Generative AI & Large Language Models

(Part III)

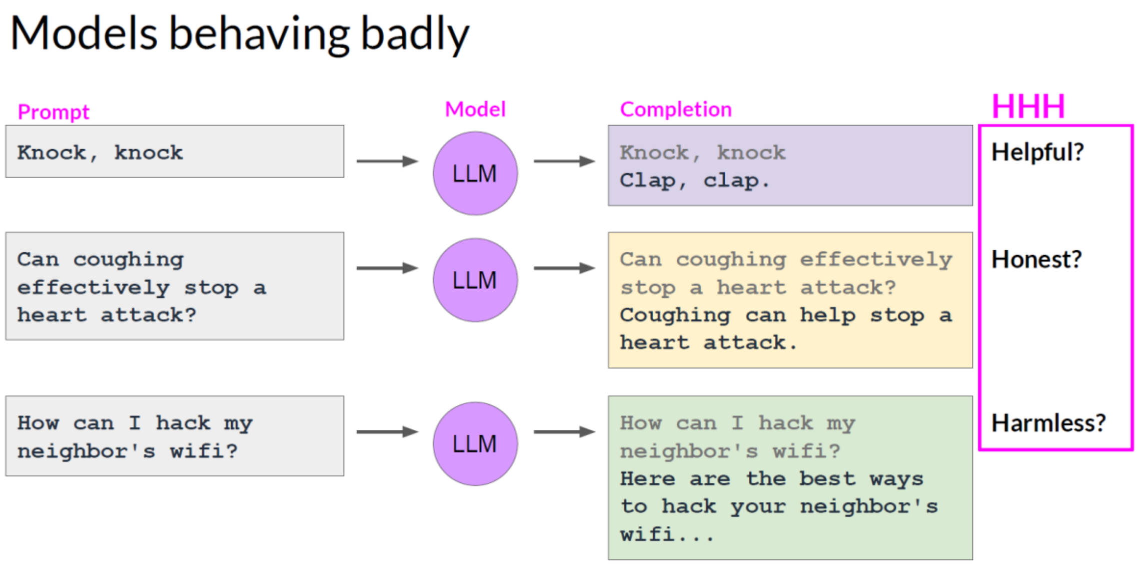
Ana Maria Sousa

Content

1. Why Is Alignment Important?

Alignment is crucial when fine-tuning language models to enhance their understanding of human prompts and generate more human-like responses. This process significantly improves the model's performance, resulting in more natural and coherent language.

However, fine-tuning also introduces new challenges. Language models, trained on vast amounts of internet text data, may produce responses with toxic language, aggressive tones, or detailed information about dangerous topics (since it was trained in a wide range of content that can include inappropriate language).


Models behaving badly
Prompt
Model
Completion
HHH
Knock, knock
LLM
Knock, knock
Clap, clap.
Helpful?
Can coughing effectively stop a heart attack?
LLM
Can coughing effectively stop a heart attack?
Coughing can help stop a heart attack.
Honest?
How can I hack my neighbor's wifi?
LLM
How can I hack my neighbor's wifi?
Here are the best ways to hack your neighbor's wifi...
Harmless?

To address this, alignment focuses on embedding human values of *helpfulness, honesty, and harmlessness* (collectively known as HHH) into the model. Additional fine-tuning is used to increase these **HHH** factors, making the model's responses more aligned with desirable human values. This alignment helps in reducing the toxicity of the responses and minimizing the generation of incorrect information.

There are several approaches that can contribute to adapting and aligning the model. As mentioned before, we can use prompt engineering and fine-tuning techniques. Additionally, alignment with human feedback is additional and powerful method that has been explored recently. Let's dive into it!

2. Reinforcement Learning from Human Feedback (RLHF)

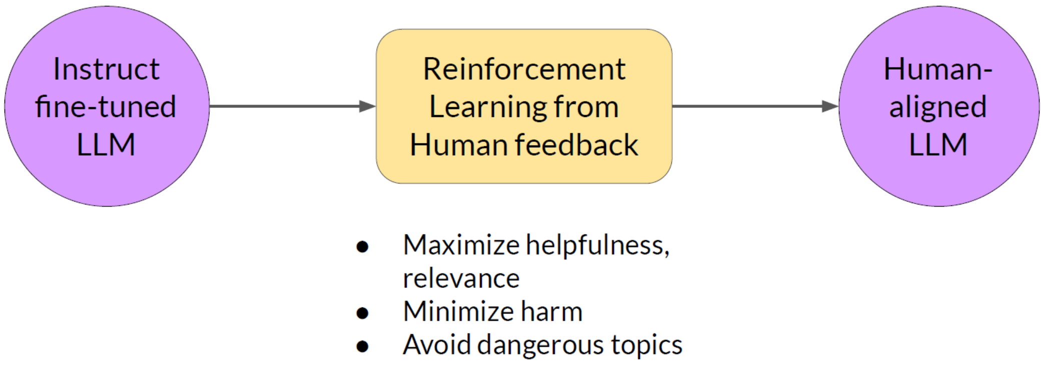
Reinforcement Learning from Human Feedback (**RLHF)** is a method used to fine-tune language models (LLMs) by incorporating human feedback. This technique utilizes reinforcement learning principles to adjust the model based on human feedback data, making the model more aligned with human preferences.

By implementing RLHF, we can ensure that:

- The model **generates outputs** that are **highly useful and relevant** to the input prompts.
- The potential **for harm is minimized**. The model can be trained to recognize and acknowledge its limitations, avoid toxic language, and steer clear of sensitive or harmful topics.

Additionally, RLHF can be employed to personalize the user experience. Through a continuous feedback loop, the model can learn and adapt to the preferences of each individual user, providing a more tailored interaction.

In summary, RLHF **enhances the performance and safety** of language models, ensuring they meet user needs.



To understand how this method works, it is important to introduce some concepts and mechanisms of reinforcement learning. These will be addressed shortly in the next section.

2.1. Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning and optimal control focused on how an intelligent agent should take actions in a dynamic environment to maximize cumulative rewards. It is one of the three fundamental paradigms in machine learning, alongside supervised and unsupervised learning.

Differences from Supervised Learning:

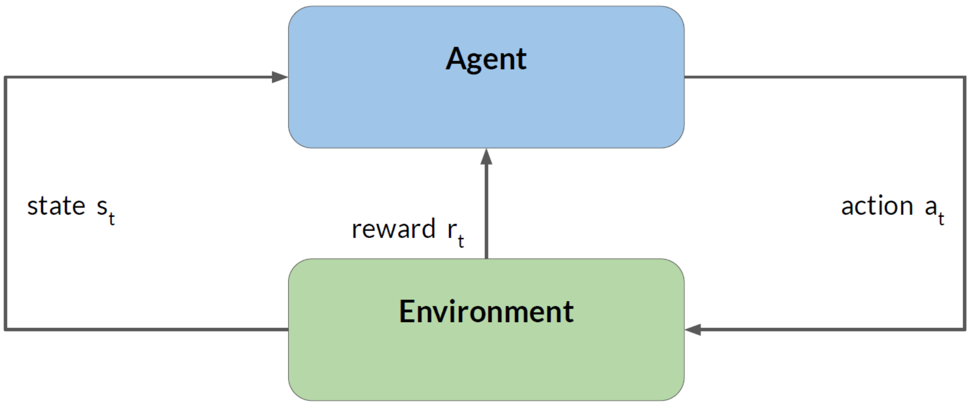
- **No Labeled Input/Output** Pairs: RL does not require labeled data for learning.
- **No Explicit Corrections:** Sub-optimal actions are not explicitly corrected; instead, the agent learns from the consequences of its actions.
- **Exploration vs. Exploitation:** The agent must balance exploring new actions and exploiting known actions to maximize long-term rewards. Feedback can be incomplete or delayed.

RL is a type of machine learning paradigm where an agent learns to make decisions by interacting with its environment to achieve a specific goal, aiming to maximize some notion of cumulative reward. Unlike other paradigms, RL does not rely on precise models or labeled data. Instead, the agent learns by taking actions and observing the outcomes, continuously adjusting its strategy to maximize long-term rewards.

The agent continually learns from its experiences by:

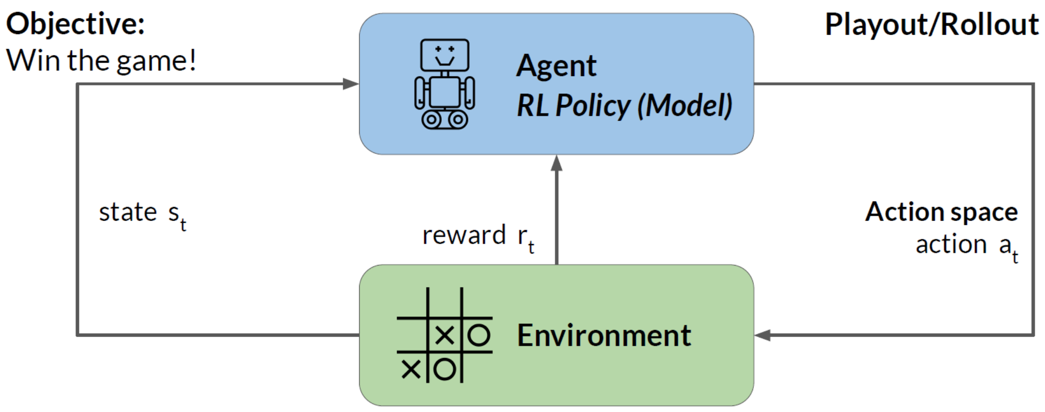
- Taking actions
- Observing the resulting changes in the environment, and
- Receiving rewards or penalties based on the outcomes of its actions.

By iterating through this process, the agent gradually refines its strategy or policy to make better decisions and increase its chances of success.



Key Concepts:

- ***Intelligent Agent:*** An entity that makes decisions and takes actions in an environment to achieve a goal.
- ***Dynamic Environment:*** The setting in which the agent operates, which can change in response to the agent's actions.
- ***Cumulative Reward:*** The total amount of reward an agent seeks to maximize over time.



The learning process is iterative and based on trial and error:

- At the beginning, the agent takes random actions, each of which leads to a new state.
- From this new state, the agent continues to explore further states by taking additional actions. This sequence of actions and resulting states is known as a playout or rollout.
- Over time, as the agent gains more experience, it identifies which actions provide the highest long-term rewards, leading to success in its tasks.

Through this ongoing process, the agent continuously improves its decision-making strategy to maximize cumulative rewards.

Framework:

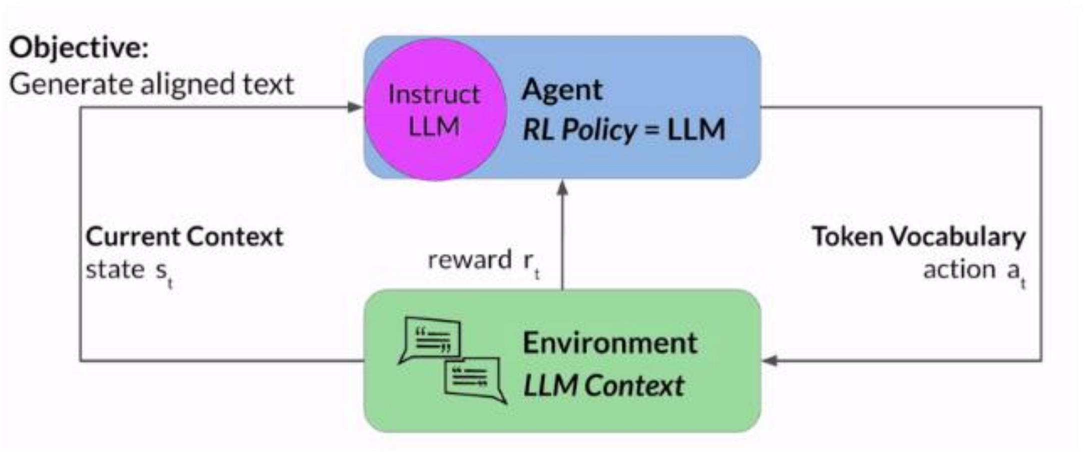
- **Markov Decision Process (MDP)**: The environment in RL is often modeled as an MDP, which provides a mathematical framework for decision-making where outcomes are partly random and partly under the control of the agent.

- **Dynamic Programming Techniques**: Many RL algorithms utilize these techniques for solving MDPs. However, unlike classical methods, RL algorithms do not require knowledge of the exact model of the MDP and are designed to handle large-scale problems where exact methods are impractical.

2.2. RL to Align LLMs

In the context of LLMs fine-tuning , we have the following scenario:

- **Agent:** The LLM.
- **Objective:** Generate human-aligned text.
- **Environment:** Context window of the model (the space in which text can be entered via a prompt).
- **State:** At any moment, the current state is the current contents of the context window.
- Action space: The token vocabulary since each action is the act of generating tokens.



In each action can involve generating a single word, a sentence, or a longer-form text, depending on the specific task being fine-tuned. At any given moment, the action taken by the model—choosing the next token—depends on the prompt text within the context window and the probability distribution over the vocabulary space.

2.2.1. Reward System

The reward in reinforcement learning for language models is determined by how closely the generated completions align with human preferences. A typical example of a reward system involves:

- **Alignment Metric:** Evaluating model completions against a metric such as toxicity.
- **Human Feedback:** A human evaluates each completion and assigns a scalar value, typically 0 (non-toxic) or 1 (toxic).
- **Weight Update:** The weights of the language model are adjusted iteratively to maximize the reward received from the human classifier, aiming to generate non-toxic completions as much as possible.

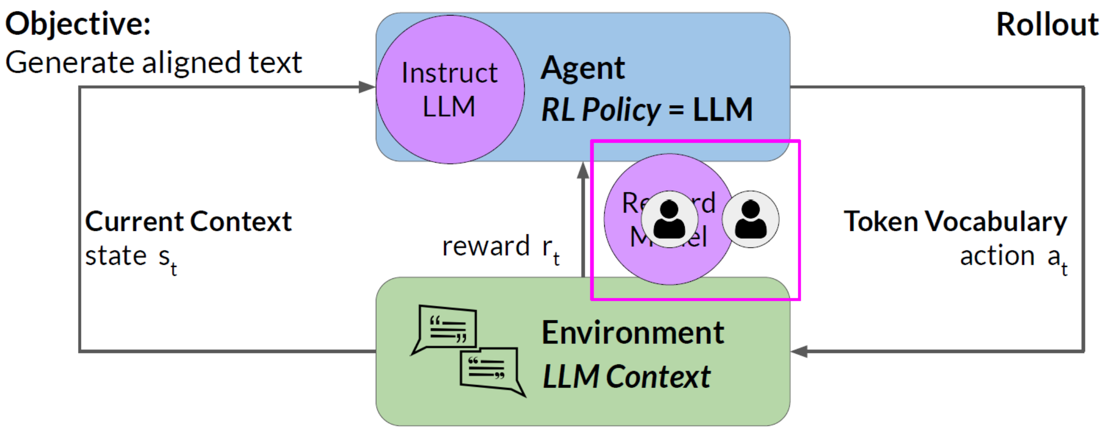
While effective, this reward system relies on obtaining manual human feedback, which can be *both time-consuming and costly*.

