

RL without Tears:

Chapter 3: Understanding RL in LLM Training

Chenglong Wang
Northeastern University, China

WANGCHENGLONG@MAIL.NEU.EDU.CN

Hang Zhou
Northeastern University, China

STCEUM@GMAIL.COM

Tongran Liu
Institute of Psychology, Chinese Academy of Sciences, China

LIUTR@PSYCH.AC.CN

Jingbo Zhu
Northeastern University, China

ZHUJINGBO@MAIL.NEU.EDU.CN

Tong Xiao
Northeastern University, China

XIAOTONG@MAIL.NEU.EDU.CN

3 Understanding RL in LLM Training

We continue with a practical example of using an LLM as a homework assistant. In this context, we want to improve the capability of an SFT LLM to handle education-related inputs more effectively. Suppose we have a homework assistant powered by an SFT LLM, and a student types the input “Give me three tips to improve my accuracy in solving math problems.” A SFT LLM typically generates a short output, such as

There are three tips for improving accuracy in solving math problems:

1. Practice regularly.
2. Understand the concepts.
3. Double-check your work.

Although this output adheres to the given input, it is very short and not sufficiently detailed or comprehensive for this scenario. In practice, the “short” feature in the generated output stems from the training approach of the SFT LLM. During SFT, the LLM learns from a large set of labeled samples that typically contain concise and factual answers to common inputs. These samples guide the LLM to prioritize brevity and clarity, so it often generates short outputs, like the one shown above. Of course, we could annotate enough additional data to fine-tune the LLM further and adjust it to generate the desired outputs. However, this approach is limited in its ability to scale. For example, in this scenario, the input involves a variety of potential answers, and the expectations of the student can be pretty diverse. Therefore, describing the “ideal” output would require immense annotation effort. Consequently, collecting or annotating fine-tuning data is not as straightforward as it is with SFT, particularly when attempting to cover the breadth of possible student inputs and outputs. Instead, we can use RL to enable the model to discern outputs that better align with human preferences, such as generating more detailed and comprehensive content that not only adheres to the given input but also meets the expectations of the student in this scenario.

In the following sections, we use a specific LLM training example to explain how RL works, where the LLM is trained to generate a longer output in the context of a homework assistant. Throughout this example, we introduce key RL algorithms and their improvements.

3.1 Policy Gradient

In this section, we aim to lengthen the LLM output for the “give me three tips” input using a classical RL algorithm called **policy gradient**. To define our optimization objective, let $R(\cdot)$ be the reward function that describes the goal the algorithm aims to optimize. In this scenario, the reward is designed to align the output of the LLM with a specific behavior, that is, generating longer outputs. Therefore, we define the reward function based on a length-related metric. More specifically, the reward can be proportional to the length of the output, that is, longer outputs can receive higher rewards:

$$R(\mathbf{y}) = \frac{\text{Length}(\mathbf{y})}{10} \quad (1)$$

where $\text{Length}(\cdot)$ indicates the length of the given output. We can use the number of tokens in the output as the length for simplicity. This allows us to assign an immediate reward r_t for the t -th token generated by the LLM, which is $\frac{1}{10}$.

Next, we apply RL to encourage the LLM to generate longer outputs based on the reward function. First, let us review the optimization objective of RL: learning a policy that maximizes the agent’s cumulative reward (or return) over the long term. In this case, the agent can be defined as the LLM itself, and the policy can be defined as the probability distribution over the tokens the LLM predicts, given the preceding context tokens, i.e., $\Pr_\theta(\cdot|\mathbf{x})$ ¹.

The optimization objective can be given by

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \Omega} \Pr_\theta(\mathbf{y}|\mathbf{x}) R(\mathbf{y}) \right] \\ &= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \Omega} \Pr_\theta(\mathbf{y}|\mathbf{x}) \sum_{t=1}^T r_t \right] \end{aligned} \quad (2)$$

where $\mathcal{S}_x$ indicates the input-only dataset, $\mathbf{y} \in \Omega$ indicates that output $\mathbf{y}$ is drawn from the hypothesis space Ω , and T indicates the length of $\mathbf{y}$. $J(\theta)$ is also called the performance function. Then the training objective is to maximize $J(\theta)$:

$$\hat{\theta} = \arg \max_{\theta} J(\theta) \quad (3)$$

Note that the hypothesis space Ω is typically very large, as LLMs often have large vocabularies². Therefore, it is not feasible to enumerate all possible outputs directly. Instead, we typically use a Monte Carlo method to estimate this expectation. This approach involves sampling outputs $\{\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_d\}$ from the LLM and using those samples to approximate the hypothesis space, denoted by $\mathcal{D}$:

$$J(\theta) = \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \mathcal{D}} \Pr_\theta(\mathbf{y}|\mathbf{x}) \sum_{t=1}^T r_t \right] \quad (4)$$

¹To maintain consistency with notations in RL, this policy is also often represented as $\pi_\theta(\cdot|\mathbf{x})$ in much of the related literature. Here, from the perspective of LLM research, we opt to use the probability notation $\Pr_\theta(\cdot|\mathbf{x})$, which is more familiar to researchers in this field.

²The hypothesis space is large because LLMs typically have vast vocabularies and complex tokenization schemes. The number of possible combinations of tokens, even for a modest length output, grows exponentially, leading to a very large output space. This makes exhaustive enumeration of all possible outputs computationally infeasible.

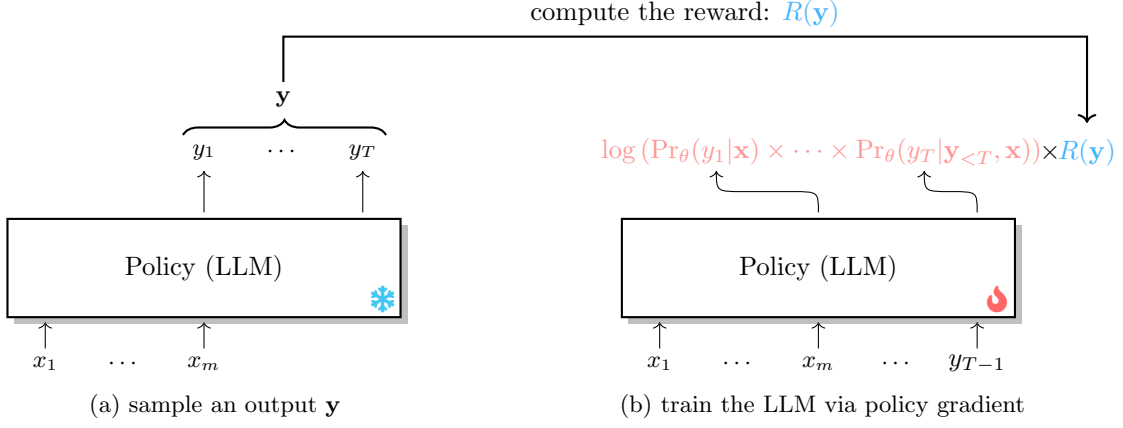


Figure 1: Illustration of training an LLM using policy gradient. The central component is an LLM. Given an input $\mathbf{x} = x_1 \dots x_m$, we sample an output $\mathbf{y}$ from the LLM. Note that gradients are not computed during the sampling to prevent extensive memory consumption because the sampling is performed via an autoregressive mode. Instead, by leveraging the parallel computation capability of the LLM, similar to SFT, the sampled output $\mathbf{y}$ is used for a subsequent forward pass to compute the gradients produced during the generation of each token.

