

Reinforcement Learning without Tears: An Introduction for Large Language Model Researchers

Chenglong Wang, Hang Zhou, Tong Xiao & Jingbo Zhu

School of Computer Science and Engineering

Northeastern University

Shenyang, China

{clwang1119,stceum}@gmail.com

{xiaotong,zhujingbo}@mail.neu.edu.cn

Tongran Liu

CAS Key Laboratory of Behavioral Science

Institute of Psychology

Chinese Academy of Sciences

Beijing, China

liutr@psych.ac.cn

Abstract

abstract.

Contents

1 Introduction

Reinforcement learning (RL) is an interdisciplinary field, and has its origins in both psychology and optimal control. Over the years, the concept has been further developed and formalized within the realms of artificial intelligence and machine learning. The basic idea is that an agent learns to make decisions by receiving feedback on its actions from the environment. This approach is very general and applies to a wide range of problems, from playing complex games like chess and Go to controlling autonomous vehicles.

A recent breakthrough is the large-scale application of RL in the field of natural language processing (NLP), in particular in the development of large language models (LLMs). For example, RL has been widely considered an effective way of aligning pre-trained LLMs with human preferences and values (?). One such method, called reinforcement learning from human feedback (RLHF), forms the basis for the development of advanced LLMs like OpenAI’s GPT-4. More recently, RL has shown great promise in enabling LLMs to perform complex reasoning (??). As a result, interest in NLP has exploded. There have been many, many conference papers discussing RL in LLMs, and even more in the past few months. The research community is highly enthusiastic, and many in the field are anticipating a new turning point where large-scale RL algorithms could further advance AI.

However, RL was not so popular in the long history of NLP, and the field has just begun to explore its potential. Although applying RL to LLMs seems a natural choice for machine learning researchers, it introduces many new concepts to the NLP community, which has traditionally been dominated by supervised learning methodologies. As NLP researchers, when we attended presentations on LLMs, we often heard other researchers discuss advanced RL techniques, such as “use a value function to estimate the expected cumulative reward” or “learn a policy via PPO”. It seems that these techniques are so simple that presenters do not even need to give any explanation. But we were lost when seeing the math of RL, such as all those expectation signs, as well as the names of various algorithms that we are unfamiliar with.

Of course, we’d like to learn about RL, which appears straightforward. However, common RL textbooks, such as ?’s book, are mostly based on classic robotics or control problems. This makes it difficult to align the concepts of RL with those of LLMs. On the other hand, most LLM literature lacks an in-depth discussion of RL techniques, often omitting related details. As a result, we find ourselves in an awkward middle ground between RL and LLMs, struggling to bridge the two fields.

The aim of this paper is to provide a comprehensive introduction to RL from the perspective of LLMs. We begin by introducing basic concepts and algorithms of RL using the LLM language. In particular, we illustrate their application through an example of training LLMs, making the concepts more accessible. We then discuss a series of refinements to the basic RL framework for LLM alignment, including advanced reward modeling, improved advantage estimation, efficient sampling, and direct LLM optimization without RL. Furthermore, we discuss how to apply RL techniques to LLM reasoning. These methods are closely related to recent LLMs, such as OpenAI o1/o3 and DeepSeek R1, which adopt large-scale RL and test-time scaling to significantly advance their reasoning abilities. In addition, we discuss applications of RL to multimodal LLMs to demonstrate how RL can be adapted to various problems. Finally, we conclude by outlining promising future research directions and discussing relevant systems and datasets.

2 Preliminary

2.1 Fundamentals of Large Language Models

In this subsection, we introduce the fundamentals of supervised fine-tuning for LLMs and focus on presenting the key concepts and notions necessary for the discussions in future sections.

Although pre-trained LLMs possess a vast amount of general knowledge, they are typically limited to tasks related to language modeling. Our ultimate goal is to enable them to perform various specific tasks,

such as summarization, question-answer, and machine translation. One straightforward method to achieve this goal is supervised fine-tuning (SFT), in which the pre-trained LLM is further trained on a dataset comprising task-specific input (i.e., instruction+user input)¹ paired with their expected outputs (??).