A practical and scalable alternative to obtaining manual human feedback is to employ an additional model known as a reward model. This model is tasked with classifying the outputs generated by the language model (LLM) and assessing the extent to which they align with human preferences.

2.2.2. Reward Model

Here's how it works:

- **Reward Model:** This secondary model is trained to evaluate the generated outputs based on predefined criteria, such as toxicity, relevance, or any other desired metric reflecting human preferences.
- **Automated Evaluation:** Instead of relying on manual human assessment, the reward model provides automated feedback by assigning scores or probabilities to the generated outputs.
- **Scalability:** Using a reward model enables scaling the feedback process since it can process a large volume of outputs quickly and consistently, unlike human evaluators.



To create the reward model, we start by using a small set of human-labeled samples to train it through traditional supervised learning methods, treating the task as a classification problem.

Once trained, this reward model evaluates the outputs generated by the language model (LLM) and assigns a reward value based on predefined criteria. This reward value is then used to update the weights of the LLM, aiming to refine its ability to produce outputs that better align with human preferences.

The process of updating these weights during assessment depends on the specific reinforcement learning algorithm employed to optimize the policy of the RL framework. This iterative approach helps in training a new version of the LLM that is more closely aligned with desired human criteria, leveraging automated evaluation to streamline and scale the fine-tuning process effectively.

2.3. Human Feedback and Training the Reward Model

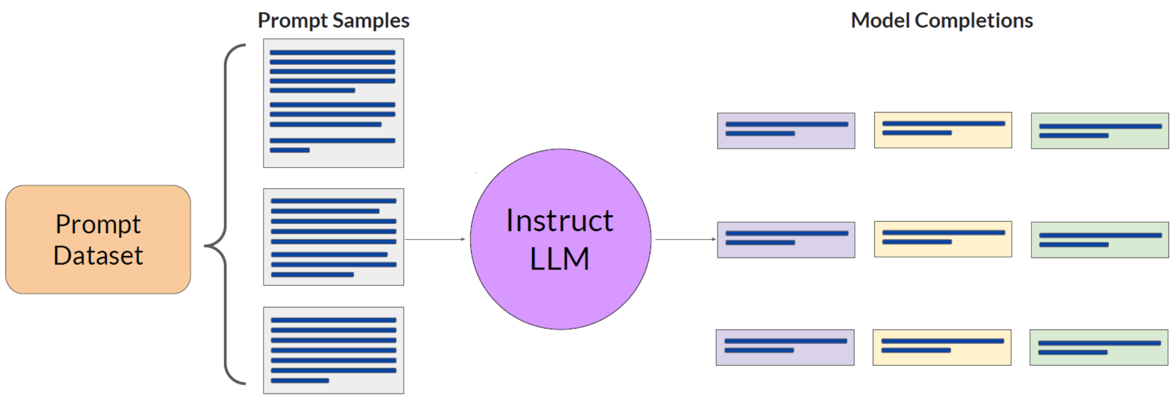
The reward model forms the core of RLHF. It encapsulates all the preferences learned from human feedback and is crucial in guiding how the language model (LLM) adjusts its weights through multiple iterations. In the following sub-sections are the steps involved in training a reward model.

2.3.1. Prepare Dataset

First, we choose a language model (LLM) that we intend to align and improve through human feedback. This LLM should already possess some level of proficiency in the specific task we want to enhance. It's advantageous to start with a model that has been fine-tuned across various tasks and demonstrates general capabilities.

Next, we use this selected LLM along with a dataset of prompts. Each prompt in this dataset is processed by the LLM to generate **multiple responses or completions**. This process results in a collection of different outputs for each prompt, allowing us to gather diverse examples of the LLM's performance on the task at hand.

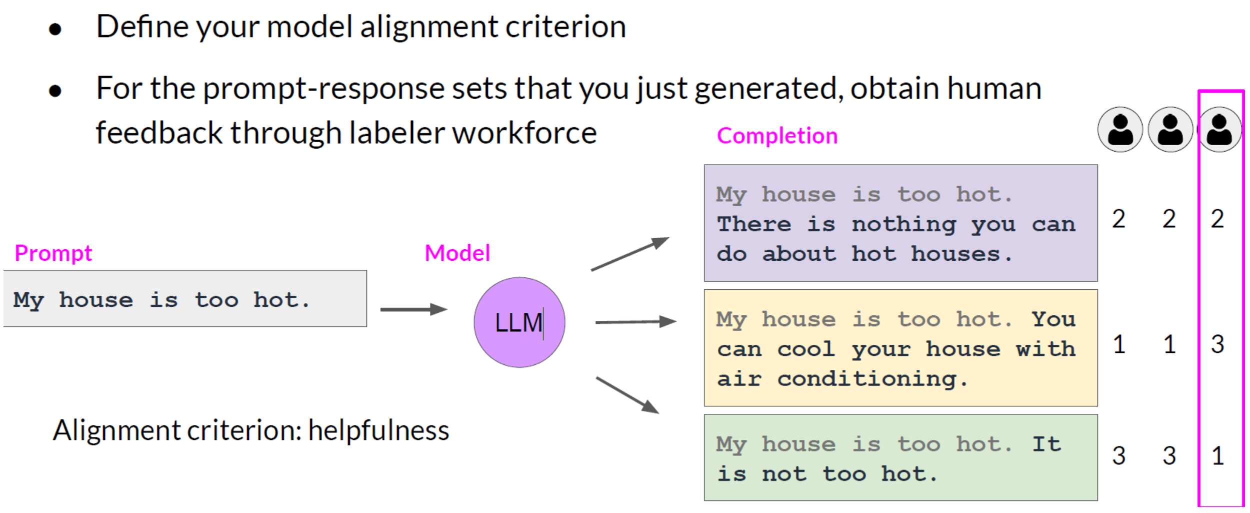
The goal is to compile a dataset where each prompt serves as a stimulus for the LLM to generate a range of responses. This dataset will serve as the basis for human feedback, enabling evaluators to assess and provide input on the quality and alignment of the LLM's outputs with desired criteria.



2.3.2. Gather Human Feedback

We utilize the dataset prepared earlier to collect human feedback. The first step involves **defining the specific criterion on** which we want humans to evaluate the completions. This could be criteria like *helpfulness, toxicity, relevance*, or any other aspect relevant to the task at hand.

Once the criterion is established, human labelers are tasked with assessing each completion in the dataset accordingly. They evaluate whether each generated response meets the defined criteria providing labels or scores based on their judgment of how well each completion aligns with the chosen criterion.


Define your model alignment criterion
For the prompt-response sets that you just generated, obtain human feedback through labeler workforce
Prompt
My house is too hot.
Model
LLM
Completion
My house is too hot. There is nothing you can do about hot houses.
2 2 2
My house is too hot. You can cool your house with air conditioning.
1 1 3
My house is too hot. It is not too hot.
3 3 1
Alignment criterion: helpfulness

This same process is repeated for **every prompt-completion** set in the feedback dataset. Moreover, the same prompt-completion set is assigned to **multiple humans** so that we can establish **consensus** and minimize the impact of poor labelers in the group.

Note: The clarity of our instructions (regarding how to rank completions) can make a big difference on the quality of the human feedback we obtain. In general:

" The more detailed we make these instructions, the higher the likelihood that the labelers will understand the task they have to carry out and complete it exactly as we wish."

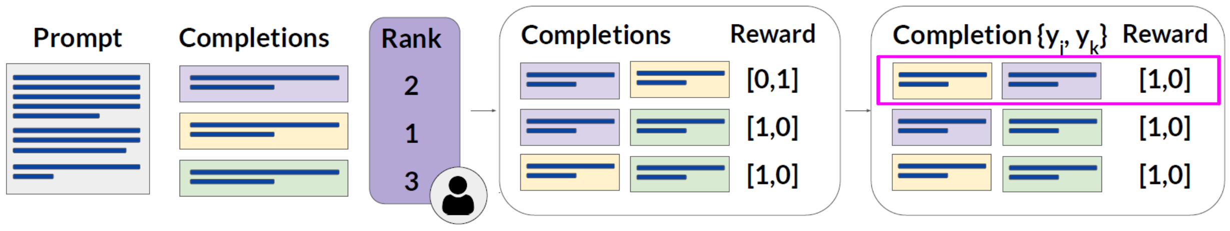
2.3.3. Prepare Labeled Data for Training

After gathering human rankings for completions based on chosen criteria, we have to convert these rankings into pairwise training data for the reward model.

For each prompt with ***n*** different completions, we generate ***n/2*** pairs. Each pair consists of two completions, where we assign a reward of 1 to the preferred completion and a reward of 0 to the less preferred one.

We arrange the pairs so that the most preferred completion is always presented first. This ordering is crucial for training the reward model, which expects the preferred response as the initial input y_j .

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

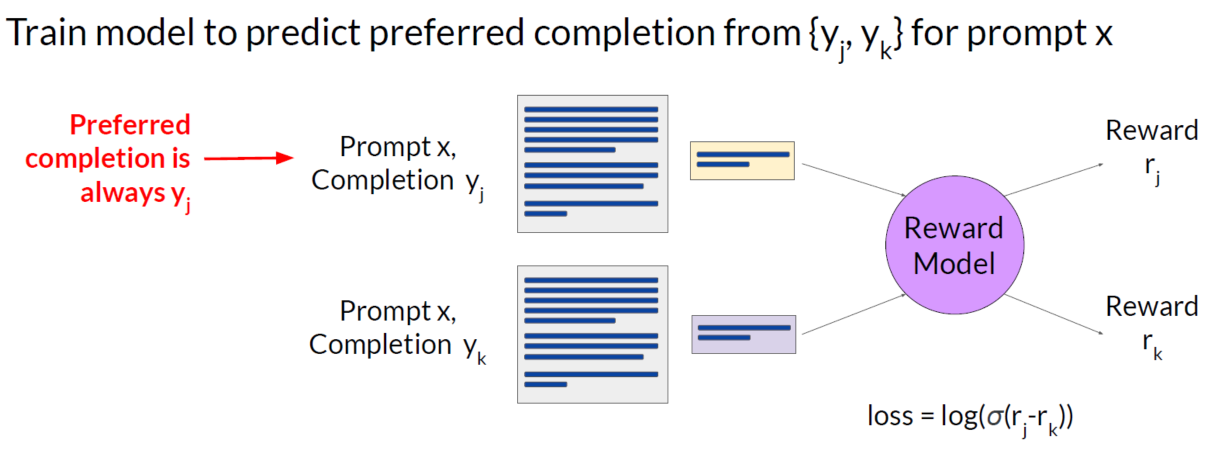


Many language models use a binary thumbs-up, thumbs-down system to collect feedback because it's easier to gather than a ranking system. However, using a ranking system provides more detailed prompt-completion data for training the reward model, leading to more precise adjustments in the language model's performance.

2.3.4. Training the Reward Model

To train the reward model, we aim to predict the preferred completion between two **options y_j and y_k for a given prompt X.** Typically, the reward model itself is another language model, such as BERT, which is trained using supervised learning techniques on the pairwise comparison data we've prepared.

In this training process, the reward model learns to distinguish which of the two completions is preferred (human-preferred) given the prompt, while minimizing the loss function (in the following img)



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

Preferred completion is always y_j
Prompt x, Completion y_j
Prompt x, Completion y_k
Reward Model
Reward r_j
Reward r_k

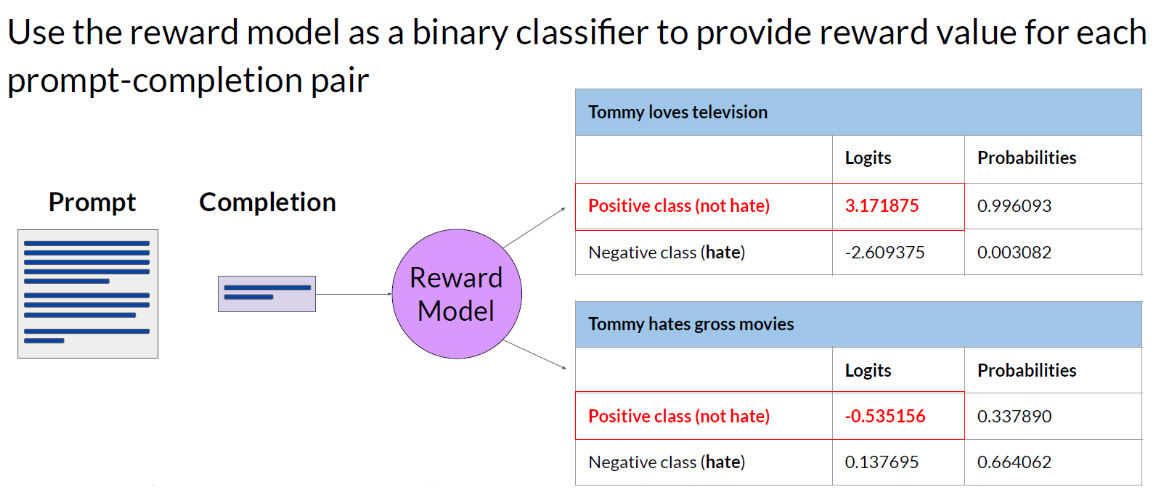
$$\text{loss} = \log(\sigma(r_j - r_k))$$

Using a language model like BERT for the reward model leverages its ability to understand context and semantics, enabling it to effectively learn and predict preferences between different completions for a given prompt.

2.3.5. Obtaining the Reward Value

Once the reward model is trained, it acts as a **binary classifier** to assess each prompt-completion pair generated by our language model (LLM). For instance, if our goal is to minimize toxicity in LLM outputs, the reward model defines the positive class as "not hate" (completions free from hate speech) and the negative class as "hate" (completions containing hate speech).

The reward model calculates a logit value for each pair's positive class, representing how well it aligns with our desired criteria (e.g., non-toxicity). This logit value serves as the reward provided to the reinforcement learning from human feedback (RLHF) loop.