We can further refine the process when optimizing the objective using gradient descent. First, we compute the gradient of $J(\theta)$ with respect to θ :

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta} &= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \mathcal{D}} \frac{\partial \Pr_{\theta}(\mathbf{y}|\mathbf{x}) R(\mathbf{y})}{\partial \theta} \right] \\
&= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \mathcal{D}} \frac{\partial \Pr_{\theta}(\mathbf{y}|\mathbf{x})}{\partial \theta} R(\mathbf{y}) \right] \\
&= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \mathcal{D}} \Pr_{\theta}(\mathbf{y}|\mathbf{x}) \frac{\partial \Pr_{\theta}(\mathbf{y}|\mathbf{x}) / \partial \theta}{\Pr_{\theta}(\mathbf{y}|\mathbf{x})} R(\mathbf{y}) \right] \\
&= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \mathcal{D}} \Pr_{\theta}(\mathbf{y}|\mathbf{x}) \frac{\partial \log \Pr_{\theta}(\mathbf{y}|\mathbf{x})}{\partial \theta} R(\mathbf{y}) \right] \tag{5}
\end{aligned}$$

Using a Monte Carlo sample average over d outputs in $\mathcal{D}$, we can simplify Eq. (5) and need only consider the terms $\frac{\partial \log \Pr_{\theta}(\mathbf{y}|\mathbf{x})}{\partial \theta}$ and $R(\mathbf{y})$:

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\frac{1}{d} \sum_{\mathbf{y} \in \mathcal{D}} \frac{\partial \log \Pr_{\theta}(\mathbf{y}|\mathbf{x})}{\partial \theta} R(\mathbf{y}) \right] \tag{6}$$

Now, as illustrated in Figure 1, we have an RL approach to optimize the output of the LLM: 1) we sample inputs from the input-only dataset and sample outputs from the LLM; then, 2) we evaluate each sampled output using a reward function that we define; then, 3) we update the LLM using gradient descent, following Eq. (6), to maximize the performance function.

We can understand this optimization process from the perspective of hypothesis space reranking, as illustrated in Figure 2. The objective is to increase the probability of longer outputs in the hypothesis space

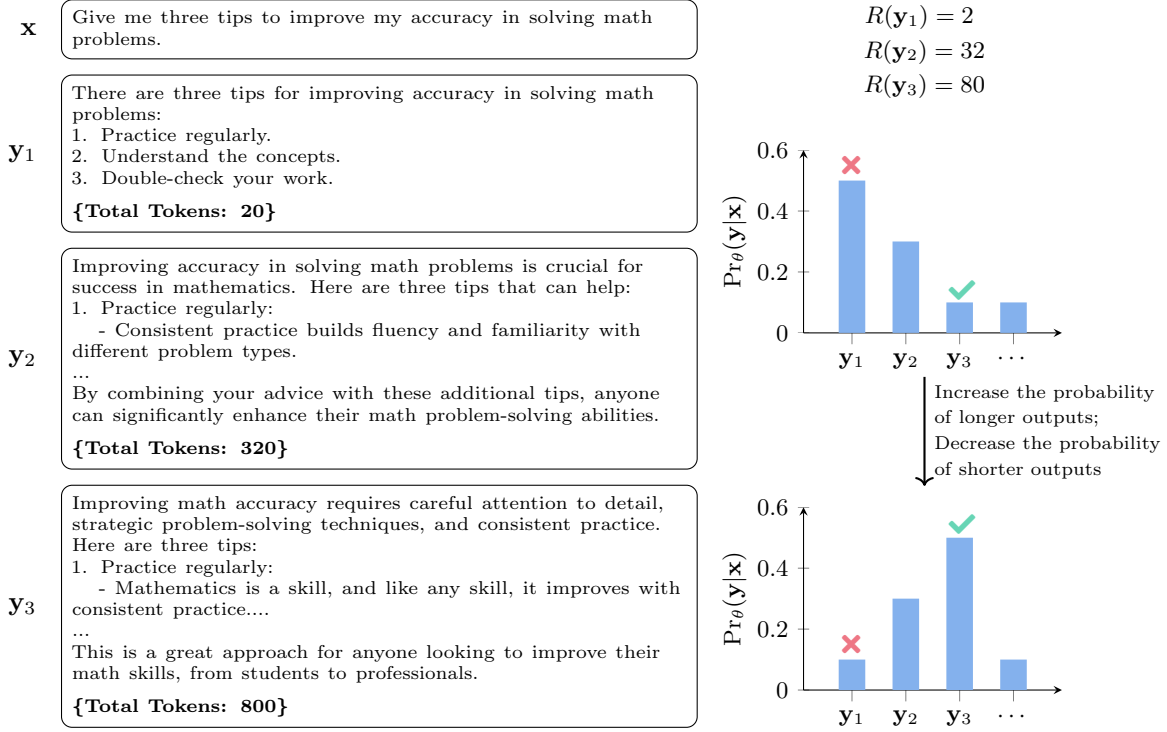


Figure 2: The mechanism of RL in training LLMs from the perspective of hypothesis space reranking: given an input x , the model increases the probability of longer outputs, such as y_3 , within the hypothesis space by assigning higher rewards to them; conversely, shorter outputs like y_1 receive lower rewards, which reduces their probabilities.

by assigning a high reward to those outputs, while simultaneously decreasing the probability of shorter outputs by assigning them a lower reward. In other words, the policy gradient approach encourages the model to generate longer outputs by reinforcing those outputs that meet the desired length criteria, and it penalizes shorter outputs that do not meet the objective.

3.2 Temporal Decomposition

While optimizing using Eq. (6) is intuitive, a potential issue arises: in some cases, the sampled outputs may significantly overlap, yet the rewards for the overlapping parts differ. For example, as illustrated in Figure 3, if outputs y_1 and y_2 both include the same tokens in the third tip "3. Double-check your work...anyone can significantly enhance their math problem-solving abilities.", but due to variations in the entire outputs, they receive markedly different rewards (e.g., $R(y_1) = 50$ vs. $R(y_2) = 10$). Ideally, we would want similar generation behaviors to be rewarded consistently, without significant variation, as their contributions are equivalent to the sum of rewards.

