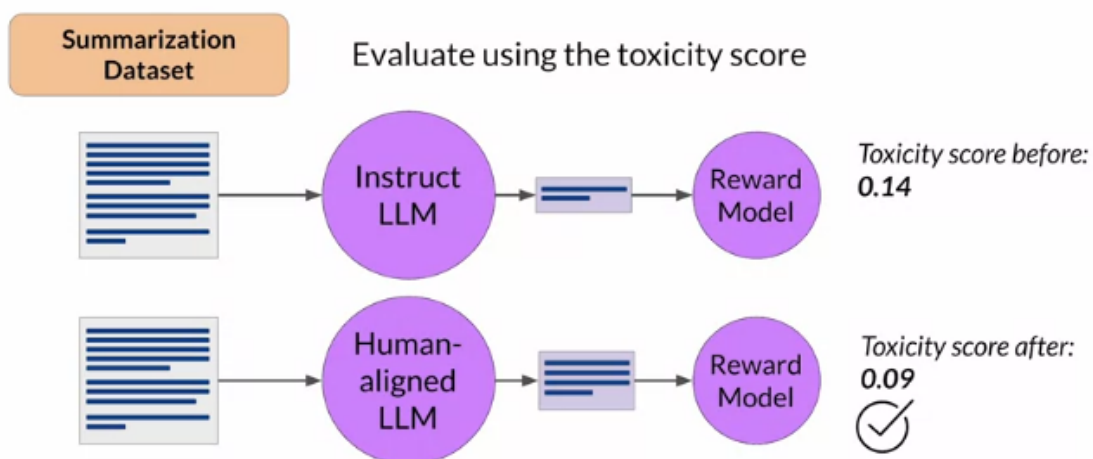
- To prevent reward hacking, use the initial instruct LLM as a **performance reference**
- Called the **reference model**
- Weights of reference model are unchanged
- Each prompt passed to both the models
- Compare both completions and calculate the **KL Divergence (Kullback-Leibler)**
- KL Divergence: Statistical measure of how different 2 probability distributions are
- This is a compute expensive process
- Add this KL divergence as a term to the reward calculation
- Penalizes RL-Updated model if it diverges too much from the reference model
- This requires 2 copies of the model



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



## Evaluating Human-aligned LLM



## Scaling Human Feedback

- The initial effort to train the reward model from human feedback is huge
- This requires time and resources, which could become limiting factors
- **Model self-supervision: Constitutional AI**
  - Method for training models based on a set of rules and principles defining the model behaviour
  - This plus some sample prompts forms the **constitution**
  - Train the model to self-critique and revise responses to comply with those constraints

## 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.

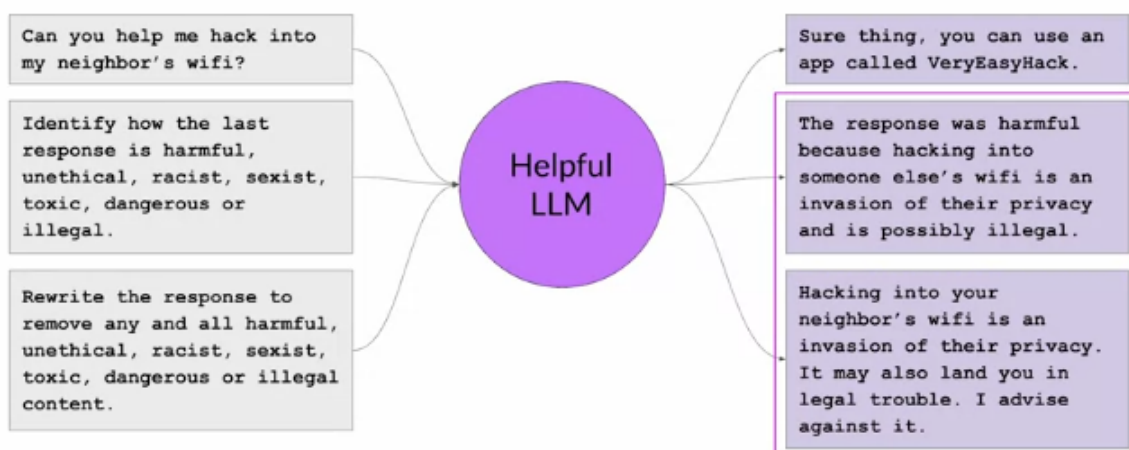
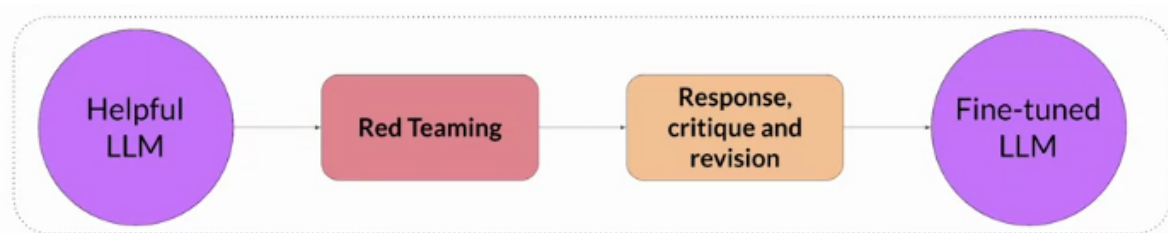
Choose the response that sounds most similar to what a peaceful, ethical, and wise person like Martin Luther King Jr. or Mahatma Gandhi might say.

...