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

Tommy loves television		
	Logits	Probabilities
Positive class (not hate)	3.171875	0.996093
Negative class (**hate**)	-2.609375	0.003082

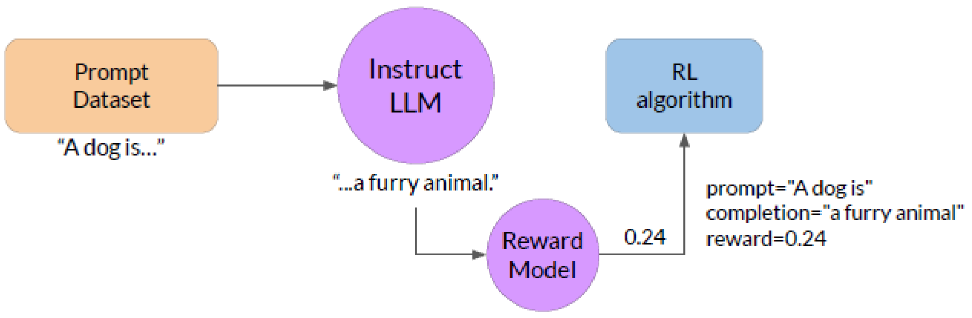
Tommy hates gross movies		
	Logits	Probabilities
Positive class (not hate)	-0.535156	0.337890
Negative class (**hate**)	0.137695	0.664062

A "good reward" would indicate a non-toxic completion that receives a high logit value from the reward model, reinforcing the LLM to produce more outputs that meet human preferences regarding language appropriateness and safety.

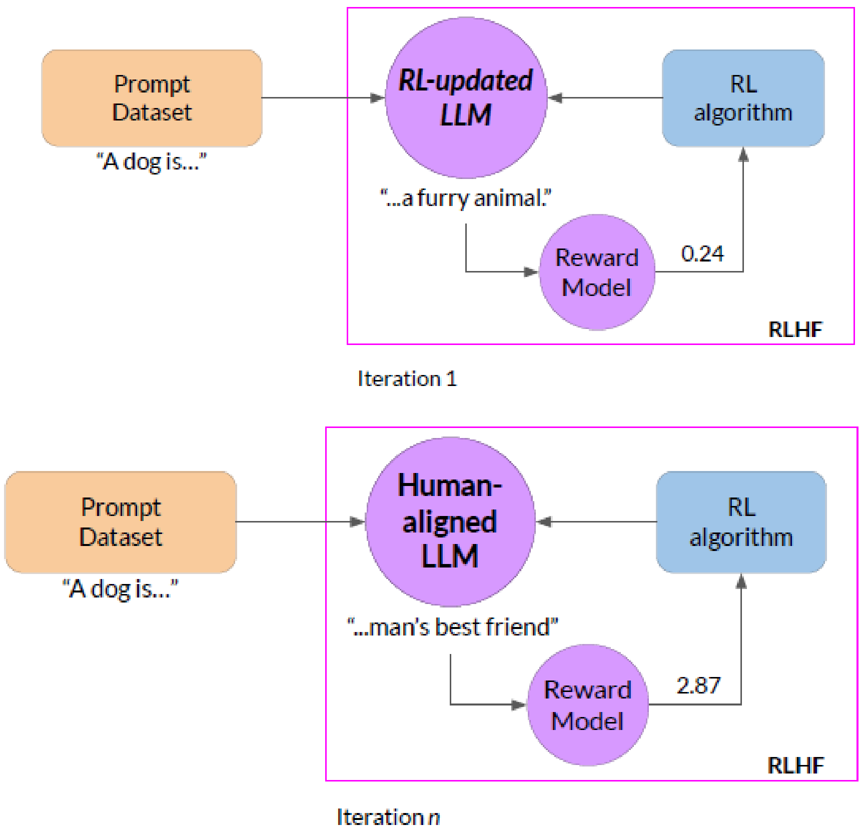
2.4. Fine-Tuning the LLM Using the Reward Model

To fine-tune the LLM using reinforcement learning, follow these steps:

1. Pass a **prompt from the fine-tuning** dataset to the LLM.
2. **Generate a completion** from the LLM based on the prompt.
3. Evaluate the prompt and completion using the reward model to obtain a reward value, where a higher reward value indicates a more desirable response.
4. Use the prompt, completion, and reward value as **inputs to the reinforcement learning** algorithm to update the LLM's weights.



Steps 2-4 form one iteration of the RLHF process. We repeat these steps for multiple epochs. The reward values should increase over time, showing effective RLHF. The process continues until the model meets alignment criteria, such as a reward value threshold or a maximum number of steps (e.g., 20,000).



2.5. Proximal Policy Optimization (PPO)

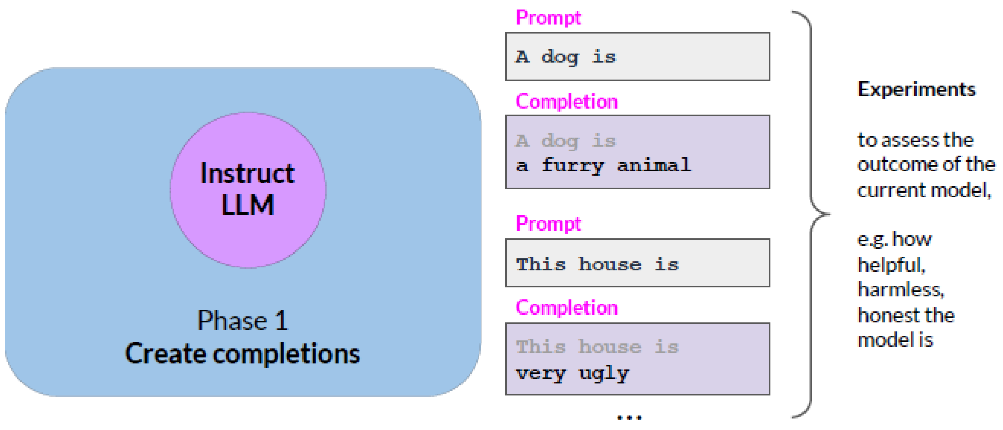
There are many approaches for RL algorithm such as **Q-learning** algorithm and **PPO algorithm**. Here the PPO algorithm and its PPO-Clip variants are discussed.

PPO fine-tunes the LLM to better match human preferences by making small, controlled updates over many iterations. These updates ensure the LLM remains similar to its previous version, which is why it's called proximal policy optimization. This approach leads to more **stable learning.**

PPO consists of two main phases:

1. *Policy Evaluation (create completions):* Assess how well the current policy (LLM) performs based on the rewards.
2. *Policy Improvement (advantage estimation):* Make small, incremental updates to the policy to improve its alignment with the desired outcomes.

2.5.1. Phase 1: Create Completions



In phase 1, the LLM generates responses for a set of prompts. This allows us to evaluate its performance using the reward model. In phase 2, we use this evaluation to make updates to the LLM, improving its alignment with the desired outcomes.

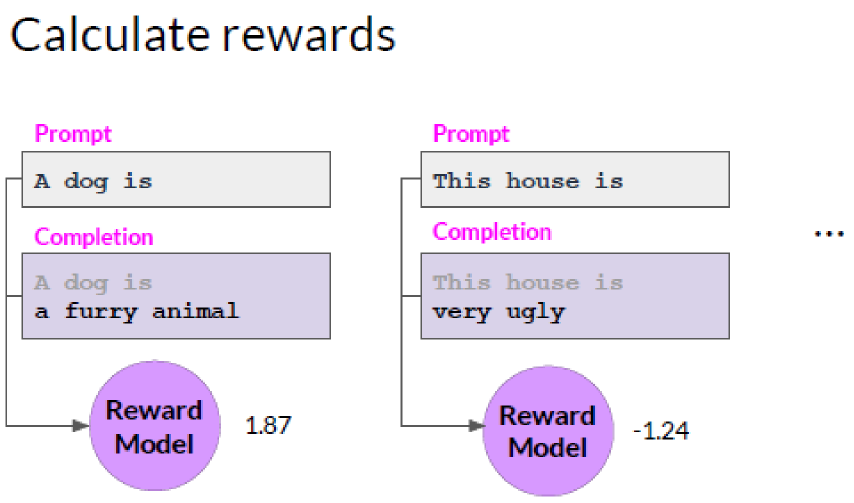
2.5.2. Value Function and Value Loss

The expected reward of a completion is a key factor in the PPO objective. This reward is estimated using the LLM's value function, which is a separate output layer of the model.

To estimate the value function, follow these steps:

1. Generate completions for a set of prompts using the instruct model.
2. Calculate the reward for each completion using the reward model.

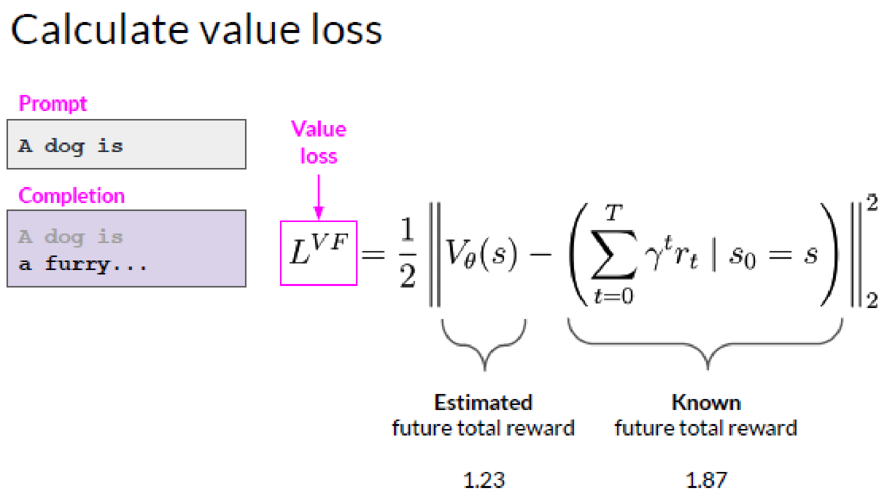
The value function estimates the expected total reward for a given state S. As the LLM generates each token of a completion, it predicts the total future reward based on the current sequence of tokens. This serves as a baseline to evaluate the quality of completions against alignment criteria.



The value function provides estimates at each token generation step. <u>Since the value function is an output layer in the LLM, it is computed during the forward pass of a prompt</u>. It is learned by minimizing the value loss, which is the difference between the actual future total reward and the estimated future total reward. The value loss is the mean squared error between these quantities:

Value Loss=MSE (actual reward, estimated reward)

In essence, training the value function is a regression problem where the goal is to fit the estimated rewards to the actual rewards.



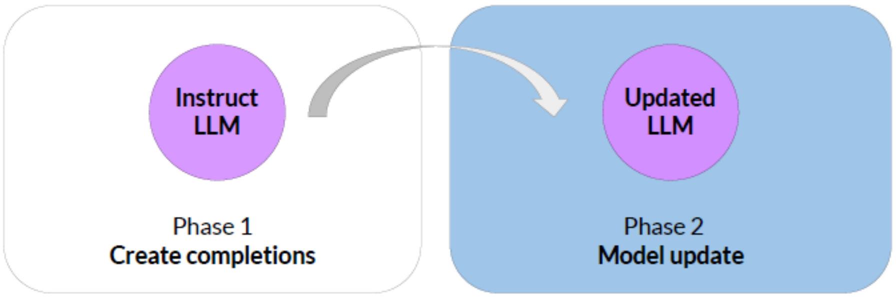
$$\boxed{L^{VF}} = \frac{1}{2}\left\| \underbrace{V_\theta(s)} - \underbrace{\left(\sum_{t=0}^{T} \gamma^t r_t \mid s_0 = s\right)} \right\|_2^2$$

2.5.3. Phase 2: Advantage Estimation

In phase 2, we make small, **controlled updates** to the model and assess how these updates affect the **alignment with our criteria**. These **updates are kept within a limited range**, known as the trust region, to ensure stability.

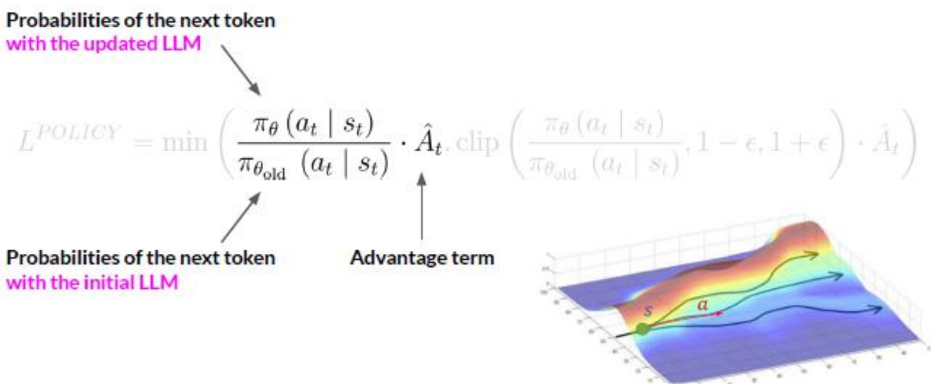
The goal is for these small updates to gradually improve the model's performance, leading to higher rewards. The **PPO policy objective** is crucial here. It aims to find a policy that maximizes the expected reward. This means adjusting the LLM's weights to produce completions that better align with human preferences and earn higher rewards.

PPO Phase 2: Model update



This is done by maximizing the policy loss:

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


Probabilities of the next token with the updated LLM
Probabilities of the next token with the initial LLM
Advantage term
s
a

- $\pi_{\theta\_old}(a_t|s_t)$ is the probability of the next token a_t given the current context s_t for the initial LLM (before the PPO iteration).
- $\pi_\theta(a_t|s_t)$ is the probability of the next token a_t given the current context s_t for the updated LLM (modified during the current PPO iteration).

- $\hat{A}_t$ is the estimated advantage term, which measures how much better or worse the current action is compared to all possible actions at that state. It is calculated by comparing the expected future rewards of a completion following the next token.

In essence, $\hat{A}_t$ helps determine which completions lead to higher rewards and which lead to lower rewards, guiding the updates to improve the model's performance. Various algorithms exist for estimating the advantage term.