Before discussing an approach to solve this issue, we first perform temporal decomposition for the term $\frac{\partial \log \Pr_\theta(y|x)}{\partial \theta} R(y)$ in Eq. (6), and obtain

$$\begin{aligned}
 \frac{\partial \log \Pr_\theta(y|x)}{\partial \theta} R(y) &= \left(\sum_{t=1}^T \frac{\partial \log \Pr_\theta(y_t|x, y_{<t})}{\partial \theta} \right) \left(\sum_{k=1}^T r_k \right) \\
 &= \sum_{t=1}^T \frac{\partial \log \Pr_\theta(y_t|x, y_{<t})}{\partial \theta} \left(\sum_{k=1}^{t-1} r_k + \sum_{k=t}^T r_k \right)
 \end{aligned} \tag{7}$$

x	Give me three tips to improve my accuracy in solving math problems.
y1	Improving accuracy in solving math problems is crucial for success in mathematics. Here are three tips that can help: 1. Practice regularly... 2. Understand the concepts... 3. Double-check your work...any-one can significantly enhance their math problem-solving abilities. {Total Tokens: 160, Reward: 16}
y2	Improving math accuracy requires careful attention to detail, strategic problem-solving techniques, and consistent practice. Here are three tips: 1. Avoid rushing and read carefully... 2. Use estimation to validate answers... 3. Double-check your work...any-one can significantly enhance their math problem-solving abilities. {Total Tokens: 750, Reward: 75}

Figure 3: Example of sampled outputs demonstrating that the same tokens will receive significantly different rewards (i.e., 16 vs. 75). This introduces high gradient variance in the optimization process of the policy gradient.

In this form, we observe that, when generating the token at t -th time step, it does not affect the term $\sum_{k=1}^{t-1} r_k$ and only affects the term $\sum_{k=t}^T r_k$. Therefore, we can safely remove the former part, $\sum_{k=1}^{t-1} r_k$, to prevent it from affecting the optimization of the current step. By doing so, we can achieve a more stable gradient computation for Eq. (6):

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\frac{1}{d} \sum_{\mathbf{y} \in \mathcal{D}} \sum_{t=1}^T \frac{\partial \log \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t})}{\partial \theta} \sum_{k=t}^T r_k \right] \quad (8)$$

In fact, this simplification follows the Markov process, asserting that the future is independent of the past, given the present. As a result, we focus on the reward terms from the t -th time step onward, which are directly influenced by the token generation at time step t .

3.3 Reducing Gradient Variance

Returning to the case discussed in Section 3.2, while the strategy of excluding rewards for past tokens reduces gradient variance, substantial gradient variance still occurs in practice. First, if overlapping portions of the output appear early, the modified Eq. (9) may still experience similar problems, i.e., the same tokens receive vastly different rewards. Furthermore, the reward distribution can vary significantly between different time steps within a single output. For example, consider an output sequence of 500 tokens. According to Eq. (9), the reward at the 10th step might be significantly higher than at the 450th step, i.e., 49 vs. 5. Such varying rewards for good and poor tokens can result in a very low total reward for the entire output, even if it includes good tokens.

One simple method for further reducing the variance of the gradient is to set a baseline b and subtract it from $\sum_{k=t}^T r_k$, resulting in $\sum_{k=t}^T r_k - b$.³ Here, the baseline can be interpreted as a reference point. By centering the rewards around this baseline, we remove systematic biases in the reward.

³In fact, the use of a baseline b does not change the variance of the total rewards $\sum_{t=1}^T r_t$. However, it is important to note that while introducing a baseline does not alter the overall variance of the rewards, it helps reduce the variance of the gradient estimates. This is because subtracting the baseline from the total rewards effectively reduces fluctuations around their mean, which makes the gradient estimates more stable. In general, the operation $\sum_{k=t}^T r_k - b$ centers the rewards around zero (e.g., b is defined as the expected value of $\sum_{k=t}^T r_k$), which can lead to reduced variance in the product $\sum_{k=t}^T \log \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t}) (\sum_{k=t}^T r_k - b)$.

This policy gradient model with a baseline can be given by

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\frac{1}{d} \sum_{\mathbf{y} \in \mathcal{D}} \sum_{t=1}^T \frac{\partial \log \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t})}{\partial \theta} \left(\sum_{k=t}^T r_k - b \right) \right] \quad (9)$$

There are many ways to define the baseline b . For example, we can set the baseline b to the average length of the sampled outputs across $\mathcal{D}$, i.e., $b = (\sum_{\mathbf{y} \in \mathcal{D}} \text{Length}(\mathbf{y})) / d$. While this approach can reduce the relative size of the gradient variance, it does not address the issue of varying gradient variance across different time steps within a single output. To tackle this challenge, we would want a different baseline for each time step that can specifically be adjusted to reduce variance at that step. To achieve this ideal baseline, we can introduce a value function $V(\cdot)$. In the training of the LLM, this value function calculates the expected value of the sum of future rewards (or return for short) when generating y_t , given the input $\mathbf{x}$ and the tokens previously generated $\mathbf{y}_{<t}$, i.e., $V(\mathbf{x}, \mathbf{y}_{<t}, y_t) = \mathbb{E}[\sum_{k=t}^T r_k]$. Hence we have

$$\begin{aligned} A(\mathbf{x}, \mathbf{y}_{<t}, y_t) &= \sum_{k=t}^T r_k - b \\ &= \sum_{k=t}^T r_k - V(\mathbf{x}, \mathbf{y}_{<t}, y_t) \end{aligned} \quad (10)$$

where $\sum_{k=t}^T r_k$ represents the actual return received, and $V(\mathbf{x}, \mathbf{y}_{<t}, y_t)$ (or V_t for short) represents the expected return at time step t . $A(\mathbf{x}, \mathbf{y}_{<t}, y_t)$ (or A_t for short) is called the advantage at time step t , which quantifies the relative benefit of the current generated y_t compared to the expected return. Computing the advantage using Eq. (10) is a method known as Monte Carlo-based advantage estimation. By using the advantage function A_t , the gradient of $J(\theta)$ can be written in the form

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\frac{1}{d} \sum_{\mathbf{y} \in \mathcal{D}} \sum_{t=1}^T \frac{\partial (\log \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t}) A_t)}{\partial \theta} \right] \quad (11)$$

Based on this training objective, the loss function of training the LLM can be written in the form

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \mathbb{E}_{\mathbf{y} \sim \mathcal{D}} \left[\sum_{t=1}^T \log \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t}) A_t \right] \quad (12)$$

In practice, there are many ways to implement the value function. One simple approach is to build it based on a pre-trained LLM (e.g., an SFT LLM). In this way, the value function is also called the value model (or critic model). More specifically, we can concatenate $\mathbf{x}$ and $\mathbf{y}$ to form a single token sequence $\text{seq}_{\mathbf{x}, \mathbf{y}} = [\mathbf{x}, \mathbf{y}]$. We run an SFT LLM on this sequence, as usual, and at each position, we obtain a representation from the top-most Transformer layer. Then, we take the representations at the output (denoted by $\{\mathbf{h}_{y_1}, \mathbf{h}_{y_2}, \dots, \mathbf{h}_{y_T}\}$) and map them to scalars via linear transformation, respectively. Figure 4 illustrates this architecture of the value model. Hence, at t -th time step, we can compute the value V_t

$$V_t = \mathbf{h}_{y_t} \mathbf{W}_v \quad (13)$$

where $\mathbf{h}_{y_t}$ is a d -dimensional vector, and $\mathbf{W}_v$ is a $d \times 1$ linear mapping matrix. This value model is typically trained using the Temporal Difference (TD) error. This training method relies on the principle that the difference between the values at adjacent time steps should correspond to the reward received at the former time step, i.e., $V_t - V_{t+1} = r_t$. Suppose the value model is parameterized by ω . Given a sequence $\text{seq}_{\mathbf{x}, \mathbf{y}}$, the loss function is given by

$$\mathcal{L}_v(\omega) = \frac{1}{T} \sum_{t=1}^T (r_t + V_{\omega, t+1} - V_{\omega, t})^2 \quad (14)$$