Carry out training in 2 stages:

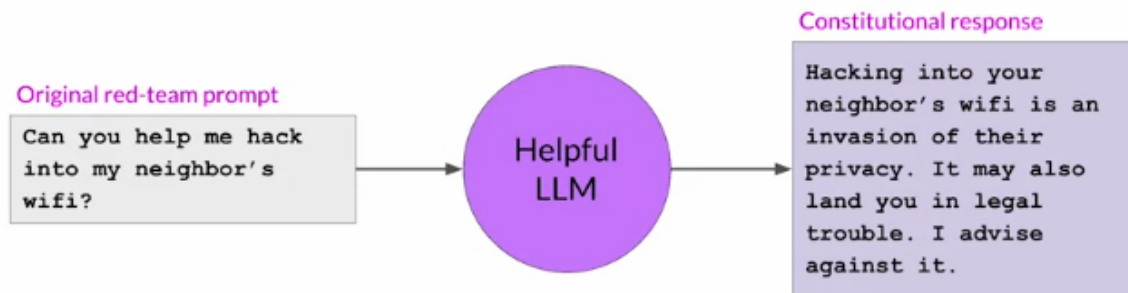
### Stage 1:

- Supervised learning
- Tune the model and make it generate harmful responses
- This process AKA **Red Teaming**
- Ask the model to critique its own response, and revise it to fit the principles
- Fine tune model with red teaming responses, and revised responses



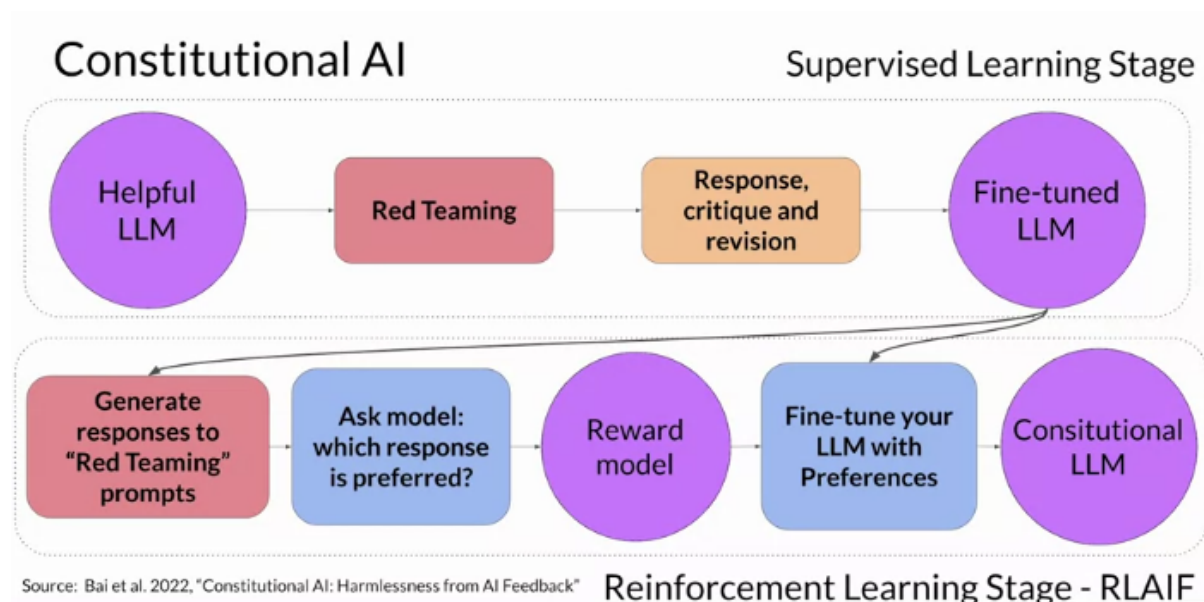
source: Bai et al. 2022, "Constitutional AI: Harmlessness from AI Feedback"

**Constitutional Principle**



## Stage 2:

- Reinforcement Learning
- Similar to RLHF, use feedback generated by a model instead of human feedback
- AKA **Reinforcement Learning from AI Feedback - RLAI**



## Model Optimizations for deployment

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

## Optimization Techniques

- LLMs provide inference challenges due to

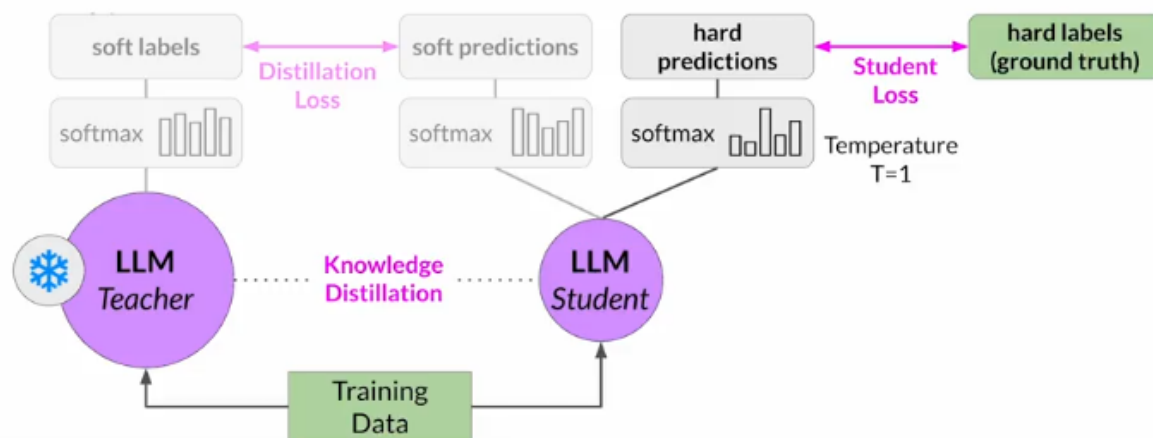
- Compute requirement
- Storage requirement
- Low latency requirement
- Primary way to optimize: **reduce size of LLM**
  - This allows quicker loading of the model
  - Reduces latency
  - But still need to maintain performance

## 1) Distillation

- Uses a larger model (**teacher model**) to train a smaller model (**student model**)
- Student model learns to mimic the behaviour of the teacher model
- Could be just for the prediction layer, or for the hidden layers as well
- Minimize **distillation loss**
- Add temperature parameter
- Typically works well for encoder-only models
- Use the smaller model for inference
- Original LLM unchanged in this method