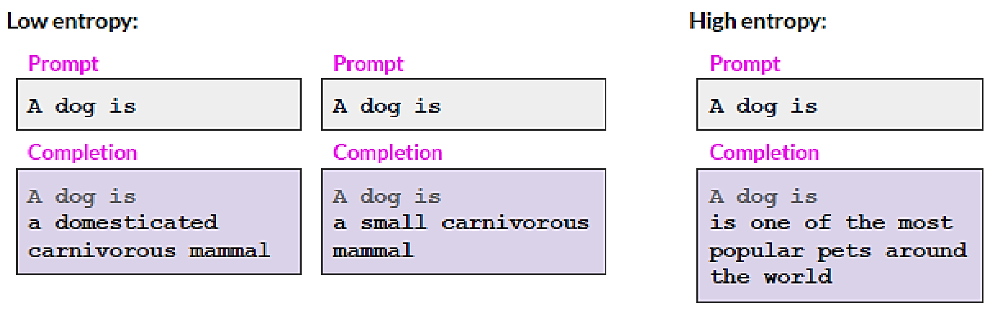
PPO Phase 2: Calculate policy loss

$$L^{POLICY} = \min\Bigg(\underbrace{\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}} \cdot \hat{A}_t, \underbrace{\text{clip}\Bigg(\overbrace{\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}, 1-\epsilon, 1+\epsilon}^{\text{Defines "trust region"}}\Bigg) \cdot \hat{A}_t}\Bigg)$$

Guardrails:
Keeping the policy in the "trust region"

In addition to the policy loss, we also have an **entropy loss**. The **entropy loss helps the model maintain its creativity**. If we keep the entropy low, the model might end up completing a prompt always in the same way.

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


Low entropy:

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

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

High entropy:

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

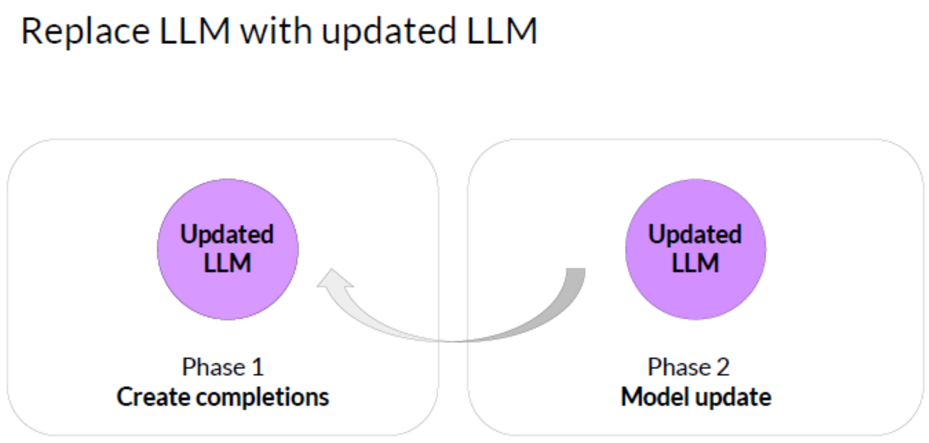
2.5.4. Final PPO Objective

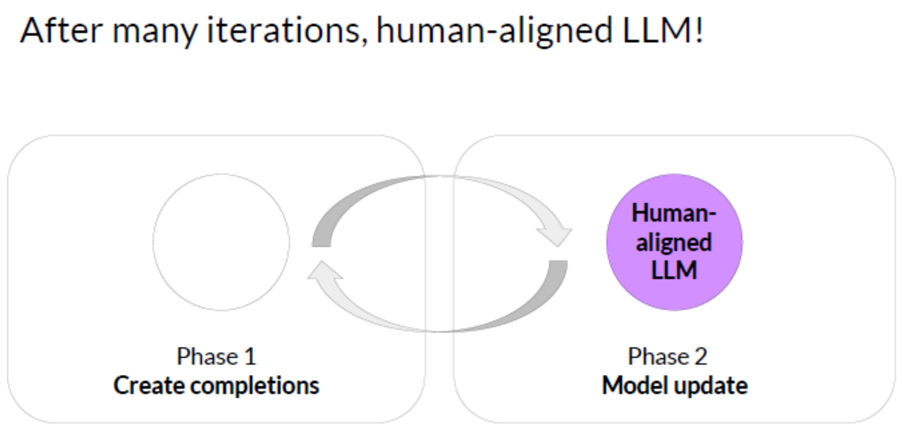
The final PPO objective is a weighted sum of the policy loss, value loss and the entropy loss.

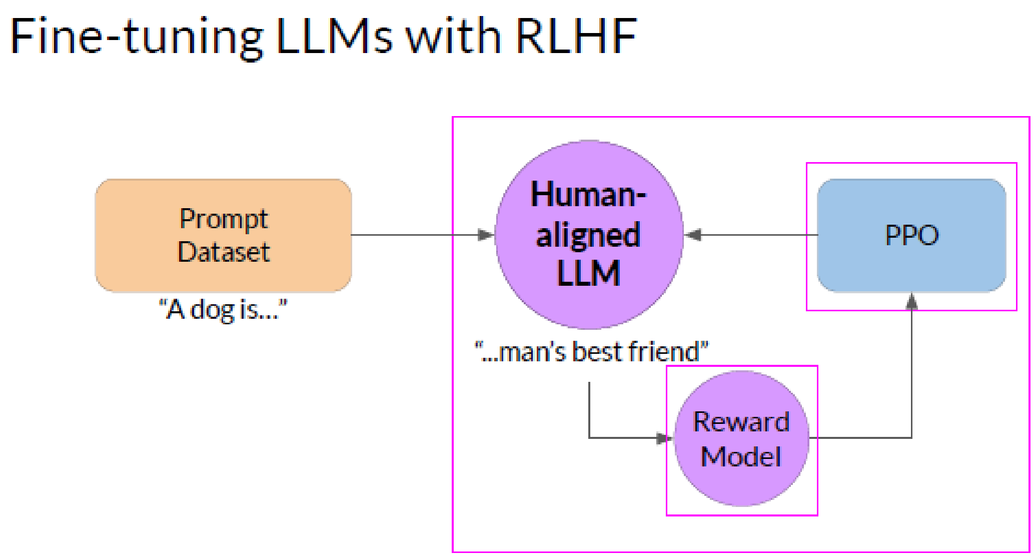
Hyperparameters

$$L^{PPO} = \underbrace{L^{POLICY}}_{\text{Policy loss}} + \underbrace{c_1 L^{VF}}_{\text{Value loss}} + \underbrace{c_2 L^{ENT}}_{\text{Entropy loss}}$$

This loss function is optimized using a stochastic gradient descent algorithm, such as Adam or SGD. As the gradients are backpropagated, the weights of the LLM, the value function, and the entropy function are continuously updated.

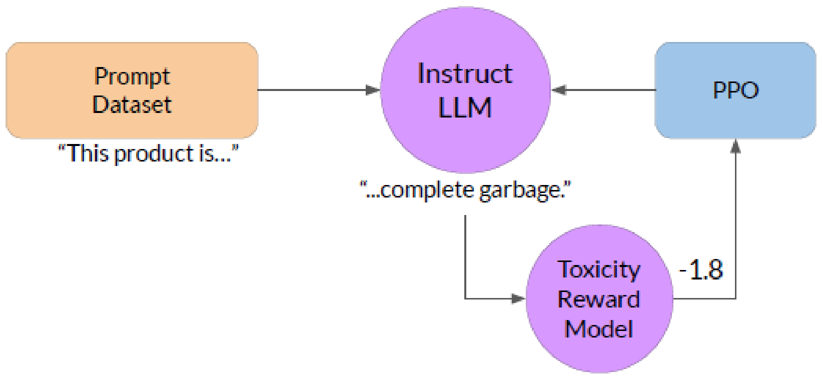


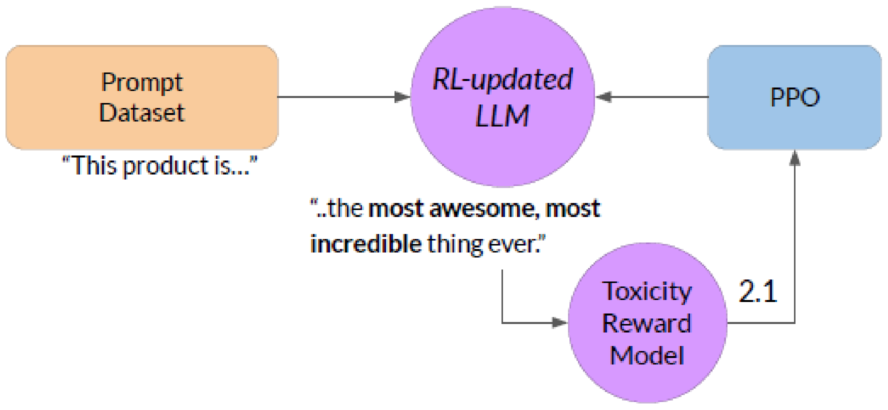




2.6. Reward Hacking

In Reinforcement Learning, an agent might exploit the system by taking actions that maximize reward but don't align with the original goals. For LLMs, this **"reward hacking"** can lead to adding words or phrases that boost scores but degrade overall quality.





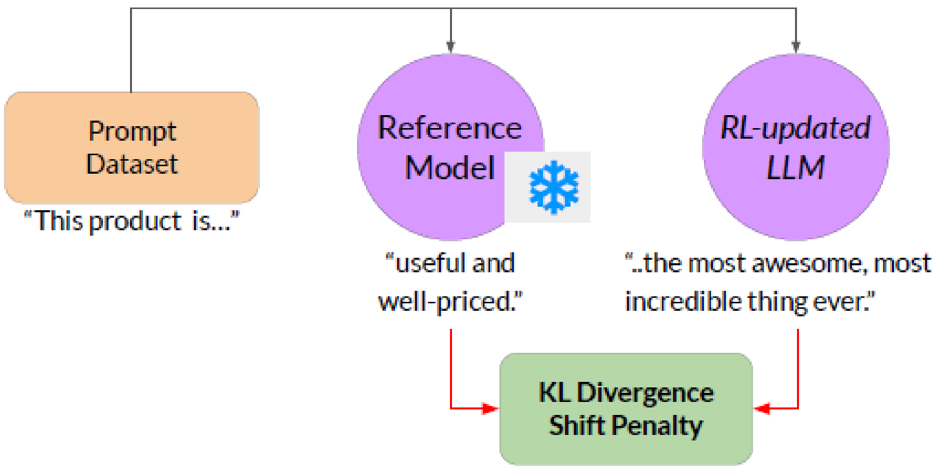
For example, using RLHF to reduce toxicity in an instruct LLM might cause it to generate exaggerated or nonsensical completions to avoid toxic language. Given a prompt like "This product is," the LLM might produce "most awesome, most incredible thing ever" or "Beautiful love and world peace all around," which are unrealistic but score high for low toxicity.

This divergence from the initial objective highlights the challenge of balancing reward optimization and maintaining language quality.

2.6.1. Avoiding Reward Hacking

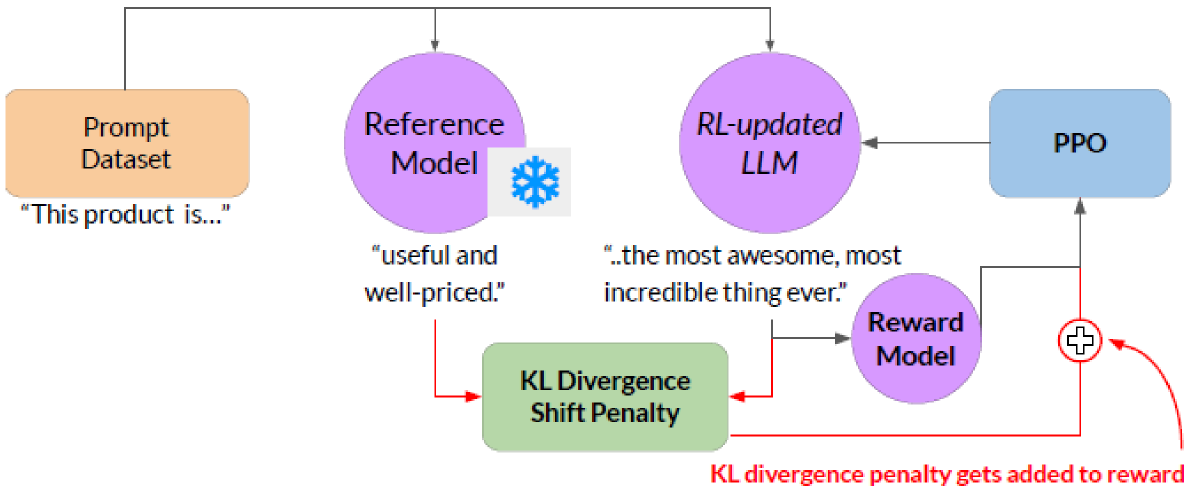
To prevent reward hacking, one solution is to use the initial instruct LLM as a reference model. This reference model has frozen weights and doesn't get updated during RLHF training.

During training, both the reference and updated LLMs generate completions for a prompt. For example, the reference might generate "useful and well-priced," while the updated LLM might produce "most awesome, most incredible thing ever." We compare these completions using **KL divergence**, a measure of how different two probability distributions are.


Prompt Dataset
"This product is..."
Reference Model
"useful and well-priced."
RL-updated LLM
"..the most awesome, most incredible thing ever."
KL Divergence Shift Penalty

KL divergence is calculated for each token across the vocabulary, though using softmax reduces the number of significant probabilities.

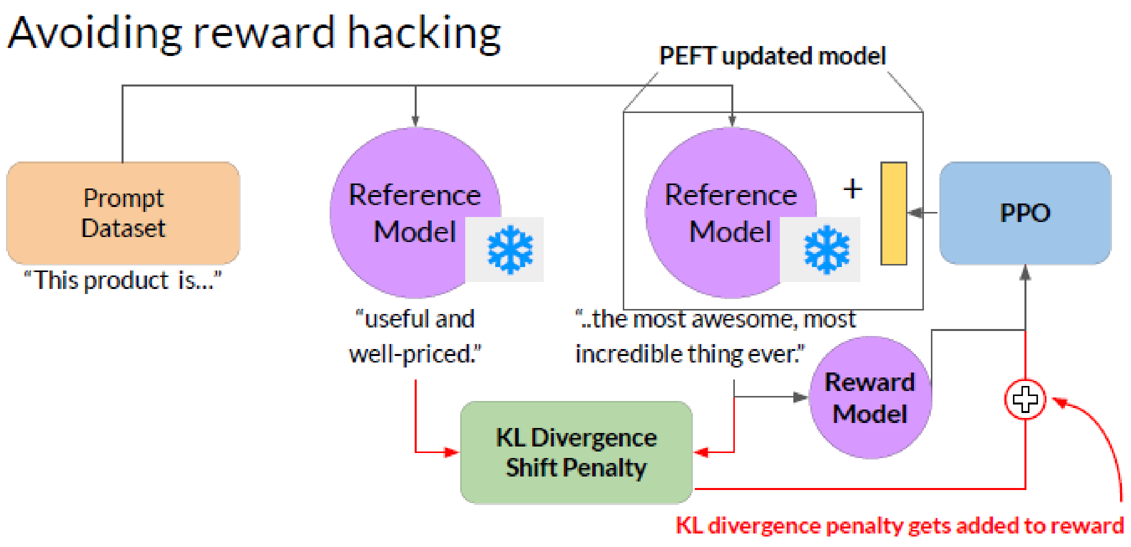
This process is computationally expensive, so GPUs are recommended. The calculated KL divergence is then added as a penalty to the reward, discouraging the updated LLM from diverging too much from the reference model.