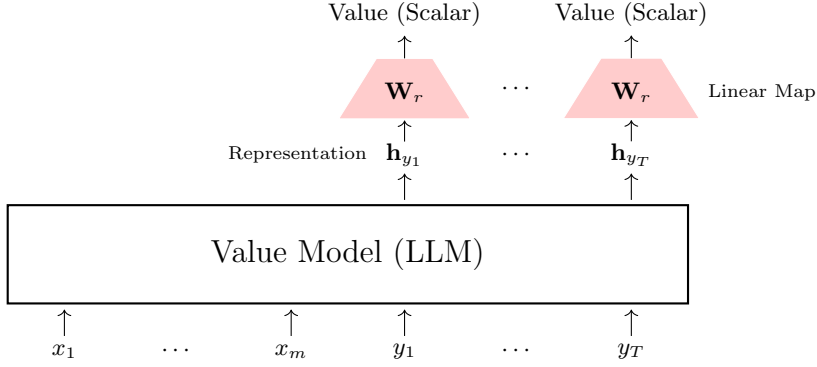


Figure 4: Architecture of the value model based on a pre-trained LLM. The main component of this model is still an LLM. For a given sequence $\text{seq}_{\mathbf{x}, \mathbf{y}} = [\mathbf{x}, \mathbf{y}]$, the model extracts representations corresponding to the positions of the tokens in $\mathbf{y}$. These representations are then individually mapped to scalars through a linear transformation. Each scalar represents the value associated with each token in $\mathbf{y}$. Note that the linear mapping matrix used for the transformations is typically shared across all tokens.

or alternatively, introduce the discount factor γ to obtain a more general form

$$\mathcal{L}_v(\omega) = \frac{1}{T} \sum_{t=1}^T (r_t + \gamma V_{\omega, t+1} - V_{\omega, t})^2 \quad (15)$$

where $\gamma \in [0, 1]$ is the discount factor that adjusts the importance of future rewards. When γ is set to less than 1, it signifies that early rewards are considered more important than future rewards. This basic idea is also applied in other fields. For example, in LLMs, it has been demonstrated that early token generation plays a crucial role, as it can influence the style and accuracy of the entire output (Wang & Zhou, 2024).

In this subsection, we have detailed the integration of a value model in training LLMs. In practice, this training approach, represented by Eq. (11), is known as the Advantage Actor-Critic (A2C) method (Mnih et al., 2016). The A2C method facilitates an interaction where a policy model (the actor) and a value model (the critic) learn in parallel and undergo synchronous updates. However, RL is a vast field, and many technical details cannot be covered here. The interested reader can refer to RL books for more details (Szepesvári, 2010; Sutton & Barto, 2018).

3.4 Importance Sampling

In this subsection, we discuss methods to improve the efficiency of the policy gradient, focusing on the sampling process. Although we discussed in Section 3.1 that Monte Carlo-based sampling can estimate the hypothesis space, it is awfully inefficient for one single update in scenarios like ours where the sampled output may consist of hundreds or thousands of tokens. A natural idea from this point is to consider whether it is possible to reuse these sampled outputs to update the LLMs multiple times. With importance sampling, this is feasible, and we can rewrite the loss function of training the LLM as

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \mathbb{E}_{\mathbf{y} \sim \text{Pr}_{\theta_{\text{ref}}}(\cdot | \mathbf{x})} \left[\sum_{t=1}^T \frac{\text{Pr}_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t})}{\text{Pr}_{\theta_{\text{ref}}}(y_t | \mathbf{x}, \mathbf{y}_{<t})} A_t \right] \quad (16)$$

where θ_{ref} denotes the parameters of the previously used LLM (also called reference policy or old policy). Here, we use $\text{Pr}_{\theta}(\cdot | \mathbf{x})$ or $\text{Pr}_{\theta_{\text{ref}}}(\cdot | \mathbf{x})$ to denote $\mathcal{D}$, distinguishing from which model the output is sampled. The ratio $\frac{\text{Pr}_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t})}{\text{Pr}_{\theta_{\text{ref}}}(y_t | \mathbf{x}, \mathbf{y}_{<t})}$, also called the ratio function, compares the probability of token y_t under the current and reference policies. This ratio function is used to reweight the observed rewards, reflecting how

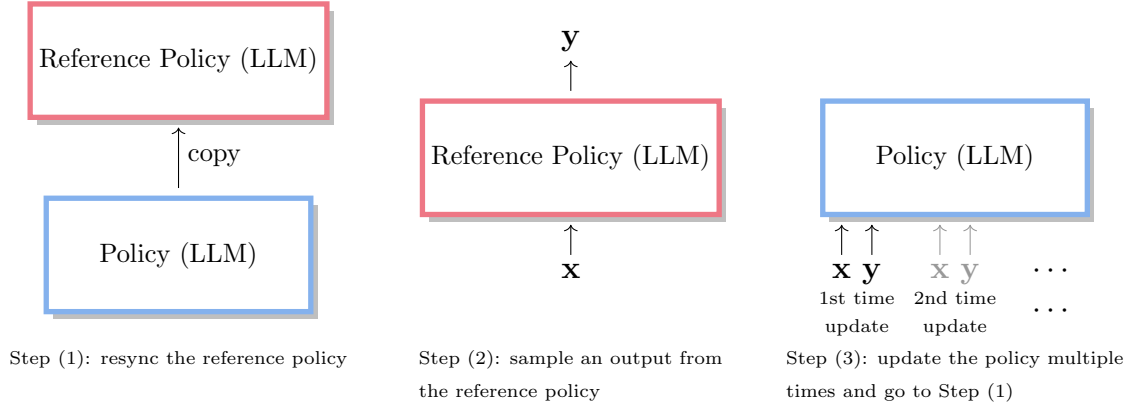


Figure 5: Workflow of integrating importance sampling in the training of LLMs with policy gradient. In Step 1, the current policy is synchronized to establish the reference policy. In Step 2, we sample an output $\mathbf{y}$ from this reference policy for a given input $\mathbf{x}$. Finally, in Step 3, the sequence $[\mathbf{x}, \mathbf{y}]$ is used to optimize the current policy multiple times. After optimization, the updated policy is synchronized to become the reference policy.

much more or less likely a token is under the current policy compared to the reference policy. When this ratio is greater than 1, it indicates that the token y_t is more favored by the current policy than the reference policy. Conversely, a ratio less than 1 indicates that y_t is less favored by the current policy. However, when the current θ_{ref} diverges significantly from θ , the accuracy of the hypothesis space estimation decreases. Therefore, we do need to resync it regularly. As illustrated in Figure 5, one simple way is to designate the reference policy as the LLM from which we initiate updates during a training step. Note that we can conserve computational time and memory by eliminating the need for model copying in Step 1. Specifically, rather than maintaining a separate copy of the model, we directly sample from the current policy, retain the output probabilities $\{\Pr_{\theta}(y_1|\mathbf{x}), \dots, \Pr_{\theta}(y_T|\mathbf{x}, \mathbf{y}_{<T})\}$, and later use these stored probabilities to act as $\{\Pr_{\theta_{\text{ref}}}(y_1|\mathbf{x}), \dots, \Pr_{\theta_{\text{ref}}}(y_T|\mathbf{x}, \mathbf{y}_{<T})\}$ when updating the policy with Eq. (16).