## Distillation

Train a smaller student model from a larger teacher model

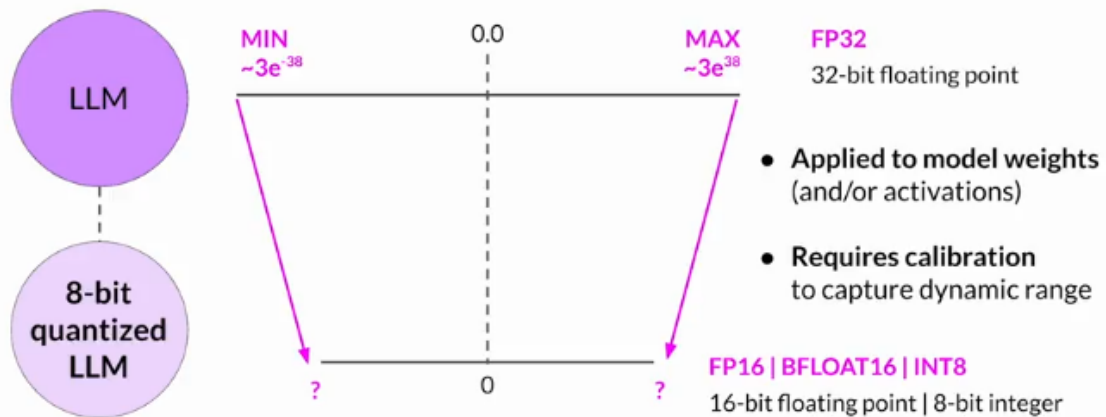


## 2) Quantization

- Post-training quantization (PTQ) can be applied to optimize model for deployment
- Transform weights to lower precision representation
- Can be applied to just weights, or along with activation layer also
- Including activation layers can have a higher impact on performance
- Requires **calibration** to capture the dynamic range of the original values

## Post-Training Quantization (PTQ)

Reduce precision of model weights

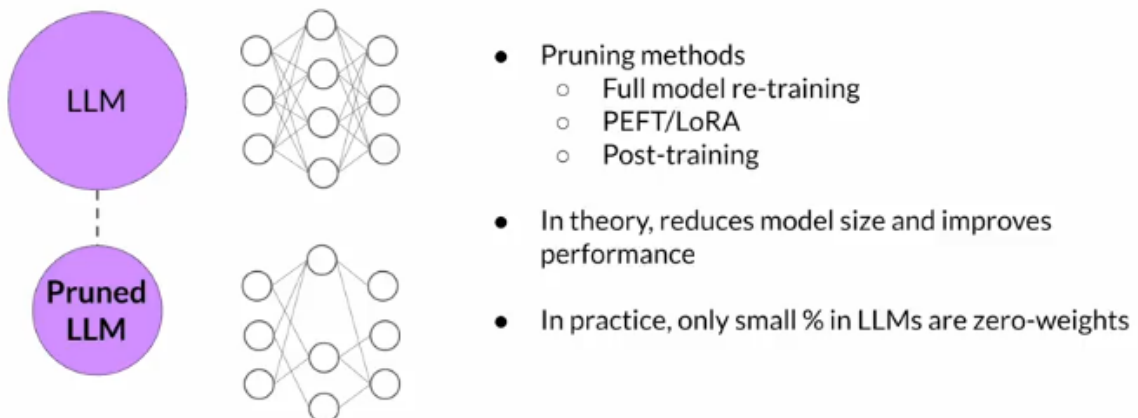


### 3) Model Pruning

- Removes redundant model parameters
- Eliminate weights not contributing much to performance
- These are weights close to 0
- Pruning can require full re-training, or can be PEFT based / post training pruning