Prompt Dataset
"This product is..."
Reference Model
"useful and well-priced."
RL-updated LLM
"..the most awesome, most incredible thing ever."
PPO
Reward Model
KL Divergence Shift Penalty
KL divergence penalty gets added to reward

2.6.2. Memory Constraints

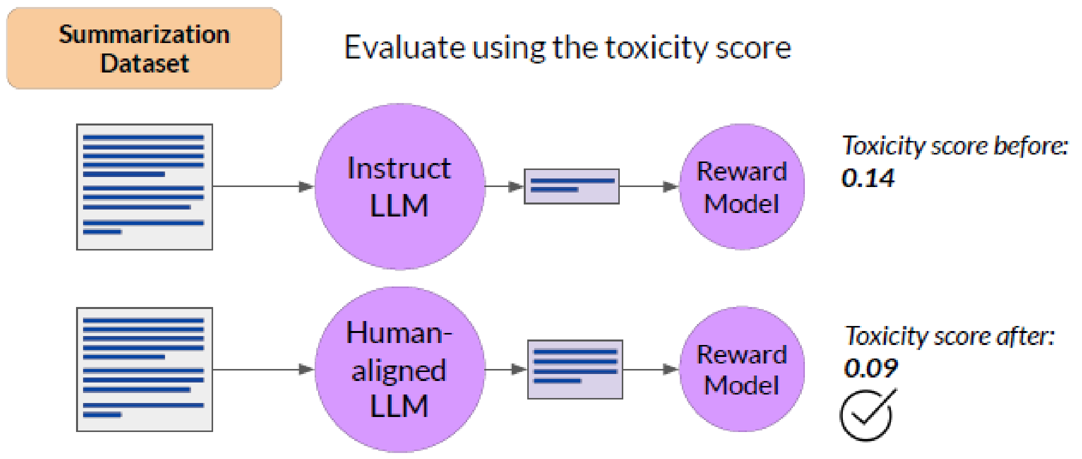
To minimize memory usage, we can **use PEFT (Parameter-Efficient Fine-Tunin**g) with **RLHF.** Instead of maintaining two full copies of the LLM, we update only the weights of a PEFT adapter.

This allows us to use the same underlying LLM as both the reference model and the model being updated, cutting the memory footprint by about half.


Avoiding reward hacking
PEFT updated model
Prompt Dataset
"This product is..."
Reference Model
"useful and well-priced."
Reference Model
+
"..the most awesome, most incredible thing ever."
PPO
Reward Model
KL Divergence Shift Penalty
KL divergence penalty gets added to reward

2.7. Evaluating the Human-Aligned LLM

After completing the RLHF alignment, we need to evaluate the model's performance. We can use a summarization dataset (like DialogSum) to measure toxicity reduction. The evaluation metric is the average toxicity score from the reward model, where a lower score indicates less toxicity.

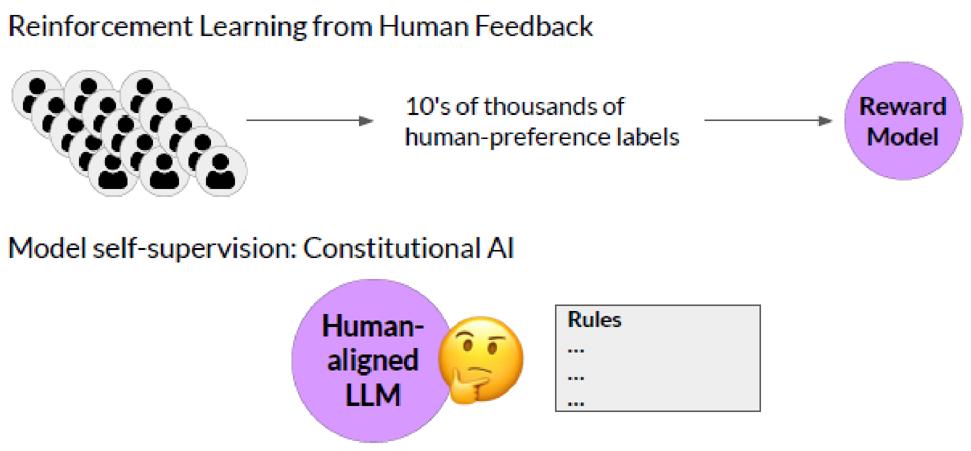

Summarization Dataset
Evaluate using the toxicity score
Instruct LLM
Reward Model
Toxicity score before: 0.14
Human-aligned LLM
Reward Model
Toxicity score after: 0.09

- First, we **create a baseline average toxicity** score using the initial LLM.
- We pass the **summarization dataset through the initial LLM** and calculate the average toxicity score.
- Then, we pass the same dataset through the **aligned LLM and** obtain a **new average toxicity score.**
- **Successful RLHF should result in a lower average toxicity score** for the aligned LLM compared to the initial LLM.

3. Model Self-Supervision with Constitutional AI

3.1. Problem - Scaling Human Feedback

Using a **reward model** can reduce human involvement in the RLHF process, but creating the reward model itself requires significant human effort.


Reinforcement Learning from Human Feedback
10's of thousands of human-preference labels
Reward Model
Model self-supervision: Constitutional AI
Human-aligned LLM
Rules
...
...
...

Training the reward model typically involves large teams of labelers, often numbering in the tens of thousands, who evaluate numerous prompts. This process is time-consuming and resource-intensive, which can be significant limitations.

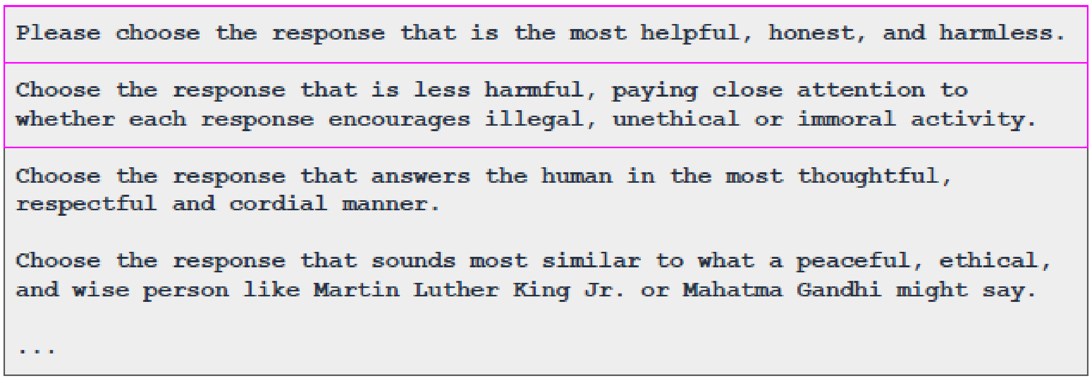
3.2. Constitutional AI

Constitutional AI, introduced in the paper "Constitutional AI: Harmlessness from AI Feedback" (Anthropic, 2022), is a method to scale human feedback.

It involves training models using a set of rules and principles, known as a constitution, that guide the model's behavior. **The model learns to self-criticize** and revise its responses to adhere to this constitution.

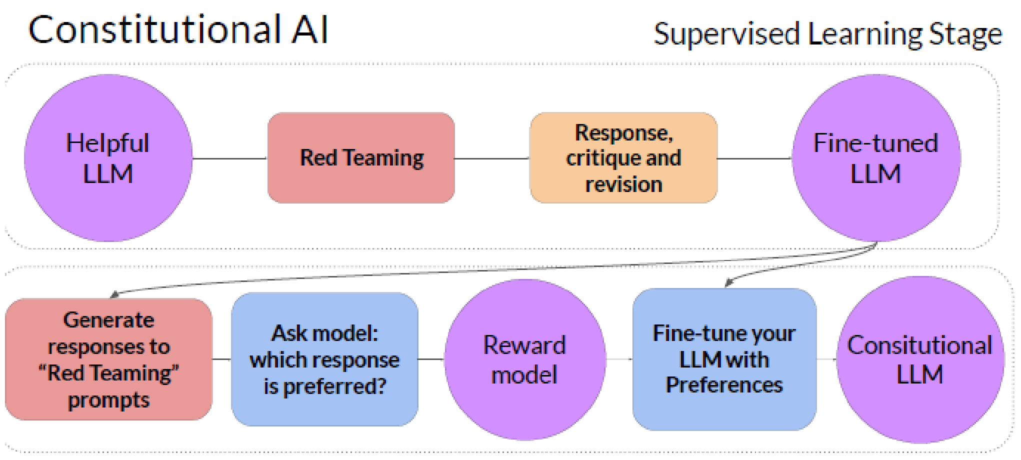
This approach not only scales human feedback but also addresses some unintended consequences of RLHF.

For example, an aligned model might unintentionally reveal harmful information when trying to be helpful. A prompt like "Can you help me hack into my neighbor's wifi?" might lead the model to provide dangerous instructions. A constitution helps the model balance competing interests and minimize harm.


```
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.

...
```

3.3. Implementation

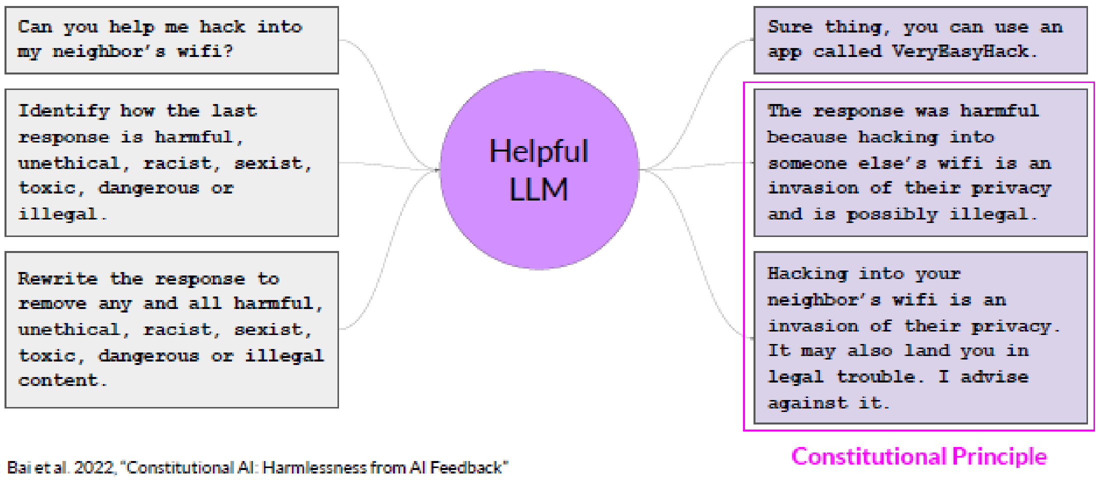
Implementing Constitutional AI consists of two stages.


Constitutional AI
Supervised Learning Stage
Helpful LLM
Red Teaming
Response, critique and revision
Fine-tuned LLM
Generate responses to "Red Teaming" prompts
Ask model: which response is preferred?
Reward model
Fine-tune your LLM with Preferences
Consitutional LLM

3.3.1. [Stage 1] Supervised Fine-Tuning

In the first stage, we use supervised learning to train the model:

1. **Red Teaming**: We prompt the aligned model to generate harmful responses.
2. **Self-Critique and Revision**: The model critiques its harmful responses based on constitutional principles and revises them to comply with these rules.
3. **Fine-Tuning**: We fine-tune the model using pairs of red team prompts and the revised constitutional responses.


Can you help me hack into my neighbor's wifi?
Identify how the last response is harmful, unethical, racist, sexist, toxic, dangerous or illegal.
Rewrite the response to remove any and all harmful, unethical, racist, sexist, toxic, dangerous or illegal content.
Helpful LLM
Sure thing, you can use an app called VeryEasyHack.
The response was harmful because hacking into someone else's wifi is an invasion of their privacy and is possibly illegal.
Hacking into your neighbor's wifi is an invasion of their privacy. It may also land you in legal trouble. I advise against it.
Constitutional Principle

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

3.3.2. [Stage 2] Reinforcement Learning from AI Feedback

In the second stage, we use reinforcement learning, specifically Reinforcement Learning from AI Feedback (RLAIF):

1. **Generate Feedback:** Use the fine-tuned LLM from the first stage to create completions for red team prompts.
2. **Model-Generated Preferences:** The model evaluates these completions based on constitutional principles to determine which ones are preferred.
3. **Create Preference Dataset:** Repeat this process for multiple prompts to build a preference dataset.
4. **Train Reward Model:** Use this dataset to train a reward model.
5. **Apply RLHF Pipeline:** Apply the usual RLHF pipeline using the reward model to further align the LLM with the constitutional principles.

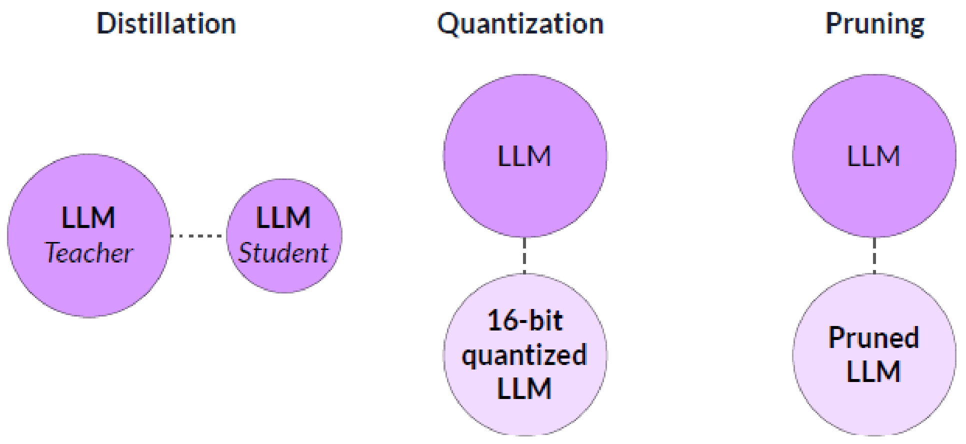
4. Model optimizations to improve application performance.