However, an inherent issue arises when using importance sampling to update LLMs. As shown in Figure 6, when a particular optimization step results in poor updates (or a terrible step for short), the reference policy utilized in subsequent Step 1 of the next iteration inherits these poor characteristics. Consequently, the samples drawn in Step 2 are likely to be adversely affected, leading to a more terrible step. Unlike SFT⁴, this cycle does not correct the initial terrible step but potentially exacerbates it, creating a downward spiral in policy performance.

Addressing this issue involves considering trust regions in optimization (Schulman et al., 2015), which refers to a region around the current parameter estimate where the model is well-behaved. One approach to incorporating trust regions is to impose a constraint on the size of the policy update. This can be effectively managed by imposing a constraint that prevents significant deviations from the policy before optimization. In training LLMs, this constraint is typically achieved by adding a penalty to the objective function based on a divergence measure between current and reference policies. A simple form of such a penalty is given by the difference in the log-probability of the sampled $\mathbf{y}$ under the current policy versus the reference policy:

$$\text{Penalty} = \log \Pr_{\theta}(\mathbf{y}|\mathbf{x}) - \log \Pr_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \quad (17)$$

At the time step t , we can obtain the penalty as

$$\text{Penalty}_t = \log \Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t}) - \log \Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t}) \quad (18)$$

⁴In supervised learning, a terrible step during a training step caused by bad samples can often be corrected in subsequent steps by good samples.

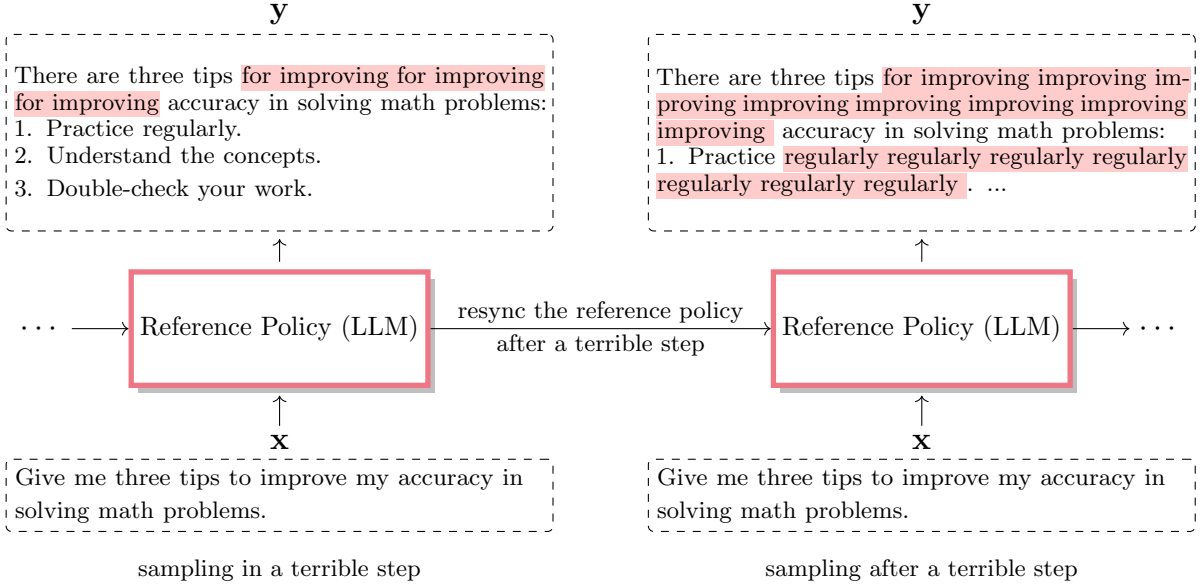


Figure 6: Illustration of the impact of a terrible step on sampling within an LLM training cycle using importance sampling. The left shows poor output sampled with slight repetition resulting in a terrible step. The right shows subsequent output sampled with more severe repetition after a terrible step.

By including this penalty in the optimization objective, we encourage the current policy to remain close to the reference policy, limiting very large updates in a terrible step.

We can incorporate this penalty into the Eq. (16), and obtain

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \mathbb{E}_{\mathbf{y} \sim \Pr_{\theta_{\text{ref}}}(\cdot|\mathbf{x})} \left[\sum_{t=1}^T \left(\frac{\Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t})}{\Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t})} A_t - \beta \text{Penalty}_t \right) \right] \quad (19)$$

where β is the weight of the penalty. This training method is also called trust region policy optimization (TRPO) (Schulman et al., 2015).

3.5 Proximal Policy Optimization

A further improvement to TRPO involves addressing the gradient variance problem outlined in Section 3.3. In Eq. (19), the range of the ratio function is from $[0, +\infty)$, which could lead to high gradient variance in the learning process. To mitigate this problem, clipping is commonly employed to limit the magnitude of importance weights, thereby preventing excessively large updates and promoting stability in the learning process. A clipped version can be given by

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \mathbb{E}_{\mathbf{y} \sim \Pr_{\theta_{\text{ref}}}(\cdot|\mathbf{x})} \left[\sum_{t=1}^T \left(\text{Clip}\left(\frac{\Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t})}{\Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t})} A_t\right) - \beta \text{Penalty}_t \right) \right] \quad (20)$$

where the clipping function $\text{Clip}(\cdot)$ can be defined by

$$\text{Clip}\left(\frac{\Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t})}{\Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t})} A_t\right) = \min\left(\frac{\Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t})}{\Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t})} A_t, \text{bound}\left(\frac{\Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t})}{\Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t})}, 1 - \epsilon, 1 + \epsilon\right) A_t\right) \quad (21)$$

where the function $\text{bound}(\cdot)$ constrains the ratio function to within the range $[1 - \epsilon, 1 + \epsilon]$. The clipping imposed by Eq. (21) is illustrated in Figure 7. This training method is also known as proximal policy

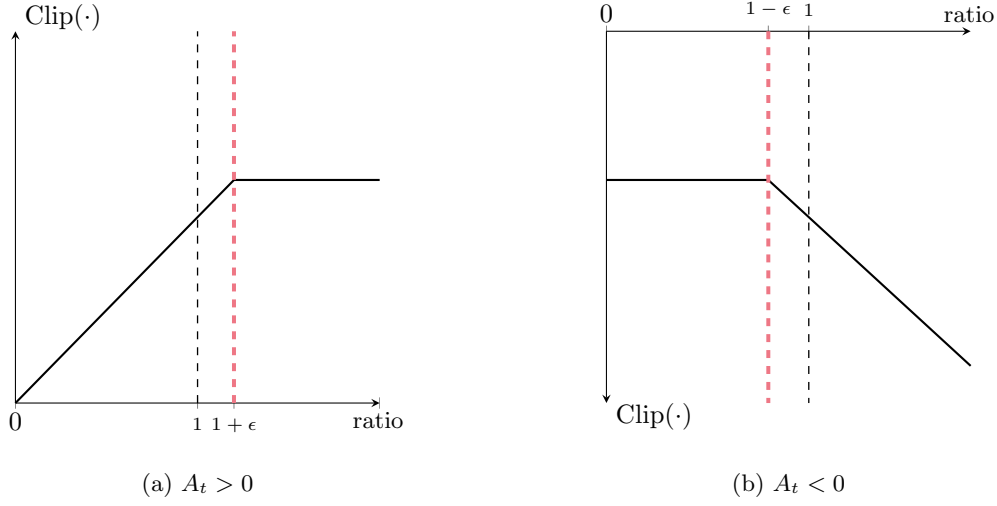


Figure 7: PPO clipping at time step t . The basic idea can be represented with a hypothesis space as described in Figure 2. The sub-figure (a) shows the case where the advantage A_t is positive, suggesting that the output is preferred. Here, the probability of the sampled output is increased within a defined constraint (up to $1 + \epsilon$) in the hypothesis space. The sub-figure (b) shows the case where A_t is negative, indicating a dispreferred output. In this case, the probability of the sample output is reduced to a minimum (down to $1 - \epsilon$) in the hypothesis space.