Specifically, let $\mathbf{x} = x_0 \dots x_m$ be an input (e.g., instruction + user input) and $\mathbf{y} = y_1 \dots y_n$ be the corresponding output. In SFT, we aim to maximize the probability of the output $\mathbf{y}$ given the input $\mathbf{x}$. Consider an LLM with pre-trained parameters $\hat{\theta}$. The fine-tuning objective can then be formulated as:

$$\tilde{\theta} = \arg \max_{\hat{\theta}^+} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{S}} \log \Pr_{\hat{\theta}^+}(\mathbf{y}|\mathbf{x}) \quad (1)$$

where $\Pr(\cdot)$ denotes the probability distribution, $\mathcal{S}$ denotes the labeled data, $\tilde{\theta}$ denotes the parameters optimized via SFT, and $\hat{\theta}^+$ represents an adjustment to $\hat{\theta}$. Here, we will omit the superscript + and use θ to represent $\hat{\theta}^+$ to keep the notation uncluttered. However, the reader should remember that the fine-tuning starts from the pre-trained parameters rather than randomly initialized ones.

The objective function $\log \Pr_{\theta}(y_i|\mathbf{x}, \mathbf{y}_{<i})$ is computed by summing the log-probabilities of the tokens in $\mathbf{y}$, conditional on the input $\mathbf{x}$ and all the previous tokens $\mathbf{y}_{<i}$:

$$\log \Pr_{\theta}(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^n \log \Pr_{\theta}(y_i|\mathbf{x}, \mathbf{y}_{<i}) \quad (2)$$

This formulation is equivalent to minimizing the cross-entropy loss. In the following section, we refer to the LLM after SFT as the “SFT LLM” for short. Beyond SFT, the model architecture design and the pre-training process are also fundamentals of building an LLM. However, we will not discuss these topics. Interested readers can find further details in ?’s book.

2.2 Fundamentals of Reinforcement Learning

RL, supported by a well-established theoretical framework, has been widely applied across a broad range of domains. In particular, with the rapid advancement of LLMs, RL has become a standard approach in the post-training stage. Before exploring the interplay between RL and LLMs, we first provide a brief overview of deep reinforcement learning in this subsection. We then introduce the general formulation of RL, along with key terminologies and notations that are essential for understanding RL.

2.2.1 Markov Decision Process

The general framework of RL is illustrated in Figure ?? . An RL system primarily consists of two components: an agent and an environment. At each timestep t , the agent observes a state s_t from the environment and selects an action a_t according to a policy π , which is typically parameterized by a neural network. After executing the action, the environment transitions to a new state s_{t+1} and returns a reward r_t corresponding to the taken action.

This interaction process is commonly modeled as a Markov Decision Process (MDP), where the agent interacts with the environment over discrete timesteps (?). A sequence of states and actions forms a trajectory, denoted as $\tau = (s_0, a_0, s_1, a_1, \dots, s_{H-1}, a_{H-1})$, where H represents the trajectory length. Each trajectory accumulates rewards from the environment.

¹In this paper, the *instruction* represents the description of a specific task, e.g., “Summarize the following article”; the *user input* represents the extra content to the instruction, e.g., “Article: In recent years, solar energy has seen a significant increase in the amount of energy available to the public. recent years, solar energy has seen unprecedented growth, becoming the fastest-growing ...”

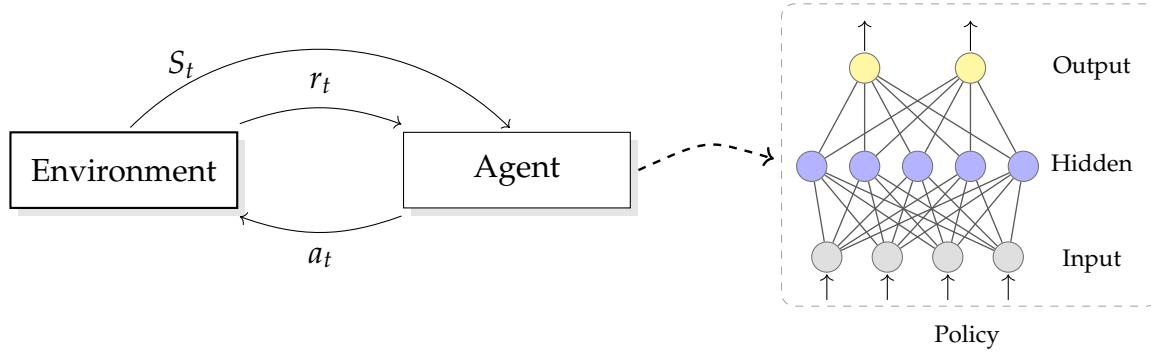


Figure 1: A general framework of deep reinforcement learning. The policy is parameterized by neural networks, and the agent learns an optimal policy through interactions with the environment to maximize cumulative rewards.