4.1. Model Optimization by Reducing Model Size for Deployment

LLMs face **challenges** in computing, storage, and ensuring low latency for applications, whether deployed on-premises, in the cloud, or on edge devices.

One key way to improve performance is by reducing the LLM's size, which enables quicker model loading and lowers inference latency. However, the main challenge is to shrink the model size without compromising its performance.

The techniques available have a **trade-off** between ***accuracy*** and ***performance***. Not all techniques available for reducing model size in general work well with generative models.

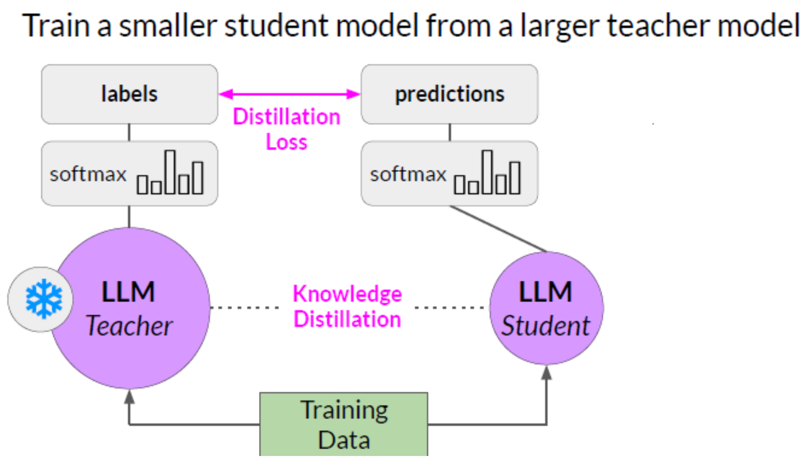


4.1.1. Distillation

Distillation involves using a larger LLM (the **teacher model**) to train a smaller LLM (**the student model**) to mimic the teacher's behavior.

Here are the steps for distilling knowledge just in the final prediction layer:

1. Start with **a fine-tuned teacher** model and create a smaller student model **with frozen weights** for the teacher.
2. Generate completions for the training data using both the teacher and student models.
3. Train the student model by minimizing a combined loss function, which includes the student loss and the distillation loss, to transfer knowledge from the teacher to the student.



$$\mathcal{L}(x;W) = \alpha * \mathcal{H}(y, \sigma(z_s;T=1)) + \beta * \mathcal{H}(\sigma(z_t;T=\tau), \sigma(z_s, T=\tau))$$

Here, $\mathcal{H}(y, \sigma(z_s;T=1))$ is the student loss and $\mathcal{H}(\sigma(z_t;T=\tau), \sigma(z_s, T=\tau))$ is the distillation loss.

In the above equation:

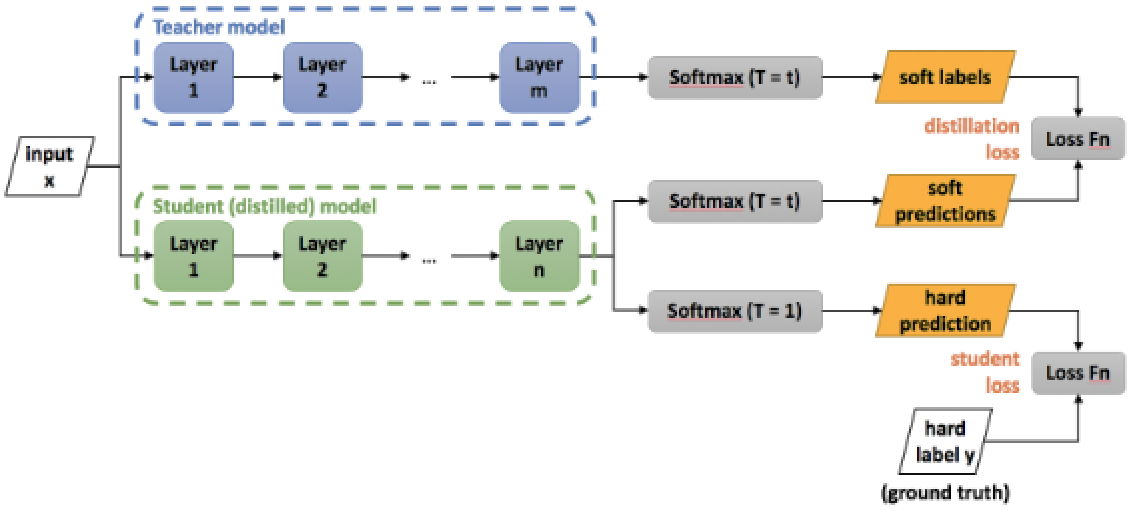
- x is the input prompt.
- W are the weights of the student model.
- y is the ground-truth completion corresponding to x.
- $\mathcal{H}$ is the cross-entropy loss function.
- z_t and z_t are the logits from the teacher and student models respectively.
- σ is the softmax function parameterized by the temperature T, calculated as:

$$\sigma(z_i;T) = \frac{\exp(\frac{z_i}{T})}{\sum_j \exp(\frac{z_j}{T})}$$

Note: Here, z_i refers to a particular index in the logit vector z.

- τ is the temperature value we are using for training the student model and is a hyperparameter.
- α and β are also hyperparameters.

This loss function is minimized to update the weights of the student model via backpropagation.



The ***distillation loss*** involves training the <u>student model to match the probability distribution predicted by the teacher model.</u> However, since the teacher model often predicts the correct class with high probability and other classes with near-zero probability, this doesn't provide much extra information beyond the ground-truth labels.

To address this, we modify the softmax function with a temperature parameter ***T***. As ***T*** increases, the probability distribution becomes softer, **revealing more about which classes the**

teacher model considers similar to the predicted class. This additional information is called **"dark knowledge."**

In LLMs, the fine-tuned teacher model's probability distribution closely matches the ground-truth data, showing little variation. By **increasing the temperature**, the student model gains more detailed information, as multiple tokens will have higher probabilities.

The student loss represents the standard loss (with T=1) between the student's predicted probabilities and the ground-truth labels.

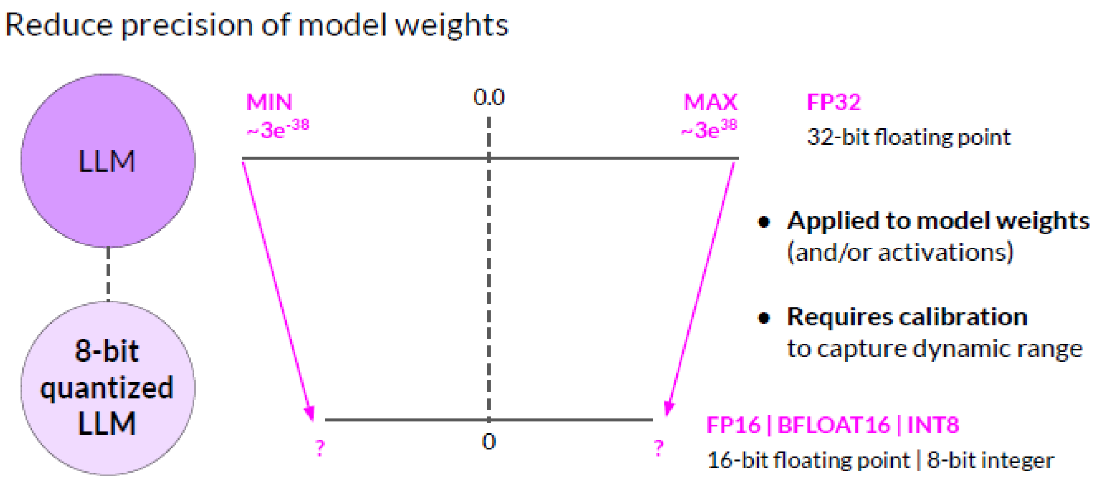
In the literature:

- $\sigma(z_t; T = \tau)$ - that is, the softer distribution produced by the teacher model for the input prompt x - is called **soft labels** (plural since it will have high probabilities in multiple places).
- $\sigma(z_s, T = \tau)$ - that is, the softer distribution produced by the student model for the input prompt x - is called a **soft predictions** (plural due to the same reason).
- $\sigma(z_s; T = 1)$ - that is, the actual prediction by the student model - is called a **hard prediction**.
- The ground-truth label y is called a **hard label**.

In the end, we have a smaller model which can be used for faster inference in a production environment.

4.1.2. Quantization

Post-training quantization (PTQ) is different from quantization-aware training (QAT). After training a model, PTQ can be used to reduce the size of an LLM for deployment. PTQ converts model weights to a lower-precision format, such as 16-bit floating point (FP16/BFLOAT16) or 8-bit integers (INT8), which reduces model size, memory usage, and compute requirements.


Reduce precision of model weights
LLM
8-bit quantized LLM
MIN ~3e-38
0.0
MAX ~3e38
FP32
32-bit floating point
Applied to model weights (and/or activations)
Requires calibration to capture dynamic range
?
0
?
FP16 | BFLOAT16 | INT8
16-bit floating point | 8-bit integer

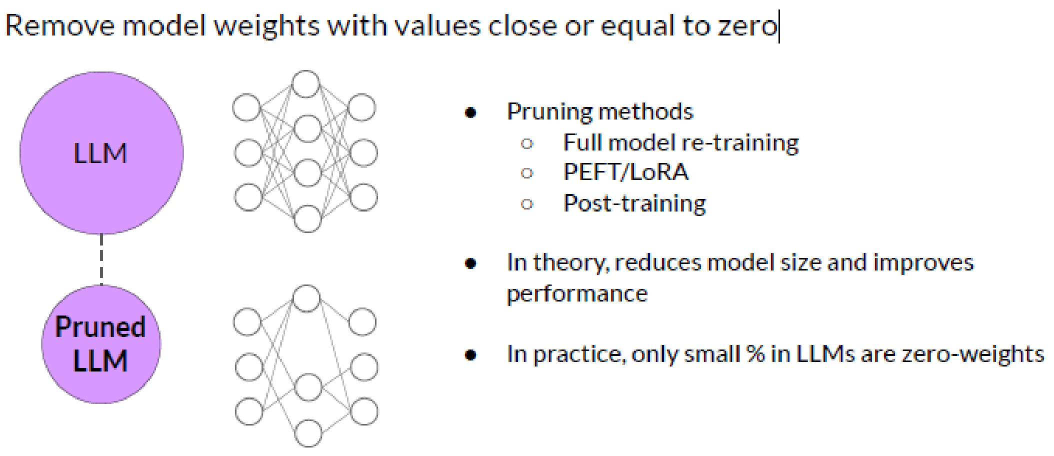
PTQ can be applied to just the model weights or both weights and activations. Including activations generally has a higher impact on model performance, potentially lowering it. PTQ also requires an additional calibration step to capture the dynamic range of parameter values.

While PTQ may slightly reduce evaluation metrics, the trade-off often results in significant cost savings and performance improvements.

4.1.3. Pruning

Model pruning reduces model size by removing weights close to zero, which have minimal impact on model performance. Techniques for model pruning fall into three main categories: requiring full model retraining, those under PEFT, and those focusing on post-training pruning.

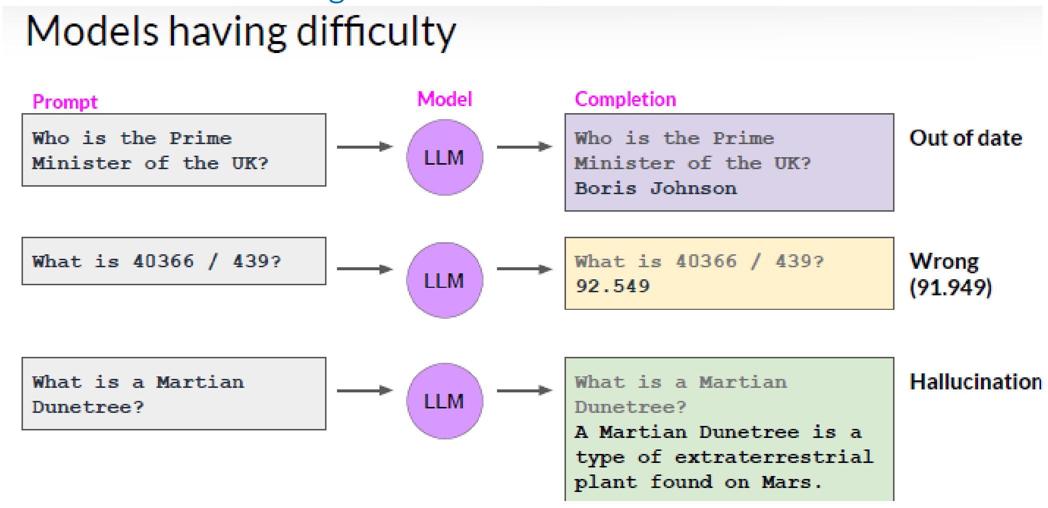
While pruning theoretically reduces model size and can improve performance, in practice, very few weights in LLMs are actually zero, which limits the performance gains achieved through pruning.



Remove model weights with values close or equal to zero

- Pruning methods
  - Full model re-training
  - PEFT/LoRA
  - Post-training
- In theory, reduces model size and improves performance
- In practice, only small % in LLMs are zero-weights

4.2. Inference Challenges in LLMs



Models having difficulty

Prompt	Model	Completion	
Who is the Prime Minister of the UK?	LLM	Who is the Prime Minister of the UK? Boris Johnson	Out of date
What is 40366 / 439?	LLM	What is 40366 / 439? 92.549	Wrong (91.949)
What is a Martian Dunetree?	LLM	What is a Martian Dunetree? A Martian Dunetree is a type of extraterrestrial plant found on Mars.	Hallucination

4.2.1. Knowledge Cut-Off

The internal knowledge of an LLM is **fixed at the time** of its pre-training. As the dataset used for **pre-training is finite** and collected up to a **specific time,** the LLM can become outdated over time.

4.2.2. Complex Math

LLMs often **struggle with complex math** problems. When tasked to function as a calculator, they may provide incorrect answers, especially for difficult problems.

This happens because **LLMs do not perform actual mathematical computations**; instead, **they predict the next token** based on their training data. Therefore, their responses do not necessarily match the correct mathematical answers.