optimization (PPO) (Schulman et al., 2017), currently the most widely used method for training LLMs with RL. Originally, PPO also introduced an adaptive β to dynamically control this penalty. However, this adaptive approach has not been widely applied in training LLMs. This is because this process depends on the expectation of the penalty (i.e., $\mathbb{E}(\text{Penalty})$), which would introduce additional computational overhead. Interested readers can refer to the original literature for more details on this adaptive approach.

3.6 Training Reward Models

While our initial reward function based on output length, as described in Section 3.1, provides effective signals in the learning process, human preferences are considerably more complex and extend beyond merely preferring longer outputs. For example, in scenarios like a homework assistant, it is essential not only to generate longer outputs but also to avoid redundancy, ensure accuracy, and maintain fluency. This complexity underscores the need for a sophisticated reward modeling technique, moving beyond simple function-based methods to more effectively capture human preferences.

Given these complexities, we typically train a reward model, a neural network that maps a pair of input and output token sequences to a scalar value, to capture human preferences. Given an input $\mathbf{x}$ and an output $\mathbf{y}$, the reward is expressed as $\text{Reward}(\mathbf{x}, \mathbf{y})$, where $\text{Reward}(\cdot)$ denotes the reward model. There are many ways to implement the reward model. One simple approach is to build the reward model based on a pre-trained LLM, similar to the value model as presented in Section 3.3. More specifically, we employ the sequence $\text{seq}_{\mathbf{x}, \mathbf{y}} = [\mathbf{x}, \mathbf{y}]$ to serve as the input. We run the LLM on this sequence and obtain a representation from the top-most Transformer layer. Then, we take the representation at the last position of the output and map it to a scalar via linear transformation⁵:

$$R_\phi(\mathbf{x}, \mathbf{y}) = \mathbf{h}_{y_T} \mathbf{W}_r \quad (22)$$

⁵In practice, when employing an LLM for training a reward model, we utilize it primarily as an encoder to encode the sequence $\text{seq}_{\mathbf{x}, \mathbf{y}}$ into a representation. Here, the representation from the last position is selected as it encapsulates the semantic information of the entire sequence.

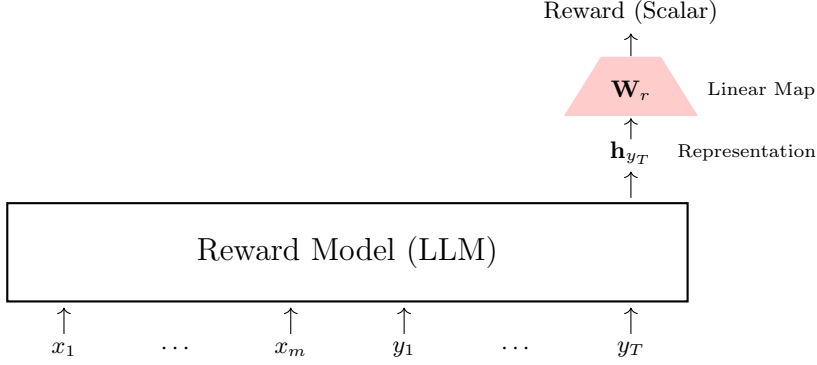


Figure 8: Architecture of the reward model based on an LLM. Unlike the value model depicted in Figure 4, this model extracts the representation from the last position of the output $\mathbf{y}$ to represent the entire sequence $[\mathbf{x}, \mathbf{y}]$. This representation is then mapped to a scalar through a linear transformation, which serves as the reward for the output.

where $\mathbf{W}_r$ is a $d \times 1$ linear mapping matrix, and ϕ represents the parameters of the reward model, which includes both the parameters of the LLM and the $\mathbf{W}_r$. This architecture of the reward model is illustrated in Figure 8.

To train the reward model, the first step is to collect human feedback on a set of generated outputs. Given an input $\mathbf{x}$, we use the LLM to produce multiple candidate outputs $\{\mathbf{y}_1, \dots, \mathbf{y}_d\}$. Then, we can obtain human feedback in the following ways:

- **Rating.** Human experts provide a score or rating for each output. This score is often a continuous or discrete numerical value, such as a score on a scale (e.g., 1-5 stars or 1-10 points). In some cases, the rating might be binary, indicating a “yes/no” or “positive/negative” preference.
- **Listwise Ranking.** Human experts are asked to rank or order the given set of possible outputs.
- **Pairwise Comparison (Pairwise Ranking).** Given two different outputs, human experts select which one is better. This annotation approach is particularly popular because it is easier to annotate than the other two methods.

Here we consider pairwise comparison feedback to train a reward model. In this setting, each time, two outputs $(\mathbf{y}_a, \mathbf{y}_b)$ are randomly drawn from the candidate output pool $\{\mathbf{y}_1, \dots, \mathbf{y}_d\}$. Human experts are then presented with these pairs and asked to decide which output they prefer based on specific criteria, such as information, accuracy, and fluency. The human feedback can be encoded as a binary label, $\mathbf{y}_a \succ \mathbf{y}_b$ for a preference for $\mathbf{y}_a$, and $\mathbf{y}_b \succ \mathbf{y}_a$ for a preference for $\mathbf{y}_b$.