The objective of RL is to learn an optimal policy π^* that maximizes the expected cumulative reward over trajectories, which can be formulated as:

$$\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t=0}^{H-1} r_t \mid \pi \right] \quad (3)$$

In practice, the agent improves its policy through repeated interactions with the environment by observing states and receiving feedback in the form of rewards. Each such interaction sequence constitutes a trajectory τ . The process of generating one or more trajectories under a given policy is commonly referred to as *sampling* in the RL literature.

2.2.2 Key Elements

To better understand the application of reinforcement learning to large language models (LLMs), we summarize the key elements of the RL framework and reinterpret them in the context of language modeling.

- **Agent.** The agent is the learner or decision-maker in reinforcement learning. In the context of LLMs, the agent corresponds to the language model itself, which generates tokens sequentially and updates its behavior based on feedback signals.
- **Environment.** The environment comprises everything external to the agent with which it interacts. Unlike traditional RL settings that involve physical or simulated environments, the environment in LLM-based RL is typically abstract, consisting of the training framework that provides feedback (e.g., reward models, human annotations, or evaluation metrics) for generated outputs.
- **State (S).** A state represents the current situation of the environment. For language modeling, the state at timestep t can be defined as the sequence of observed tokens up to that point, i.e., the context used to predict the next token. Formally, the state can be represented as $S = (x, y_{<t})$, where x denotes the input prompt and $y_{<t}$ denotes the previously generated tokens.
- **Action (a).** An action corresponds to a decision made by the agent. In LLMs, actions are naturally defined as selecting the next token from the vocabulary, i.e., $a = y_t$.
- **Reward (r).** The reward provides feedback from the environment to evaluate the quality of an action. In general, the reward function can be defined as $r(s, a, s')$, representing the feedback received when the agent transitions from state s to s' by taking action a . At timestep t , this can be written as $r_t = r(s_t, a_t, s_{t+1})$. In deterministic settings, where the next state is uniquely determined by (s_t, a_t) , the reward can be simplified as $r(s_t, a_t)$.

- **Policy (π).** The policy defines the agent’s behavior, i.e., the probability of taking an action given a state. For LLMs, the policy corresponds to the conditional probability distribution over the next token given the context:

$$\pi(a | s) = \Pr(y_t | x, y_{<t}) \quad (4)$$

where $a = y_t$ and $s = (x, y_{<t})$. Under this formulation, an LLM can be naturally interpreted as a parameterized policy.

- **Value Function (V and Q).** The value function estimates the expected cumulative reward when following a policy. The state-value function $V(s)$ measures the expected discounted return starting from state s :

$$V(s) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t \middle| s_0 = s, \pi \right] \quad (5)$$

where $\gamma \in [0, 1]$ is the discount factor. The action-value function $Q(s, a)$ further conditions on the initial action:

$$Q(s, a) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t \middle| s_0 = s, a_0 = a, \pi \right] \quad (6)$$

By this point, the reader should have developed a foundational understanding of the core concepts and notations in RL. In contrast to conventional RL literature, which typically presents algorithms in the context of classical robotics or control tasks, *we adopt an NLP perspective to introduce RL in the following section.*

3 An Example of Using Reinforcement Learning to Train LLMs

We begin by considering a practical scenario: using an LLM as a homework assistant. In this context, we aim 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 typical SFT LLM might generate 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 will demonstrate how to use RL to train LLMs through a specific example—enabling the LLM to generate a longer output in the context of a homework assistant. Throughout this example, we introduce RL, including key 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 behaviour, 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:

$$R(\mathbf{y}) = \frac{\text{Length}(\mathbf{y})}{10} \quad (7)$$

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., $\Pr_\theta(\cdot|\mathbf{x})$ ².

The optimization objective can be given by

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \Omega} \Pr_\theta(\mathbf{y}|\mathbf{x}) R(\mathbf{y}) \right] \\ &= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \Omega} \Pr_\theta(\mathbf{y}|\mathbf{x}) \sum_{t=1}^T r_t \right] \end{aligned} \quad (8)$$

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 , T indicates the length of $\mathbf{y}$. $J(\theta)$ is also called the performance function. Then the training objective is maximize $J(\theta)$:

$$\hat{\theta} = \arg \max_{\theta} J(\theta) \quad (9)$$

Note that the hypothesis space Ω is typically very large, as LLMs often have large vocabularies³. 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 $\mathcal{D}$:

$$J(\theta) = \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \mathcal{D}} \Pr_\theta(\mathbf{y}|\mathbf{x}) \sum_{t=1}^T r_t \right] \quad (10)$$

²To 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.

³The 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.