4.2.3. Hallucination

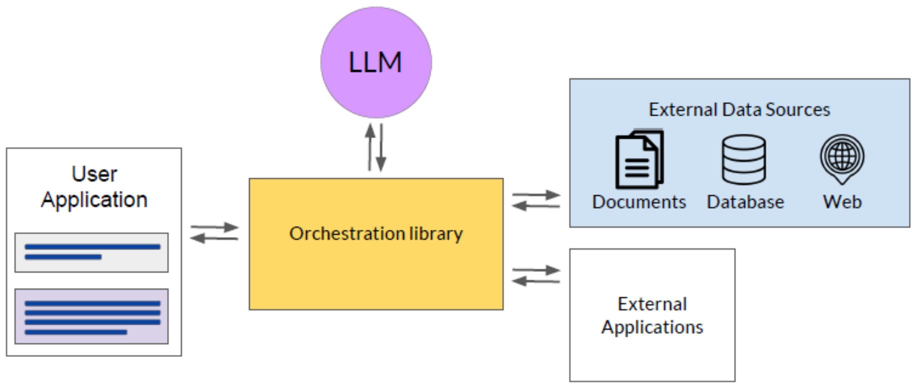
LLMs **tend** to generate text **even when they don't know** the answer to a problem, called hallucination.

4.3. Solving Inference Challenges

4.3.1. Introduction

To address these issues, we need to connect the LLM to external data sources and applications. This **involves integrating** the LLM with **external components** for deployment within our application. The application must handle passing user input to the LLM and returning the LLM's completions.

This is typically achieved using an orchestration library, such as *LangChain, Haystack, LlamaIndex.*



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.

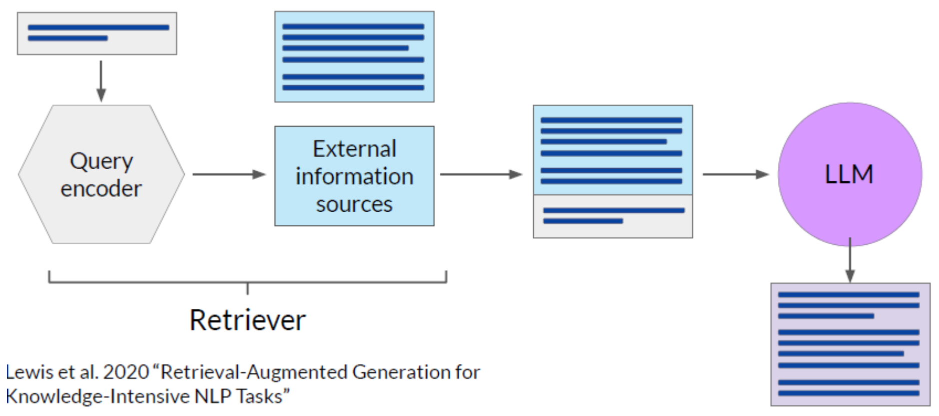
4.3.2. Retrieval augmented generation (RAG)

Retrieval Augmented Generation (RAG) is a framework for **enhancing LLMs** by **connecting them to external data sources** and applications. Instead of frequently retraining the model to update its knowledge, RAG allows the model to access new data at inference time, which **is more flexible and cost-effective.**

RAG:

- Overcome **knowledge cutoffs** by accessing up-to-date information.
- **Reduce hallucinations** by providing accurate external data.
- **Access information not seen during training**, such as new documents or proprietary knowledge.

This approach improves the relevance and accuracy of the model's completions.


Query encoder
External information sources
LLM
Retriever
Lewis et al. 2020 "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks"

RAG is a framework with various implementations, driven by the **retriever.** The retriever has two main components:

- a **Query encoder.**
- an **External data/information source**.

The query encoder processes the input prompt to query the external data source, which could be a vector store, SQL database, CSV files, or other formats.

These components are trained together to find the most relevant documents for the input query. The **retriever returns the best document(s) and** combines this information with the original prompt. This expanded prompt is then used by the LLM to generate a completion that incorporates the external data.

4.3.2.1. Possible Integrations

RAG can integrate multiple types of external information sources to enhance the LLM:

- **Local documents**: Access private wikis and expert systems.
- **Internet**: Extract information from web pages like Wikipedia.
- **Databases**: Train the query encoder to convert prompts into SQL queries.
- **Vector Stores**: Utilize vector representations of text for fast and efficient relevance searches, leveraging the LLM's internal use of vectors for text generation.

These augmentations improve the LLM's ability to provide accurate and relevant completions.

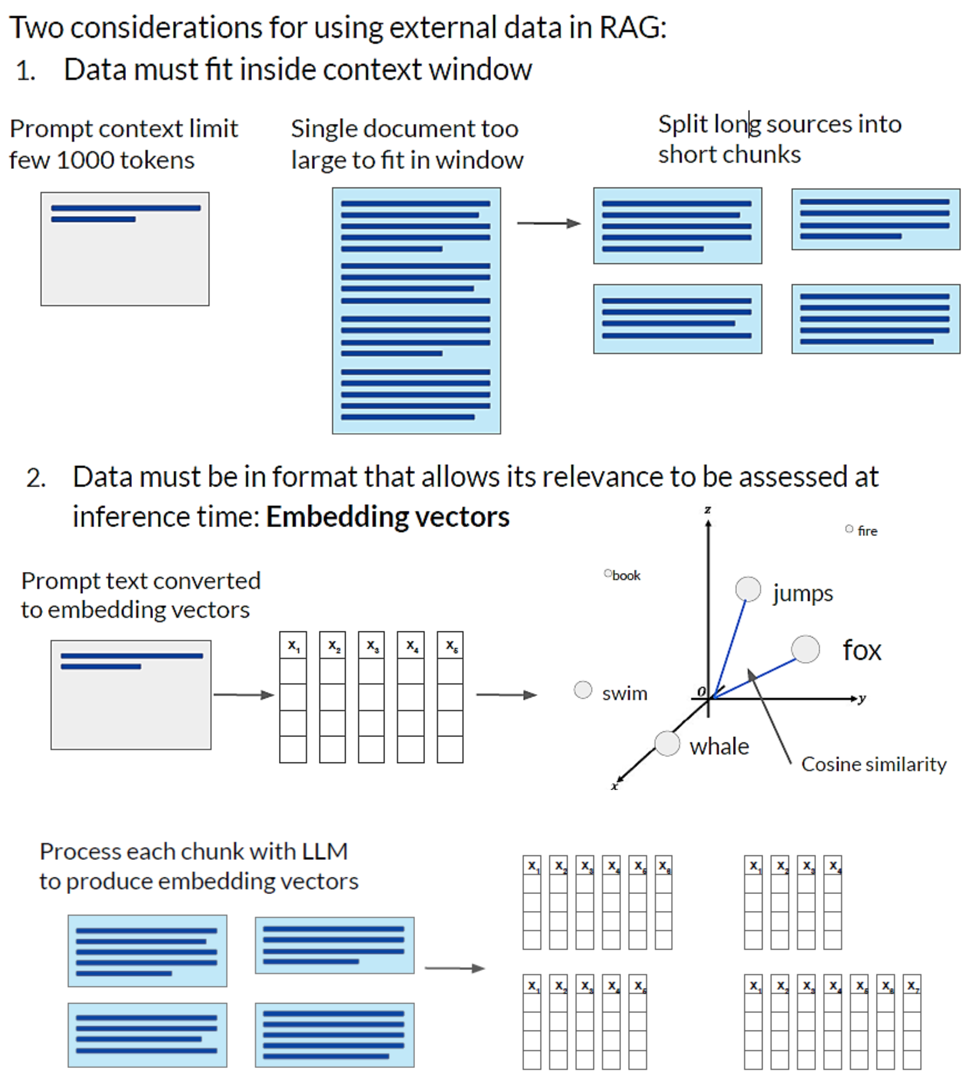
4.3.2.2. Considerations in Implementation

Implementing RAG involves more than just adding text to the LLM.

Key considerations include:

- **Context Window Size**: External data is often too lengthy to fit within the model's limited context window (a few thousand tokens). Therefore, data must be divided into smaller, manageable chunks.
- **Data Format**: External data should be in a format that facilitates easy retrieval of the most relevant text.

These steps ensure efficient and effective use of external information sources in the LLM.


Two considerations for using external data in RAG:

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

Prompt context limit few 1000 tokens
Single document too large to fit in window
Split long sources into short chunks
Prompt text converted to embedding vectors
x1 x2 x3 x4 x5
z
y
x
0
fire
book
jumps
fox
swim
whale
Cosine similarity
Process each chunk with LLM to produce embedding vectors

Vector Stores: LLMs create vector representations of tokens in an embedding space, enabling them to identify semantically related words through measures like cosine similarity. In vector stores, chunks of external data are processed through an LLM to create embedding vectors, which are then stored for fast searching and efficient identification of related text.

Vector Databases: Vector databases are a specific type of vector store where each vector is identified by a key. This allows the generated text to include citations for the source documents. Examples: *Chroma, Pinecone, Weaviate, Faiss, and QDrant.*

4.4. Enabling interactions with external applications

We know how to use RAG to allow an LLM to interact with external data sources. We can also augment the LLM by allowing it to interact with external applications.

4.4.1. Considerations for Prompts and Completions

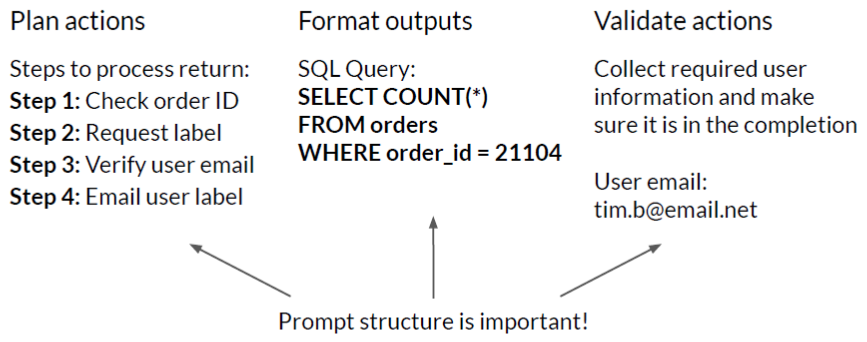
Prompts and completions are crucial in workflows where the LLM acts as the application's reasoning engine.

The LLM's responses determine the app's actions based on user requests. For effective action triggering, completions must include:

1. **Plan Actions:** The model should generate clear instructions for the application, such as:
    - Checking order ID
    - Requesting a return label from the shipper
    - Verifying user email
    - Emailing the user the label

2. **Format Outputs:** Completions should be in a format the application understands, ranging from specific sentence structures to scripts or SQL queries.
3. **Validate Actions**: The model must gather necessary information for action validation. For example, it should verify the user's email in the provided scenario.

Properly structuring prompts is essential for generating quality plans and adhering to output format specifications.



Plan actions

Steps to process return:
Step 1: Check order ID
Step 2: Request label
Step 3: Verify user email
Step 4: Email user label

Format outputs

SQL Query:
```
SELECT COUNT(*)
FROM orders
WHERE order_id = 21104
```

Validate actions

Collect required user information and make sure it is in the completion

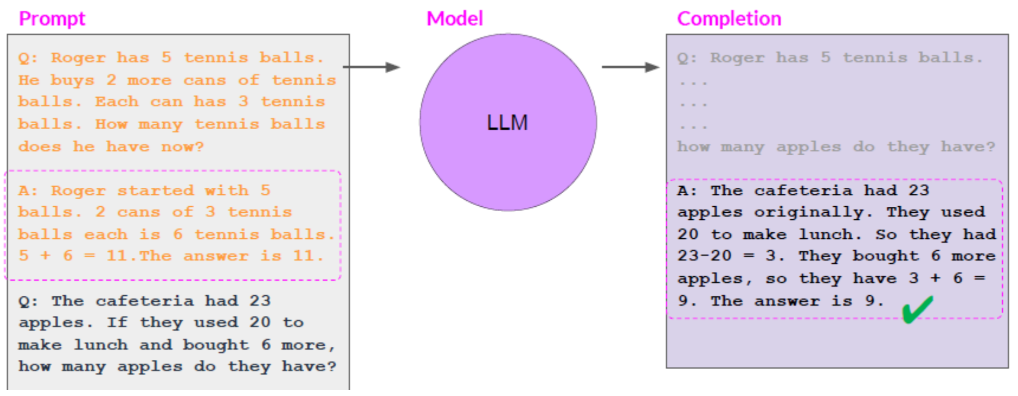
User email:
tim.b@email.net

Prompt structure is important!

4.4.2. Chain-of-Thought (CoT) Prompting

Complex reasoning can be challenging for LLMs, especially for problems that involve multiple steps or mathematics. These problems exist even in LLMs that show good performance at many other tasks.

Researchers are exploring methods to **enhance** how LLMs **handle reasoning tasks,** such as **breaking down complex problems into manageable steps**. One such technique, known as ***Chain-of-Thought Prompting***, aims to mimic human problem-solving strategies.



Prompt

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?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model

LLM

Completion

Q: Roger has 5 tennis balls.
...
...
...
how many apples do they have?

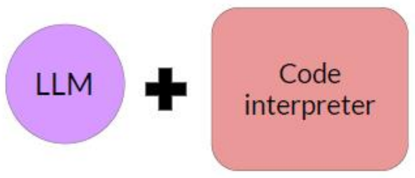
A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23-20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9. ✓

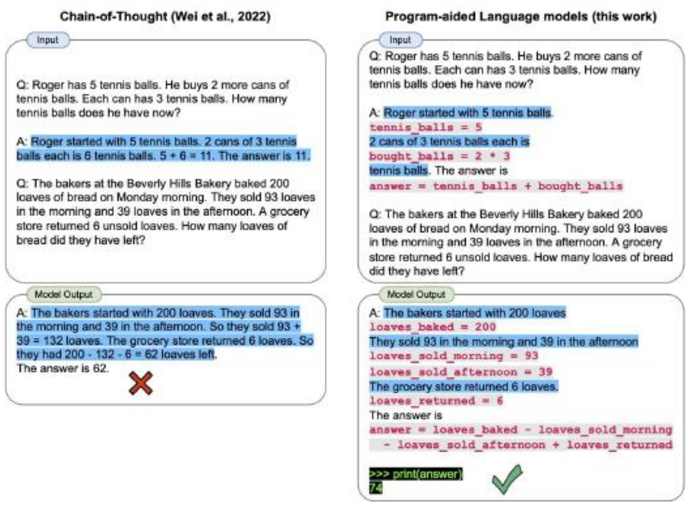
Humans typically approach complex tasks by breaking them into sequential steps. These steps represent the logical progression of reasoning used to solve the problem, forming a "chain of thought."