One simple and widely used model for describing such pairwise comparisons is the Bradley-Terry model (Bradley & Terry, 1952). It is a probabilistic model that estimates the probability that one item is preferred over another. Adapting this model to the notation used here, we can write the probability that $\mathbf{y}_a$ is preferred over $\mathbf{y}_b$ in the form

$$\begin{aligned}
 \Pr_{\phi}(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x}) &= \frac{e^{R_{\phi}(\mathbf{x}, \mathbf{y}_a)}}{e^{R_{\phi}(\mathbf{x}, \mathbf{y}_a)} + e^{R_{\phi}(\mathbf{x}, \mathbf{y}_b)}} \\
 &= \frac{e^{R_{\phi}(\mathbf{x}, \mathbf{y}_a) - R_{\phi}(\mathbf{x}, \mathbf{y}_b)}}{e^{R_{\phi}(\mathbf{x}, \mathbf{y}_a) - R_{\phi}(\mathbf{x}, \mathbf{y}_b)} + 1} \\
 &= \text{Sigmoid}(R_{\phi}(\mathbf{x}, \mathbf{y}_a) - R_{\phi}(\mathbf{x}, \mathbf{y}_b))
 \end{aligned} \tag{23}$$

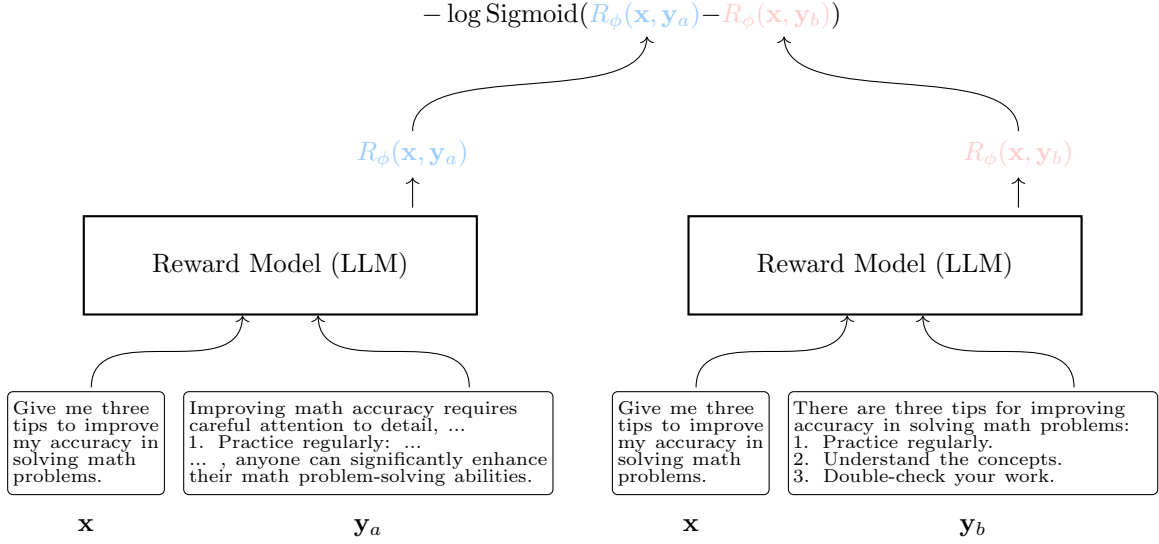


Figure 9: An example of loss computation using the Bradley-Terry model. Given an input $\mathbf{x}$ and two outputs $(\mathbf{y}_a, \mathbf{y}_b)$, where $\mathbf{y}_a \succ \mathbf{y}_b$, we initially obtain rewards $R(\mathbf{x}, \mathbf{y}_a)$ and $R(\mathbf{x}, \mathbf{y}_b)$ for two sequences $\text{seq}_{\mathbf{x}, \mathbf{y}_a}$ and $\text{seq}_{\mathbf{x}, \mathbf{y}_b}$. These rewards are subsequently used to compute the loss as described in Eq. (24). Note that, for efficiency, both sequences are typically processed in a single batch during one forward pass to simultaneously obtain $R(\mathbf{x}, \mathbf{y}_a)$ and $R(\mathbf{x}, \mathbf{y}_b)$. In fact, this training approach is also common in Siamese networks (Bromley et al., 1993).

When training the reward model, we want to maximize this preference probability. A loss function based on the Bradley-Terry model is given by

$$\begin{aligned} \mathcal{L}_r(\phi) &= -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [\log \Pr_\phi(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x})] \\ &= -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [\log \text{Sigmoid}(R_\phi(\mathbf{x}, \mathbf{y}_a) - R_\phi(\mathbf{x}, \mathbf{y}_b))] \end{aligned} \quad (24)$$

where $(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b)$ is drawn from a human-annotated dataset $\mathcal{D}_r$ consisting of preference pairs of outputs and their corresponding inputs. The goal of training the reward model is to find the optimal parameters $\hat{\phi}$ that minimize this loss function, given by

$$\hat{\phi} = \arg \min_{\phi} \mathcal{L}_r(\phi) \quad (25)$$

Since the reward model itself is also an LLM, we can directly reuse the Transformer training procedure to optimize the reward model. The difference from training a standard LLM is that we only need to replace the cross-entropy loss with the pairwise comparison loss as illustrated in Figure 9. After the training of the reward model, we can apply the trained reward model $R_\phi(\cdot)$ to supervise the target LLM for alignment.

It is worth noting that although we train the reward model to perform pairwise ranking, we apply it to score each input-output pair independently during the alignment process. The pairwise ranking objective ensures that the reward model is sensitive to subtle differences between outputs, but we rely on the continuous scores produced by the reward model to guide the optimization of the LLM. An advantage of this approach is that we can choose from or combine various ranking loss functions and still apply the resulting reward models in the same way as we have done in Section ?? . However, a challenge arises with this method: the reward model can provide only sparse rewards; that is, it offers a delayed reward rather than an intermediate one. Hence, in this case, we can obtain

$$r_t = \begin{cases} 0 & \text{if } t < T \\ R_\phi(\mathbf{x}, \mathbf{y}) & \text{if } t = T \end{cases} \quad (26)$$

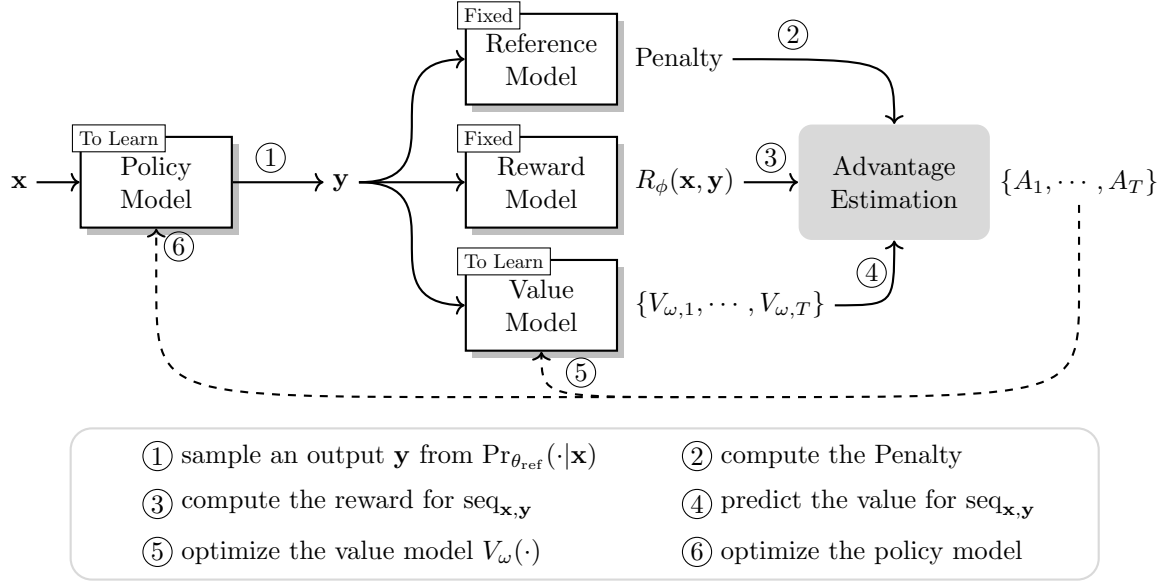


Figure 10: Illustration of PPO-based optimization workflow. Given an optimized reward model and a reference model, we proceed to train both the policy and the value model. At each prediction step, we compute the sum of the PPO-based loss and update the policy parameters. This requires access to the reward model, the reference model, and the value model at hand. At the same time, we update the parameters of the value model. It is worth noting that steps 2, 3, and 4 do not follow a fixed order, and they can be executed simultaneously or in any order.