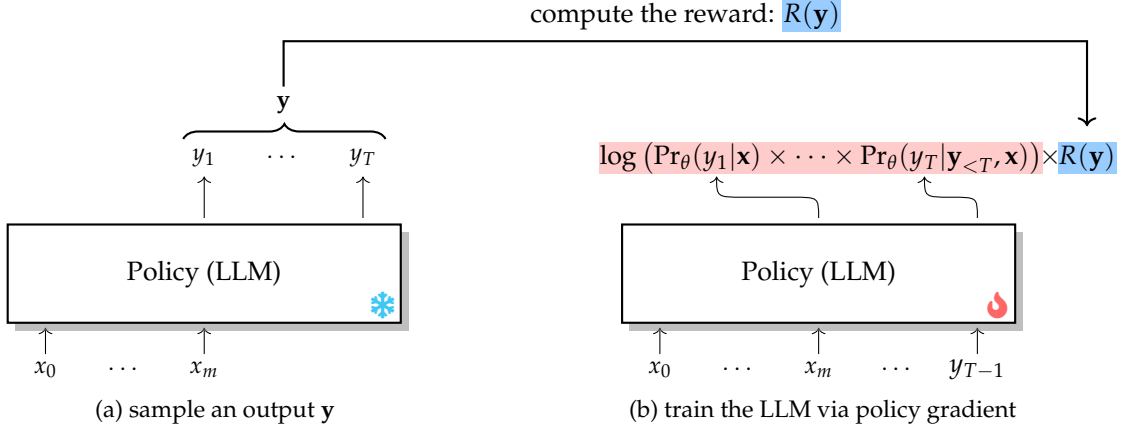


Figure 2: Illustration of training an LLM using policy gradient. The central component is an LLM. Given an input $\mathbf{x} = x_1 \dots x_m$, we sample an output $\mathbf{y}$ from the LLM. Note that gradients are not computed during the sampling to prevent extensive memory consumption because the sampling 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 :

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta} &= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \mathcal{D}} \frac{\partial \Pr_{\theta}(\mathbf{y}|\mathbf{x}) R(\mathbf{y})}{\partial \theta} \right] \\
&= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \mathcal{D}} \frac{\partial \Pr_{\theta}(\mathbf{y}|\mathbf{x})}{\partial \theta} R(\mathbf{y}) \right] \\
&= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \mathcal{D}} \Pr_{\theta}(\mathbf{y}|\mathbf{x}) \frac{\partial \Pr_{\theta}(\mathbf{y}|\mathbf{x}) / \partial \theta}{\Pr_{\theta}(\mathbf{y}|\mathbf{x})} R(\mathbf{y}) \right] \\
&= \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\sum_{\mathbf{y} \in \mathcal{D}} \Pr_{\theta}(\mathbf{y}|\mathbf{x}) \frac{\partial \log \Pr_{\theta}(\mathbf{y}|\mathbf{x})}{\partial \theta} R(\mathbf{y}) \right] \tag{11}
\end{aligned}$$

We assume that every output in $\mathcal{D}$ is equally probable (i.e., $\Pr_{\theta}(\mathbf{y}|\mathbf{x}) = 1/|\mathcal{D}|$). In this case we can simplify Eq. (??) and need only consider the terms $\frac{\partial \log \Pr_{\theta}(\mathbf{y}|\mathbf{x})}{\partial \theta}$ and $R(\mathbf{y})$:

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\frac{1}{d} \sum_{\mathbf{y} \in \mathcal{D}} \frac{\partial \log \Pr_{\theta}(\mathbf{y}|\mathbf{x})}{\partial \theta} R(\mathbf{y}) \right] \tag{12}$$

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

We can understand this optimization process from the perspective of hypothesis space reranking, as illustrated in Figure ?. The objective is to increase the probability of longer outputs in the hypothesis space by assigning a high reward to those outputs, while simultaneously decreasing the probability of shorter