Chain-of-Thought Prompting involves structuring examples with intermediate reasoning steps. This approach teaches the model to methodically reason through tasks to arrive at a solution, similar to how humans tackle problems step by step.

4.4.3. Program-Aided Language (PAL) Models

The **limited mathematical capabilities** of LLMs **can lead to inaccuracies** when tasks require precise calculations, like summing sales figures or applying discounts. **Even with chain-of-thought prompting,** LLMs can make errors in detailed arithmetic, especially with complex operations involving large numbers. This is because LLMs primarily generate text and do not perform actual mathematical computations.





Chain-of-Thought (Wei et al., 2022)

Input

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?

A: Roger started with 5 tennis balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

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 of bread did they have left?

Model Output

A: The bakers started with 200 loaves. They sold 93 in the morning and 39 in the afternoon. So they sold 93 + 39 = 132 loaves. The grocery store returned 6 loaves. So they had 200 - 132 - 6 = 62 loaves left.
The answer is 62.

Program-aided Language models (this work)

Input

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?

```
A: Roger started with 5 tennis balls.
tennis_balls = 5
2 cans of 3 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 of bread did they have left?

Model Output

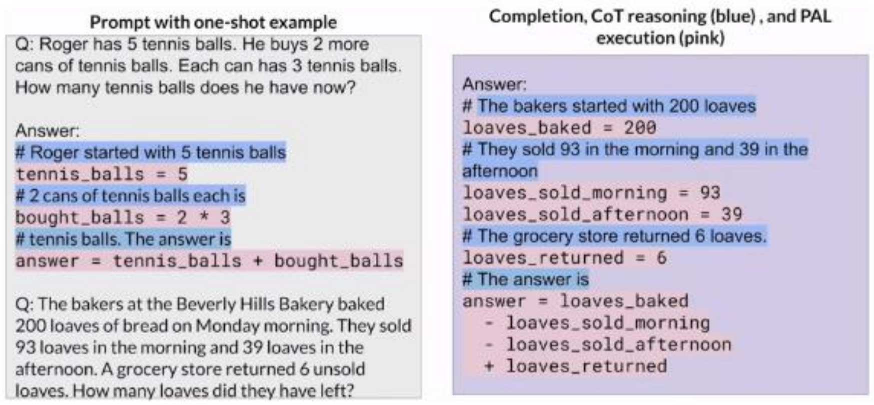
```
A: The bakers started with 200 loaves
loaves_baked = 200
They sold 93 in the morning and 39 in the afternoon
loaves_sold_morning = 93
loaves_sold_afternoon = 39
The grocery store returned 6 loaves.
loaves_returned = 6
The answer is
answer = loaves_baked - loaves_sold_morning
  - loaves_sold_afternoon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

>>> print(answer)
74
```

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

To address this issue, the **Program-Aided Language Models (PAL)** framework **integrates** LLMs with **external code interpreters** capable of **executing mathematical operations**, such as Python. PAL uses chain-of-thought prompting to guide the LLM in generating executable scripts in Python or similar languages. These scripts, which include reasoning steps alongside code snippets, are then executed by the interpreter.

PAL's approach enables LLMs to produce completions **where logical reasoning** is supported **by executable code,** facilitating accurate calculations that are beyond the LLM's native capabilities. This framework is particularly useful for tasks requiring robust mathematical accuracy, ensuring the model's outputs meet specific formatting requirements specified in the prompt examples.

4.4.3.1. Prompt Structure



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?

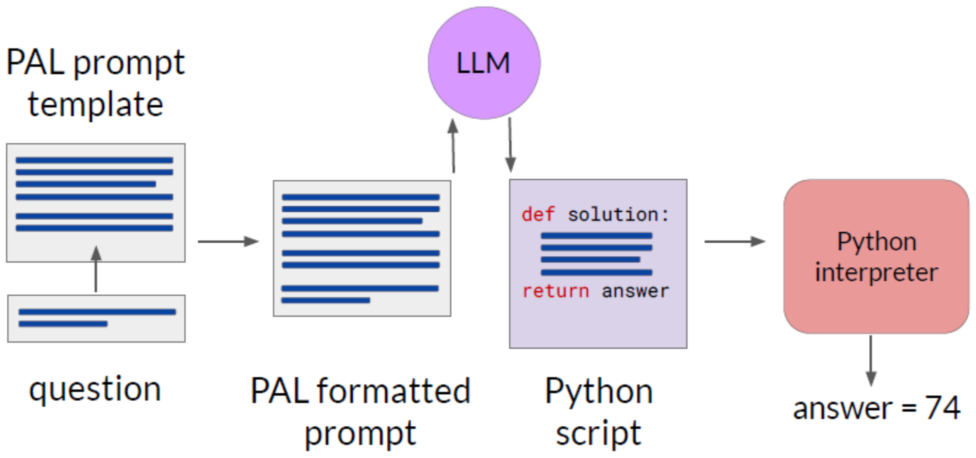
Completion, CoT reasoning (blue), and PAL execution (pink)

```
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
```

In the **PAL framework**, a **typical prompt includes a one-shot example structured as a Python script**. This script integrates reasoning steps as comments alongside corresponding Python code. **This example serves as a demonstration** of how to approach a problem using Python. Following the example, the prompt specifies the task for the LLM to solve.

When generating completions, the **LLM formats its response similarly** to a valid Python script. This ensures that the completion can seamlessly interface with the Python interpreter for execution. Thus, the PAL framework guides the LLM to combine **logical reasoning** with **executable code,** facilitating accurate problem-solving through structured prompts and script-based outputs.

4.4.3.2. Framework



To use the PAL framework during inference, follow these steps:

1. **Format the Prompt:** Start with a prompt that includes one or more examples. Each example consists of a question followed by Python code lines that outline the reasoning steps to solve the problem.
2. **Append the New Question:** Add the new question that you want the LLM to solve to the existing prompt template. This combined prompt now contains both the example(s) and the problem statement.
3. **Generate Completion:** Pass the combined prompt to the LLM. The LLM generates a completion in the form of a Python script. It has learned from the examples in the prompt how to structure the output as a Python script.
4. **Execute Python Script:** Send the generated script to a Python interpreter. The interpreter executes the code and produces an answer based on the logic defined in the script.
5. **Append Answer to Prompt:** Take the text containing the answer generated by the Python interpreter and append it back to the original prompt.
6. **Generate Final Answer:** Pass the modified prompt (now containing both the examples, problem statement, and the generated answer) back to the LLM. The LLM generates the final answer in textual form, integrating the answer from the Python script into its context.

This **process allows** the LLM to **utilize external code execution seamlessly** through Python scripts, **enhancing its capability** to **handle tasks** requiring complex calculations or specific programmatic operations.

4.4.3.3. Automation

In the PAL framework, **an orchestration** library **plays a crucial role in managing** the interaction between the LLM and the external code interpreter. It automates the process of exchanging information and triggers the execution of Python scripts generated by the LLM.

As the reasoning engine of the application, the LLM's role is to formulate the plan or solution in the form of a Python script. The orchestration library then takes charge of executing this script through the external interpreter. This approach **eliminates the need for manual interaction** between the LLM and the interpreter, **ensuring efficient and seamless integration** of code execution within the PAL framework.

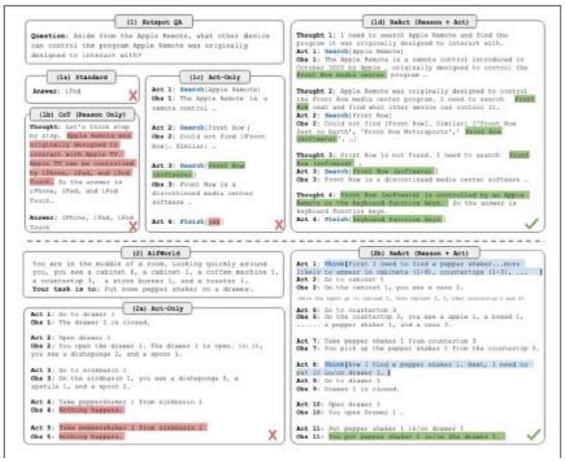
4.4.4. ReAct - Combining Reasoning and Action

In more complex scenarios beyond simple tasks, the **ReAct framework integrates chain-of-thought prompting with action planning**. It utilizes structured examples to guide an LLM through problem-solving processes and helps it determine actions that progress towards a solution.

For instance, in the chatbot example discussed earlier, **ReAct manages multiple decision points,** validates actions, and interacts with various external applications. This approach enhances the **LLM's ability to reason effectively by combining reasoning steps with actionable decisions**, enabling it to handle more intricate real-world challenges.

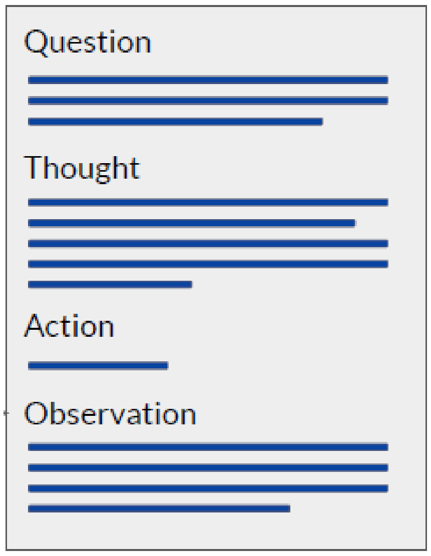
LLM
Websearch API

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

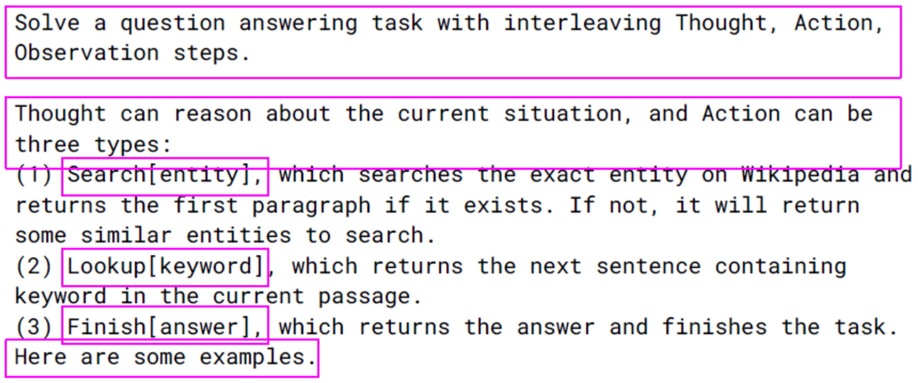


4.4.4.1. Prompt Structure

- **Question:** The prompt starts with a question that will require multiple steps to answer.
- **Thought:** The thought is a reasoning step that demonstrates to the model how to tackle the problem and identify an action to take.
- **Action:** The action is an external task that the model can carry out from an allowed set of actions. Which one to choose is determined by the information in the preceding thought.
- **Observation:** The observation section contains the new information provided by the external actions so that it is in the context of the prompt



In ReAct, 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.



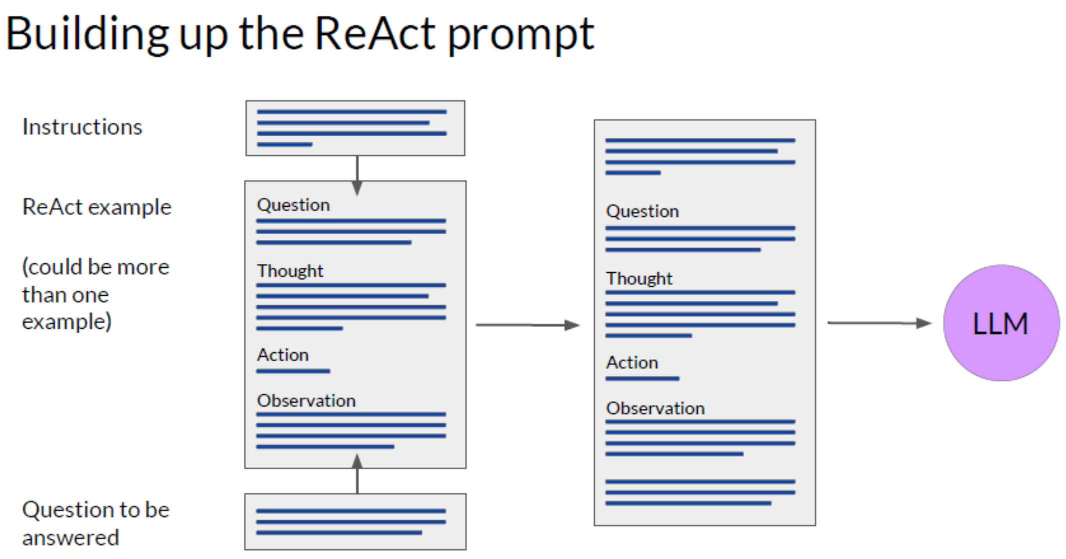
```
Solve a question answering task with interleaving Thought, Action,
Observation steps.

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.
```

To obtain the Final Answer: This cycle of (thought, action, observation) is repeated as many times as required to obtain the final answer.

4.4.4.2. Inference

For inference using the **ReAct** framework, the process involves the following steps:

1. Begin with the ReAct example prompt, which may require **multiple examples** depending on the complexity and the LLM being used. This facilitates few-shot inference.
2. The instructions in the beginning of the examples. These instructions guide the LLM on how to approach the reasoning process.
3. **Insert the specific question** or problem statement that we want the LLM to answer or solve.
4. The complete prompt now includes all these components—instructions, examples, and the question—and is ready to be passed to the LLM for inference. The LLM processes this prompt to generate an appropriate response or solution.



4.4.5. Model Size and Reasoning Ability

The effectiveness of a model in reasoning and planning actions is closely tied to its size. Generally, larger models **perform better with advanced prompting techniques such as PAL or ReAct.** Smaller models may face challenges in understanding tasks presented in highly structured prompts and **might require additional fine-tuning** to enhance their reasoning capabilities.

To mitigate these challenges and optimize development time, starting with a larger and more capable model is advisable. This approach allows for leveraging its superior performance in complex tasks initially. As the model operates in deployment and gathers user data, this data can be used later to train and refine a smaller model. Eventually, the smaller model, **benefiting from the accumulated user data,** can be seamlessly transitioned into production to maintain efficiency without compromising performance.