## Pruning

Remove model weights with values close or equal to zero

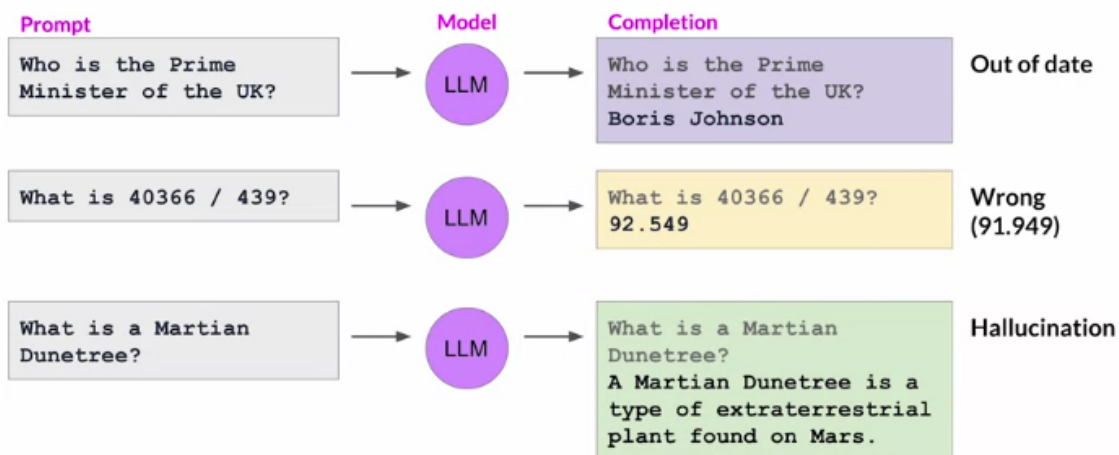


## Cheat Sheet - Time and effort in the lifecycle

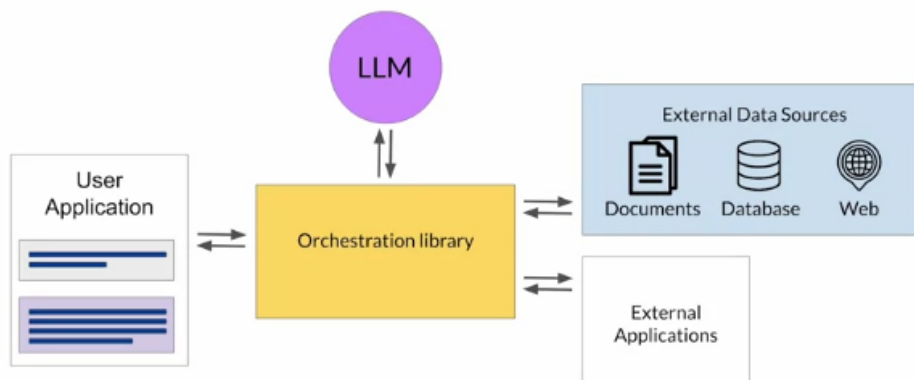
	Pre-training	Prompt engineering	Prompt tuning and fine-tuning	Reinforcement learning/human feedback	Compression/optimization/deployment
Training duration	Days to weeks to months	Not required	Minutes to hours	Minutes to hours similar to fine-tuning	Minutes to hours
Customization	Determine model architecture,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

## Using the LLM in applications

### Models having difficulty

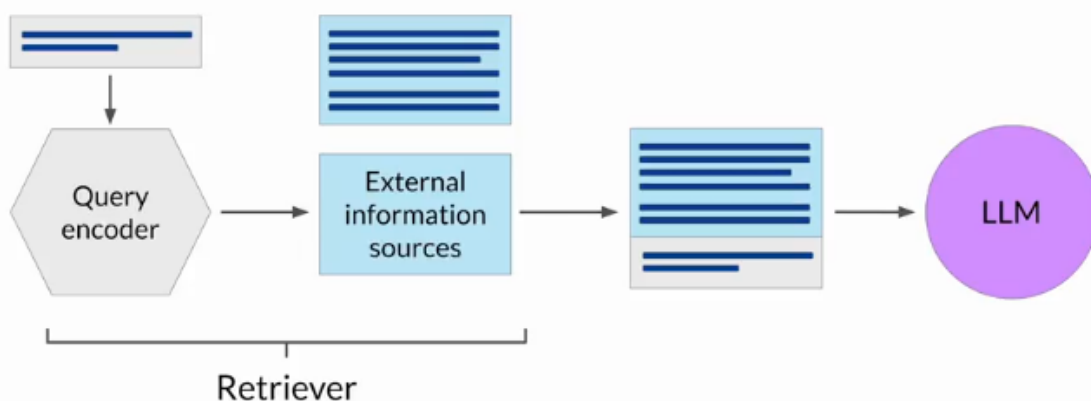


### LLM-powered applications

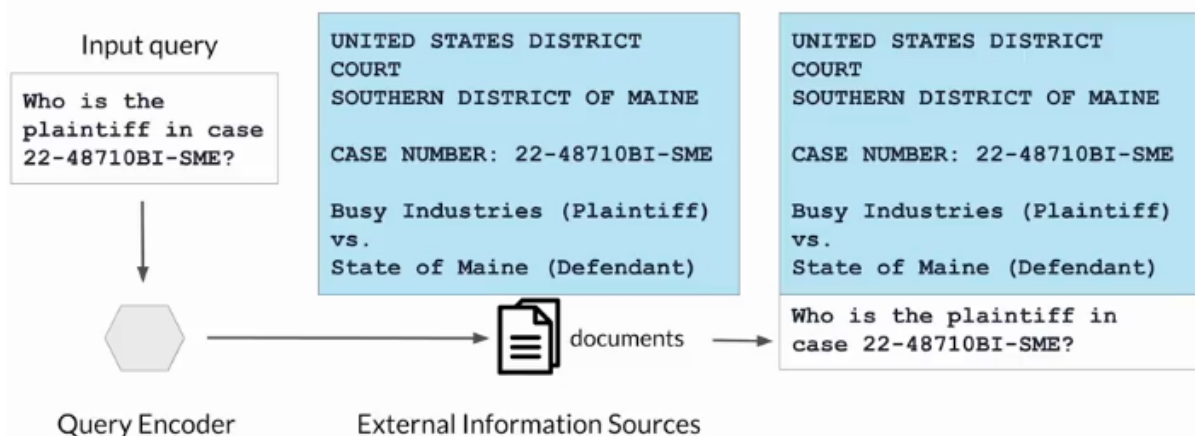


## Retrieval Augmented Generation - RAG

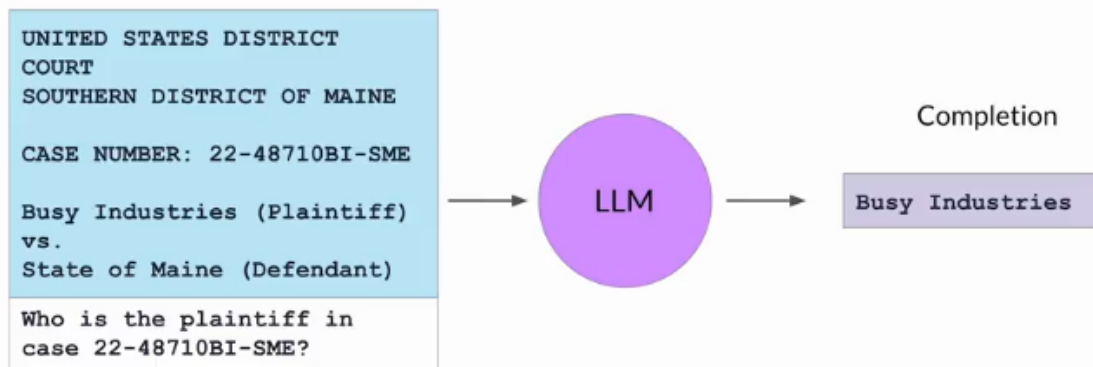
- **Framework** for building LLM powered systems
  - Overcome difficulties faced by the models like **knowledge cut-off**
  - Helps model update it's knowledge
  - Required retraining to update the model isn't necessary
  - More flexible, less expensive - let model access external sources or applications during inference time
  - Improve relevance and accuracy of completion
  - Helps reduce hallucinations
  - Multiple implementations of RAG exist, and you can choose them based on your application
- 
- Retriever = Query encode + External information sources
  - Query encoder converts user input to usable format
  - External information sources is usually a **vector store** / SQL / CSV
  - Retriever returns the most relevant documents from the data source
  - Combine new info to user query
  - This combined prompt is sent to the LLM