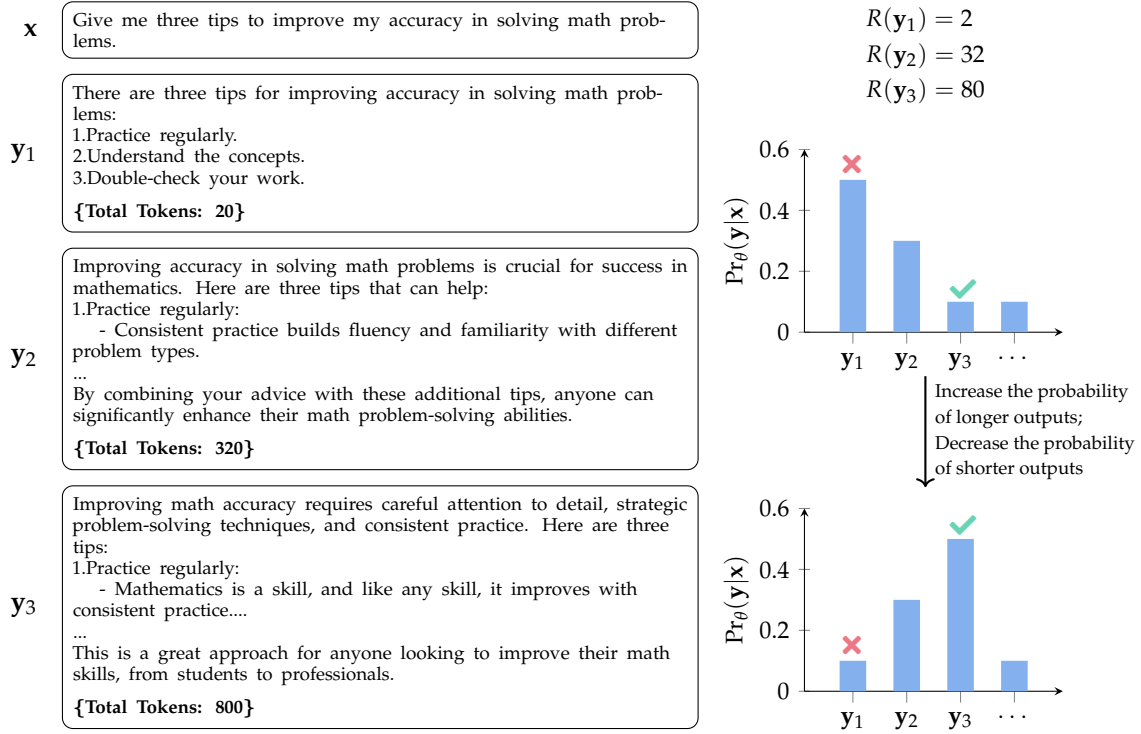


Figure 3: The mechanism of RL in training LLMs from the perspective of hypothesis space reranking: given an input x , the model increases the probability of longer outputs, such as y_3 , within the hypothesis space by assigning higher rewards to them; conversely, shorter outputs like y_1 receive lower rewards, which reduces their probabilities.

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. (??) is intuitive, a potential issue arises: in some cases, the sampled outputs we sample may significantly overlap, yet the rewards for the overlapping parts differ. For example, as illustrated in Figure ??, if outputs y_1 and y_2 both include the same tokens in the third tip “3.Double-check your work...any-one 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 generational behaviours 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. (??), and obtain

$$\begin{aligned}
 \frac{\partial \log \Pr_\theta(y|x)}{\partial \theta} R(y) &= \left(\sum_{t=1}^T \frac{\partial \log \Pr_\theta(y_t|x, y_{<t})}{\partial \theta} \right) \left(\sum_{t=1}^T r_t \right) \\
 &= \left(\sum_{t=1}^T \frac{\partial \log \Pr_\theta(y_t|x, y_{<t})}{\partial \theta} \right) \left(\sum_{k=1}^{t-1} r_k + \sum_{k=t}^T r_k \right)
 \end{aligned} \tag{13}$$

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 4: 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. (??):

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\frac{1}{d} \sum_{\mathbf{y} \in \mathcal{D}} \sum_{t=1}^T \frac{\partial \log \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t})}{\partial \theta} \sum_{k=t}^T r_k \right] \quad (14)$$

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 ??, 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. (??) 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. (??), 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 $\sum_{k=t}^T r_k - b$.⁴ Here, the baseline can be interpreted as a reference point. By centering the rewards around this baseline, we remove systematic biases in the reward.

⁴In 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

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\frac{1}{d} \sum_{\mathbf{y} \in \mathcal{D}} \sum_{t=1}^T \frac{\partial \log \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t})}{\partial \theta} \left(\sum_{k=t}^T r_k - b \right) \right] \quad (15)$$

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

$$\begin{aligned} A(\mathbf{x}, \mathbf{y}_{<t}, y_t) &= \sum_{k=t}^T r_k - b \\ &= \sum_{k=t}^T r_k - V(\mathbf{x}, \mathbf{y}_{<t}, y_t) \end{aligned} \quad (16)$$

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. (16) 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

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \left[\frac{1}{d} \sum_{\mathbf{y} \in \mathcal{D}} \sum_{t=1}^T \frac{\partial (\log \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t}) A_t)}{\partial \theta} \right] \quad (17)$$