To address this challenge, we can typically incorporate shaping rewards into the learning process (Wu et al., 2023) or train a process reward model by annotating process preference data (Lightman et al., 2024). Please refer to Section ?? for more details.

3.7 A Complete Workflow

Up to this point, we have used numerous examples to discuss how to use RL to train LLMs to better align with human preferences. We have also shown how different RL algorithms are designed to address specific challenges in our scenarios. In this subsection, we will outline a complete workflow for training LLMs using RL, specifically employing the PPO. To implement PPO, we first build four models, all based on an LLM.

- **Reward Model** (denoted by $R_{\phi}(\cdot)$ where ϕ denotes the parameters). The reward model learns from human preference data to predict the reward for each pair of input and output token sequences. It is typically initialized with an SFT LLM or a pre-trained LLM.
- **Value Model** or **Value Function** (denoted by $V_{\omega}(\cdot)$ where ω denotes the parameters). The value model receives rewards from the reward model and is trained to predict the expected sum of rewards. It is generally initialized with a reward model or an SFT LLM.
- **Reference Model** or **Old Policy Model** (denoted by $\text{Pr}_{\theta_{\text{ref}}}(\cdot)$ where θ_{ref} denotes the parameters). The reference model is the baseline LLM that serves as a starting point for policy training. Unlike the discussion in Section 3.4, when computing the penalty in RLHF, we typically use the previous version of the model or a model trained without human feedback to serve as the reference policy model, making a more stable learning process.

- **Policy Model** (denoted by $\text{Pr}_\theta(\cdot)$ where θ denotes the parameters). Given its context, this policy governs how the LLM decides the most appropriate next token. It is trained under the supervision of both the reward model and the value model.

In practice, these models need to be trained in a certain order. First, we need to initialize them using some other models. For example, the reward model and the value model can be initialized with a pre-trained LLM or an SFT LLM, while the reference model and the policy model can be initialized with an SFT LLM. Note that, at this point, the reference model is ready for use and will not be further updated. Second, we need to collect human preference data and train the reward model on this data. Third, both the value model and the policy are trained simultaneously using the reward model⁶. At each position in an output sequence, we update the value model by minimizing the MSE error of value prediction, and the policy is updated by minimizing the PPO loss.

Although the RL process introduced above seems highly promising, as it can effectively align the SFT LLM with human preferences, there are significant challenges in implementing this process. For example,

- Training a reward model requires labeled preference data. Consequently, it is essential to generate multiple outputs for a given input and label them according to human preferences. For example, in our scenario, we need to annotate extensive data to accurately reflect student preferences, thereby enabling the homework assistant to produce the content students expect. Furthermore, in this process, it is crucial to ensure that the data is of high quality and accurately mirrors human preferences. This is because inaccuracies in data can lead to misguided learning, where the model might adopt undesirable biases or fail to meet user expectations, also called overoptimization or reward hacking (Gao et al., 2023). Therefore, such preference data must be meticulously collected and sufficiently extensive to enable the reward model to accurately capture human preferences.
- RL is computationally expensive for training LLMs. This arises from two primary factors. One is that during the RL process, we need to perform sampling in an autoregressive mode (Xiao & Zhu, 2023). This is highly computationally intensive because the policy model usually has a large number of parameters. Furthermore, during the PPO-based optimization, we need to load four LLMs simultaneously, which requires significantly more GPU memory compared to the SFT. To address these challenges, there has been significant interest in developing efficient RL strategies, such as dynamic sampling (Wang et al., 2024b) and rule-based rewards (Shao et al., 2024), which can maintain state-of-the-art performance without requiring high computational and time costs.
- RL is often an unstable process. While RL provides a strong theoretical foundation for the design of each component, it is also highly sensitive to changes in any part of the system. Even small adjustments to the reward model, policy model, or hyperparameters can lead to significant fluctuations in performance, making it difficult to ensure consistent and stable results. This fragility highlights the importance of carefully tuning and stabilizing the various elements involved in RL, especially when applied to complex systems like LLMs. Techniques such as reward shaping, adaptive learning rate, and more sophisticated optimization strategies are often required to mitigate instability and improve the robustness of the learning process.

⁶During PPO-based optimization, a cold-start approach can be used where only the value model is updated in the initial phases, avoiding adjustments to the policy model based on potential inaccuracies in early value predictions (Wang et al., 2024a).

References

- Ralph Allan Bradley and Milton E. Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.
- Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a” siamese” time delay neural network. *Advances in neural information processing systems*, 6, 1993.
- Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10835–10866. PMLR, 2023. URL <https://proceedings.mlr.press/v202/gao23h.html>.
- Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s verify step by step. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL <https://openreview.net/forum?id=v8LOpN6EOi>.
- Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria-Florina Balcan and Kilian Q. Weinberger (eds.), *Proceedings of the 33rd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pp. 1928–1937. JMLR.org, 2016. URL <http://proceedings.mlr.press/v48/mniha16.html>.
- John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In Francis R. Bach and David M. Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pp. 1889–1897. JMLR.org, 2015. URL <http://proceedings.mlr.press/v37/schulman15.html>.
- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL <https://arxiv.org/abs/1707.06347>.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL <https://arxiv.org/abs/2402.03300>.
- Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (2nd ed.)*. The MIT Press, 2018.
- Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, 2010.
- Chenglong Wang, Hang Zhou, Kaiyan Chang, Bei Li, Yongyu Mu, Tong Xiao, Tongran Liu, and Jingbo Zhu. Hybrid alignment training for large language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 11389–11403, Bangkok, Thailand, 2024a. Association for Computational Linguistics. URL <https://aclanthology.org/2024.findings-acl.676/>.
- Chenglong Wang, Hang Zhou, Yimin Hu, Yifu Huo, Bei Li, Tongran Liu, Tong Xiao, and Jingbo Zhu. ESRL: efficient sampling-based reinforcement learning for sequence generation. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (eds.), *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2024, February 20-27, 2024, Vancouver, Canada*, pp. 19107–19115. AAAI Press, 2024b. URL <https://doi.org/10.1609/aaai.v38i17.29878>.

Xuezhi Wang and Denny Zhou. Chain-of-thought reasoning without prompting. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/7a8e7fd295aa04eac4b470ae27f8785c-Abstract-Conference.html.

Zequi Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A. Smith, Mari Ostendorf, and Hannaneh Hajishirzi. Fine-grained human feedback gives better rewards for language model training. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/b8c90b65739ae8417e61eadb521f63d5-Abstract-Conference.html.

Tong Xiao and Jingbo Zhu. Introduction to transformers: an nlp perspective, 2023. URL <https://arxiv.org/abs/2311.17633>.