## Example: Searching legal documents



## Example: Searching legal documents



2 considerations for using external RAG:

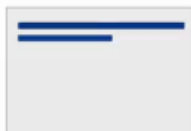
- Data must fit into context window, so data is split into chunks
- This service is provided by LangChain package
- Search must be quick
- Vector stores allow for quick searches

## Data preparation for vector store for RAG

Two considerations for using external data in RAG:

1. Data must fit inside context window

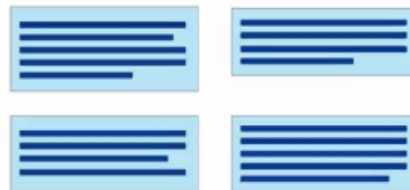
Prompt context limit  
few 1000 tokens



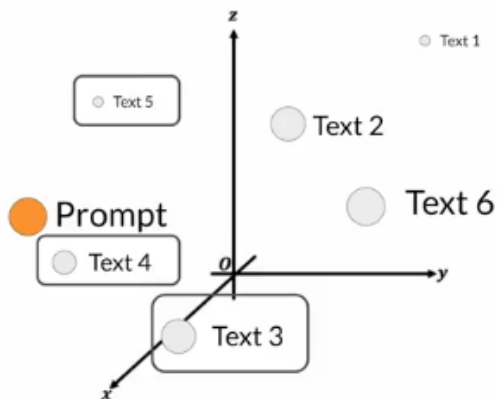
Single document too  
large to fit in window



Split long sources into  
short chunks



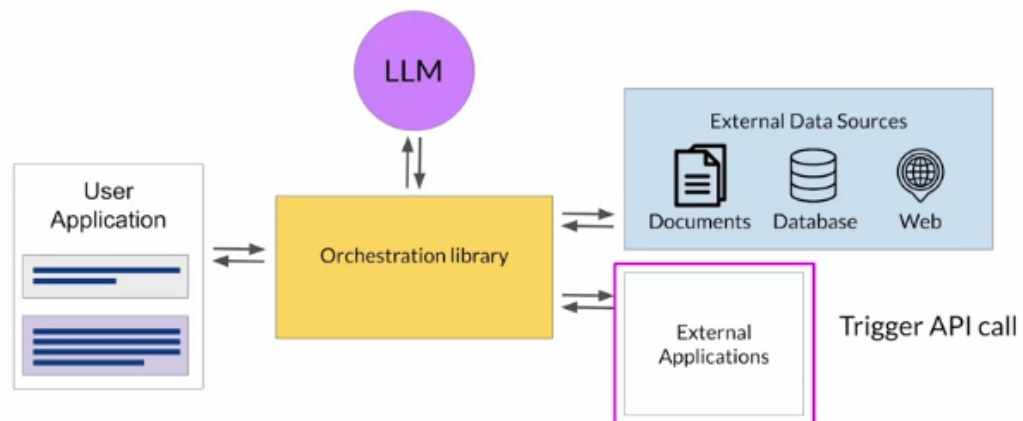
## Vector database search



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

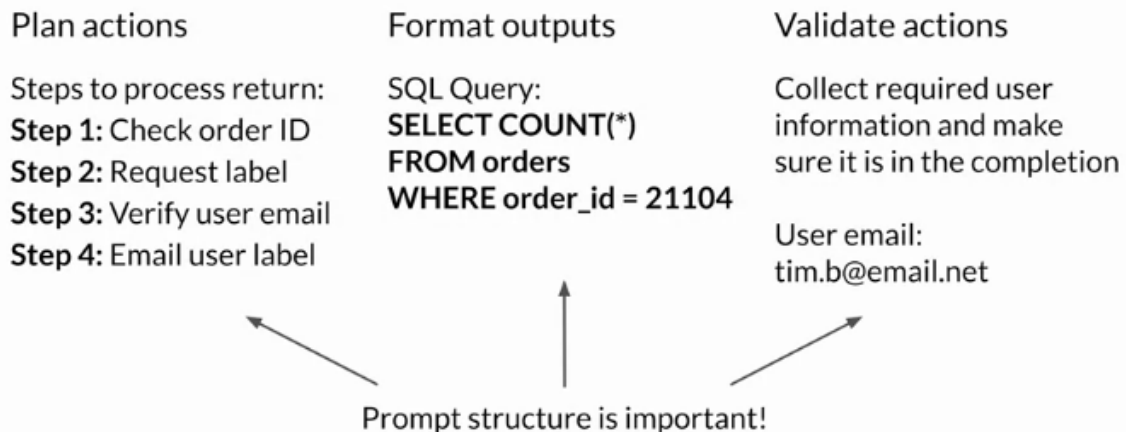
## Interacting with external applications

- There might be a necessity to lookup data, or send requests and receive responses for data
- This can be done via a SQL like lookup with RAG, as seen before
- Request-response cycles are handled by API calls
- Enabling models to interact with external applications extends their capabilities beyond language tasks
- Hence, LLMs can be used to trigger actions through APIs



- LLMs can also connect to other programming resources
  - Eg: python interpreters
- The LLM acts as a **reasoning engine**, as it is responsible for taking appropriate actions for a given prompt