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

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \mathbb{E}_{\mathbf{y} \sim \mathcal{D}} \left[\sum_{t=1}^T \log \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t}) A_t \right] \quad (18)$$

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 ?? illustrates this architecture of the value model. Hence, at t -th time step, we can compute the value V_t

$$V_t = \mathbf{h}_{y_t} \mathbf{W}_v \quad (19)$$

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

$$\mathcal{L}_v(\omega) = \frac{1}{T} \sum_{t=1}^T (r_t + V_{\omega, t+1} - V_{\omega, t})^2 \quad (20)$$

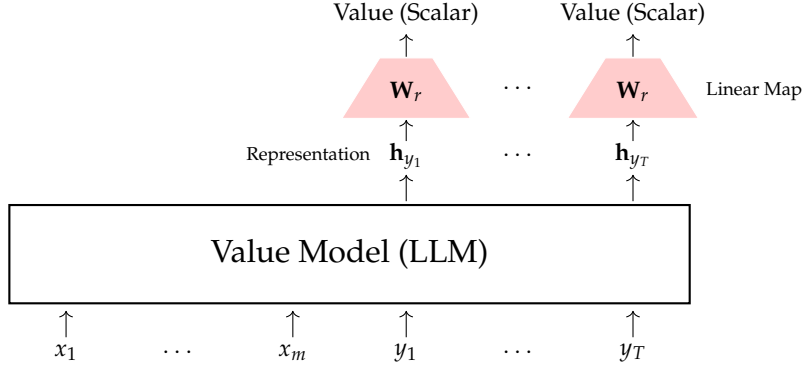


Figure 5: Architecture of the value model based on a pre-trained LLM. The main component of this model is still an LLM. For a given sequence $\text{seq}_{\mathbf{x}, \mathbf{y}} = [\mathbf{x}, \mathbf{y}]$, the model extracts representations corresponding to the positions of the tokens in $\mathbf{y}$. 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

$$\mathcal{L}_v(\omega) = \frac{1}{T} \sum_{t=1}^T (r_t + \gamma V_{\omega, t+1} - V_{\omega, t})^2 \quad (21)$$

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 (?).

In this subsection, we have detailed the integration of a value model in training LLMs. In practice, this training approach, represented by Eq. (??), is known as the Advantage Actor-Critic (A2C) method (?). 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 (??).

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 ?? 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

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \mathbb{E}_{\mathbf{y} \sim \text{Pr}_{\theta_{\text{ref}}}(\cdot | \mathbf{x})} \left[\sum_{t=1}^T \frac{\text{Pr}_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t})}{\text{Pr}_{\theta_{\text{ref}}}(y_t | \mathbf{x}, \mathbf{y}_{<t})} A_t \right] \quad (22)$$

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 log-probability of token y_t under the current and reference policies. This ratio function is used to reweight the observed rewards, reflecting how much more or less likely a token is under the current policy compared to the reference policy. When this

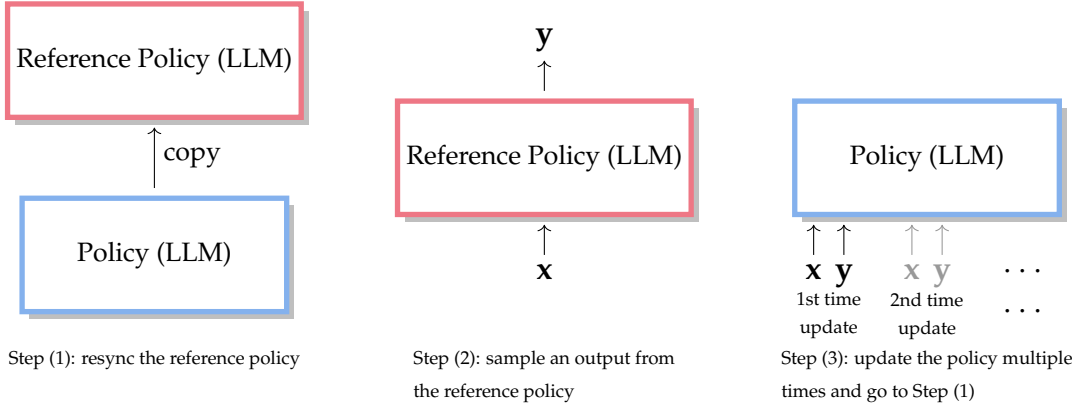


Figure 6: Workflow of integrating important sampling in the training of LLMs with policy gradient. In Step 1, the current policy is synchronized to establish the reference policy. In Step 2, we sample an output y from this reference policy for a given input x . Finally, in Step 3, the sequence $[x, y]$ is used to optimize the current policy multiple times. After optimization, the updated policy is synchronized to become the reference policy.

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 used 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 ??, 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. (??).

However, an inherent issue arises when using importance sampling to update LLMs. As shown in Figure ??, 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 SFT⁵, 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 (?), 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 y under the current policy versus the reference policy:

$$\text{Penalty} = \log \Pr_{\theta}(\mathbf{y}|\mathbf{x}) - \log \Pr_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \quad (23)$$

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

$$\text{Penalty}_t = \log \Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t}) - \log \Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t}) \quad (24)$$

⁵In supervised learning, a terrible step during a training step caused by bad samples can often be corrected in subsequent steps by good samples.

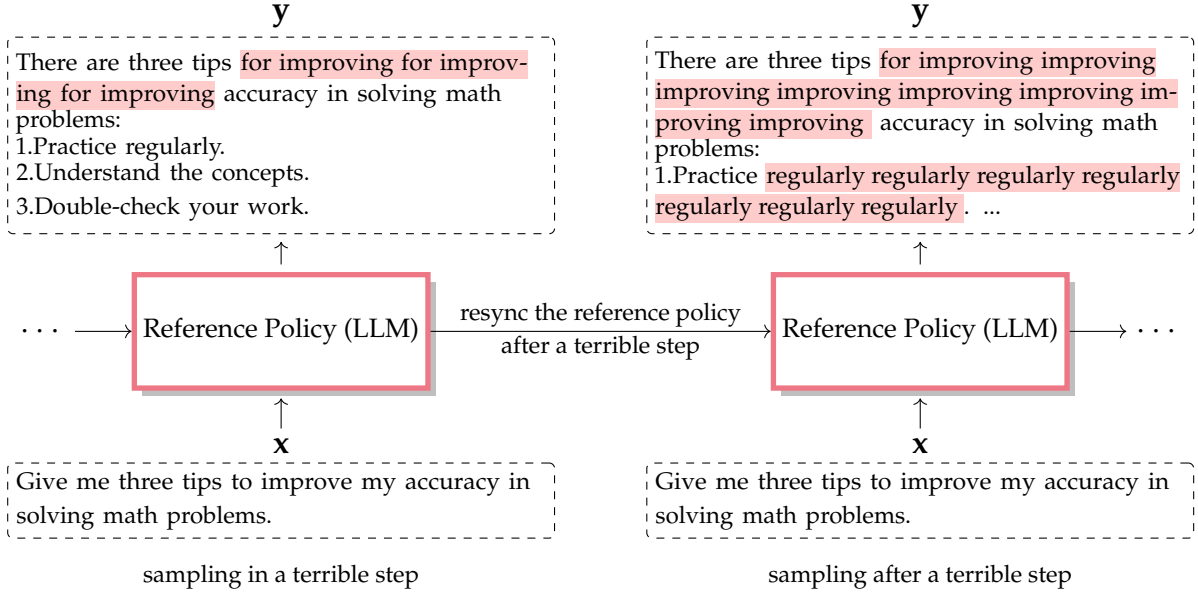


Figure 7: Illustration of the impact of a terrible step on sampling within an LLM training cycle using importance sampling. The left shows poor output sampled with slight repetition resulting in a terrible step. The right shows subsequent output sampled with more severe repetition after a terrible step.

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. (??), and obtain

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \mathbb{E}_{\mathbf{y} \sim \Pr_{\theta_{\text{ref}}}(\cdot|\mathbf{x})} \left[\sum_{t=1}^T \frac{\Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t})}{\Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t})} A_t - \beta \text{Penalty} \right] \quad (25)$$

where β is the weight of the penalty. This training method is also called trust region policy optimization (TRPO) (?).

3.5 Proximal Policy Optimization

A further improvement to TRPO involves addressing the gradient variance problem outlined in Section ??. In Eq. (??), 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

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \mathbb{E}_{\mathbf{y} \sim \Pr_{\theta_{\text{ref}}}(\cdot|\mathbf{x})} \left[\sum_{t=1}^T \text{Clip} \left(\frac{\Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t})}{\Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t})} A_t \right) - \beta \text{Penalty} \right] \quad (26)$$

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

$$\text{Clip} \left(\frac{\Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t})}{\Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t})} A_t \right) = \min \left(\frac{\Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t})}{\Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t})} A_t, \text{bound} \left(\frac{\Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t})}{\Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x}, \mathbf{y}_{<t})}, 1 - \epsilon, 1 + \epsilon \right) A_t \right) \quad (27)$$