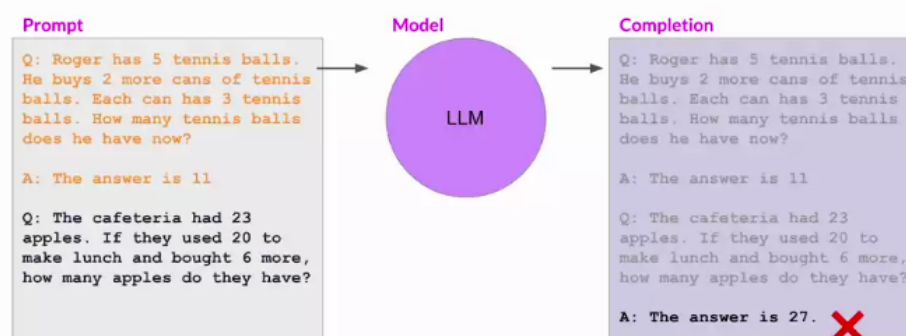
# Requirements for using LLMs to power applications



## Chain of thought

- Need to help LLMs reason and plan with chain of thought
- Need to enable LLMs to plan a set of actions that an application needs to take to satisfy the user's request
- Reasoning is a challenge for LLMs
  - Complex steps makes it hard
  - Mathematics makes it hard

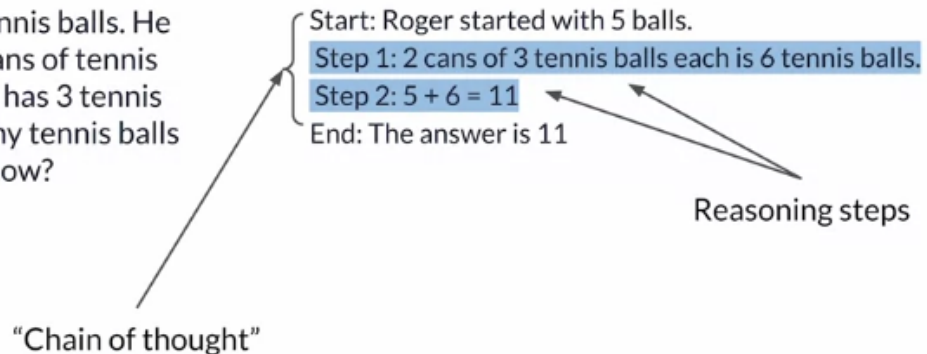
## LLMs can struggle with complex reasoning problems



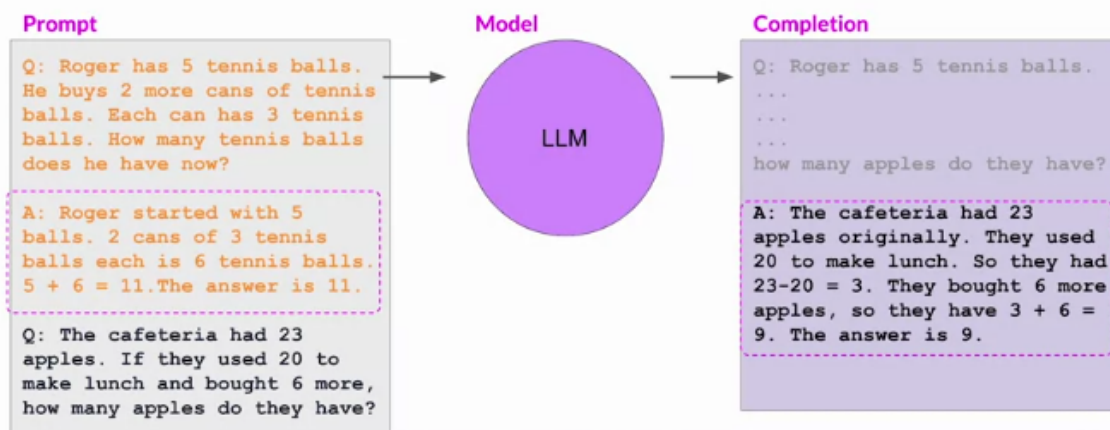
- One thing that can help is: Making the model think like a human by breaking the problem down into **steps**
- **Chain of thought prompting:** Asking the model to simulate human reasoning
- Structure examples for ICL by adding **intermediate reasoning steps**
- This can be extended to not only mathematics, but other problems too

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

Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?



## Chain-of-Thought Prompting can help LLMs reason

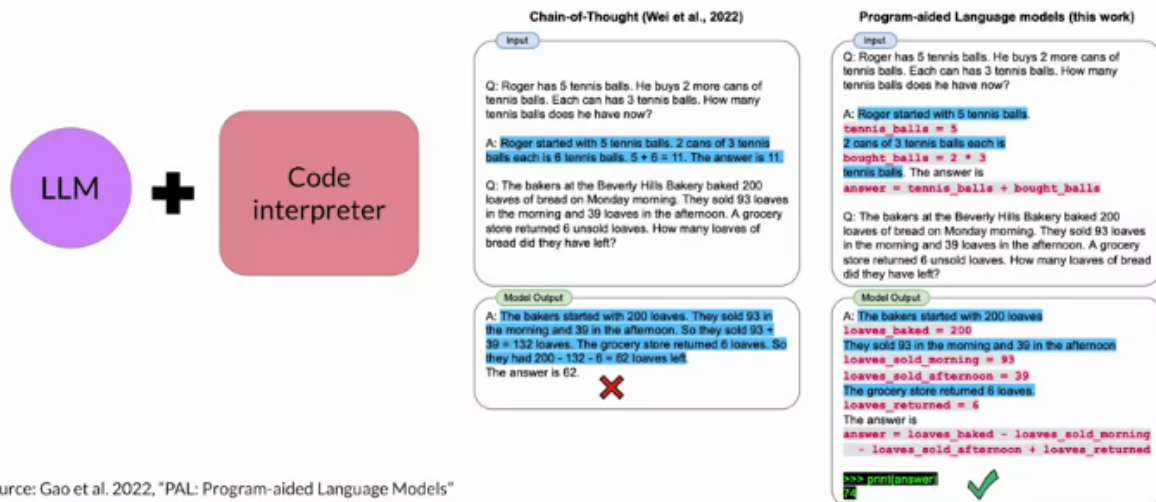


Source: Wei et al. 2022, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models"

## Program Aided Language Models (PAL)

- Ability of LLMs to carry out arithmetic stuff is limited
- Overcome this by allowing model to interact with external applications that are good at math eg: python interpreter
- PAL is a framework that helps to augment LLMs in this way
  - Use an external code interpreter
  - Chain of thought prompting is used to generate **executable** python scripts
  - Use one-shot or few-shot examples to train model

# Program-aided language (PAL) models



Source: Gao et al. 2022, "PAL: Program-aided Language Models"

## PAL example

### Prompt with one-shot example

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

Answer:

```
# Roger started with 5 tennis balls
tennis_balls = 5
# 2 cans of tennis balls each is
bought_balls = 2 * 3
# tennis balls. The answer is
answer = tennis_balls + bought_balls
```

Q: The bakers at the Beverly Hills Bakery baked 200 loaves of bread on Monday morning. They sold 93 loaves in the morning and 39 loaves in the afternoon. A grocery store returned 6 unsold loaves. How many loaves did they have left?

### Completion, CoT reasoning (blue), and PAL execution (pink)

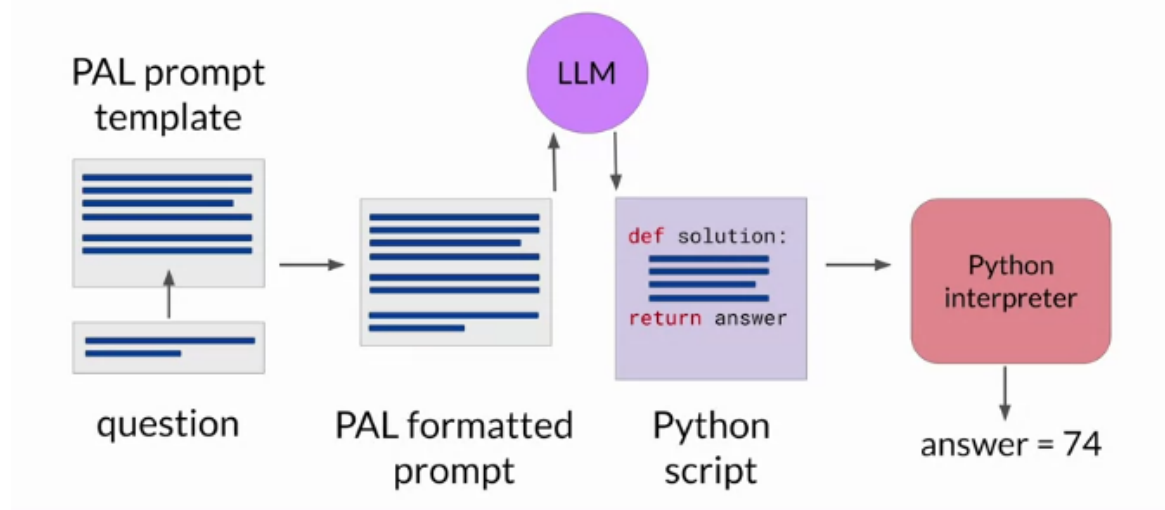
Answer:

```
# The bakers started with 200 loaves
loaves_baked = 200
# They sold 93 in the morning and 39 in the
afternoon
loaves_sold_morning = 93
loaves_sold_afternoon = 39
# The grocery store returned 6 loaves.
loaves_returned = 6
# The answer is
answer = loaves_baked
- loaves_sold_morning
- loaves_sold_afternoon
+ loaves_returned
```

PAL framework:

- Create a prompt template with a few examples in question answer format
- Append the new question to the prompt template
- This creates the **PAL Formatted prompt**
- Pass this to LLM, which generates completion in the form of a python script
- Pass this to the interpreter which computes the answer
- **Orchestrator** manages the flow of information, and calls to external sources

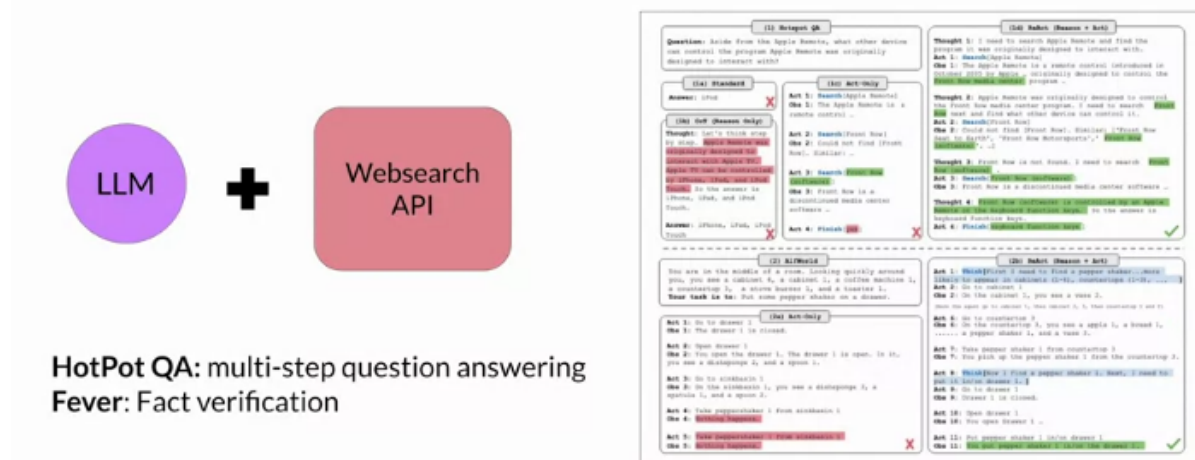
## Program-aided language (PAL) models

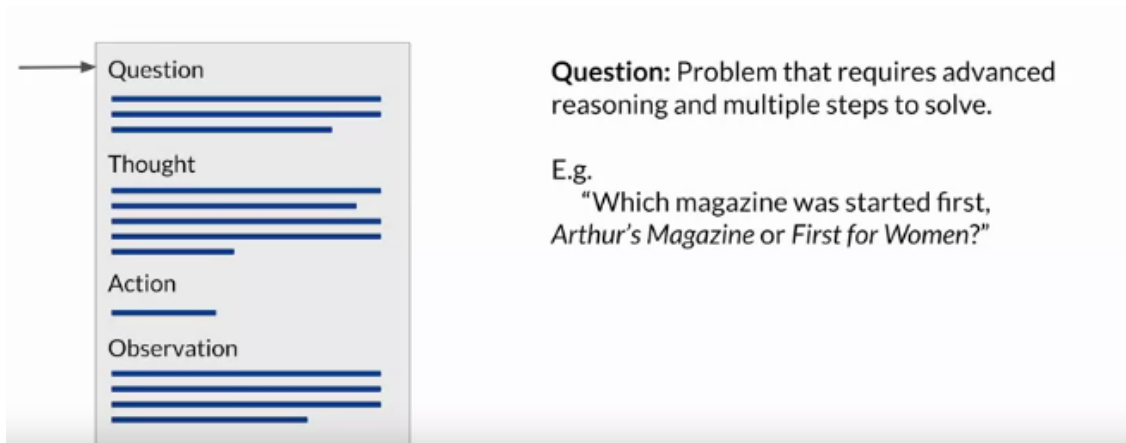


## ReAct - Reasoning and Action

- ReAct allows LLMs to plan and execute their courses of action with multiple interactions with other applications
- Used for more complex applications
- Uses structures examples to show the model how to reason through a problem and decide on actions to take
- The prompt consists of a question, along with the thought, action, and observation
- **Thought:** Reasoning step that teaches the model how to tackle the problem
- **Action:** An external task that the model can take from an allowed set of actions
- **Observation:** Result of carrying out the action
- This cycle is repeated multiple times

## ReAct: Synergizing Reasoning and Action in LLMs





- LLM can only choose from limited set of actions

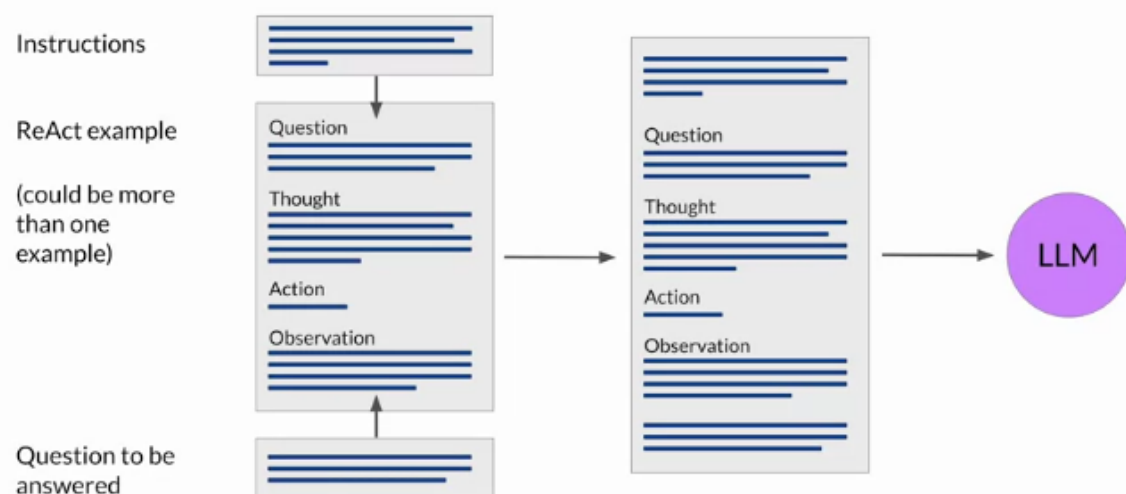
## ReAct instructions define the action space

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

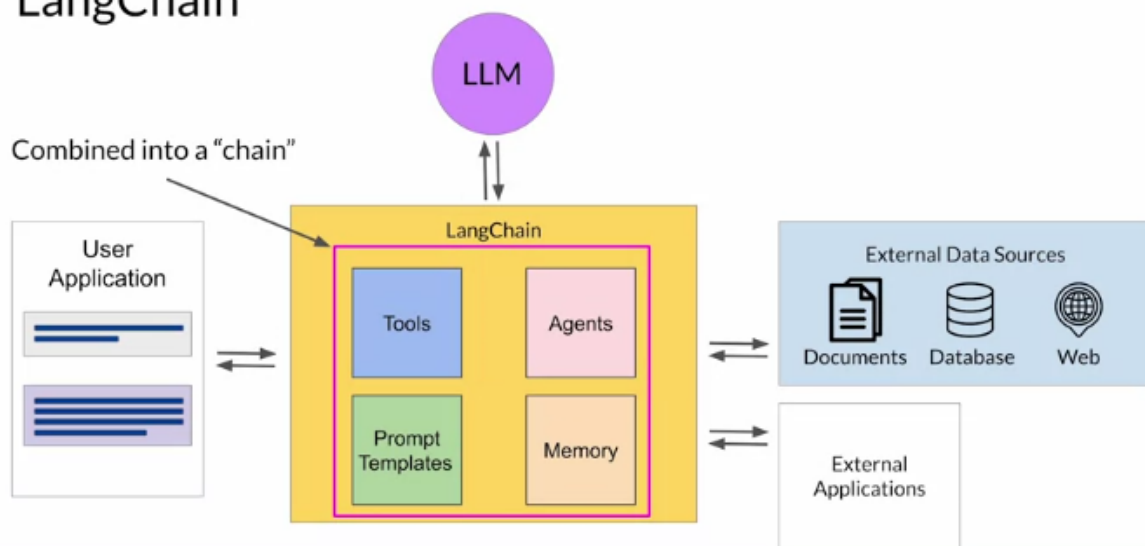
Thought can reason about the current situation, and Action can be three types:

- (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.

## Building up the ReAct prompt



# LangChain



## LLM application architectures

### Building generative applications