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

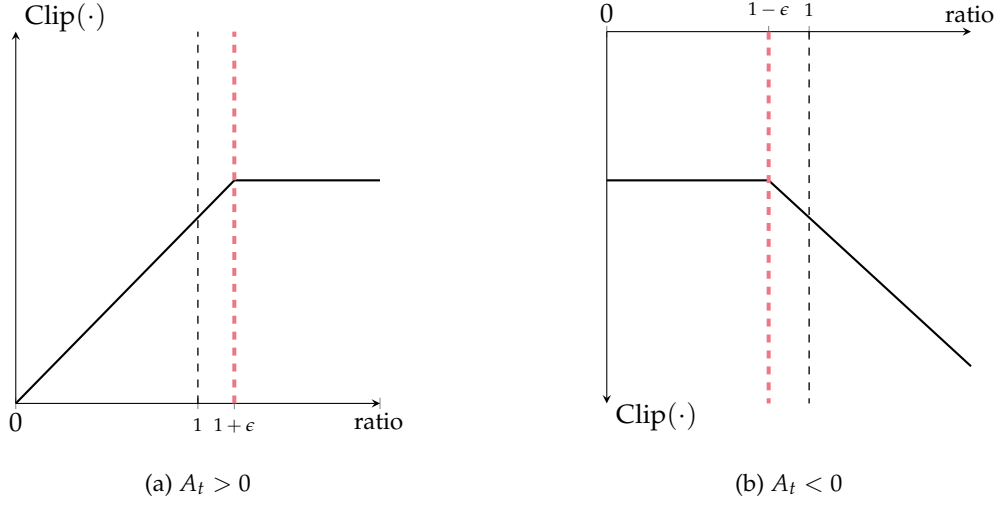


Figure 8: PPO clipping at time step t . The basic idea can be represented with a hypothesis space as described in Figure ?? . The sub-figure (a) shows the case where the advantage A_t is positive, suggesting that the output is preferred. Here, the probability of the sampled output is increased within 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) (?), 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 focused on output length, as described in Section ??, 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 ?? . 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 transformation⁶:

$$R_\phi(\mathbf{x}, \mathbf{y}) = \mathbf{h}_{y_T} \mathbf{W}_r \quad (28)$$

⁶In 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.

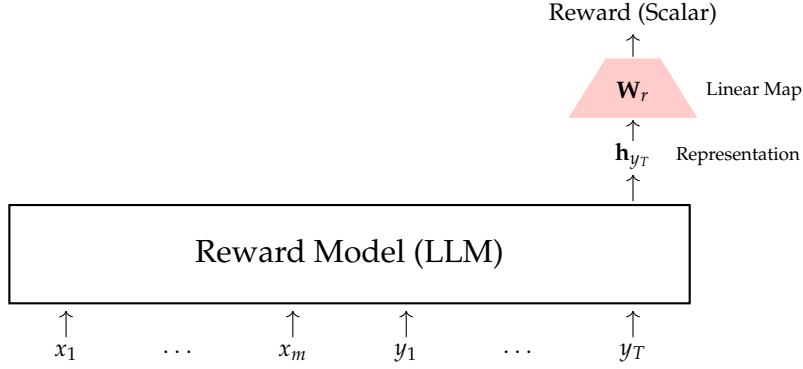


Figure 9: Architecture of the reward model based on an LLM. Unlike the value model depicted in Figure ??, this model extracts the representation from the last position of the output $\mathbf{y}$ to represent the entire sequence $[\mathbf{x}, \mathbf{y}]$. This representation is then mapped to a scalar 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 ??.

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 (?). 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

$$\begin{aligned}
 \Pr_{\phi}(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x}) &= \frac{e^{R_{\phi}(\mathbf{x}, \mathbf{y}_a)}}{e^{R_{\phi}(\mathbf{x}, \mathbf{y}_a)} + e^{R_{\phi}(\mathbf{x}, \mathbf{y}_b)}} \\
 &= \frac{e^{R_{\phi}(\mathbf{x}, \mathbf{y}_a) - R_{\phi}(\mathbf{x}, \mathbf{y}_b)}}{e^{R_{\phi}(\mathbf{x}, \mathbf{y}_a) - R_{\phi}(\mathbf{x}, \mathbf{y}_b)} + 1} \\
 &= \text{Sigmoid}(R_{\phi}(\mathbf{x}, \mathbf{y}_a) - R_{\phi}(\mathbf{x}, \mathbf{y}_b))
 \end{aligned} \tag{29}$$

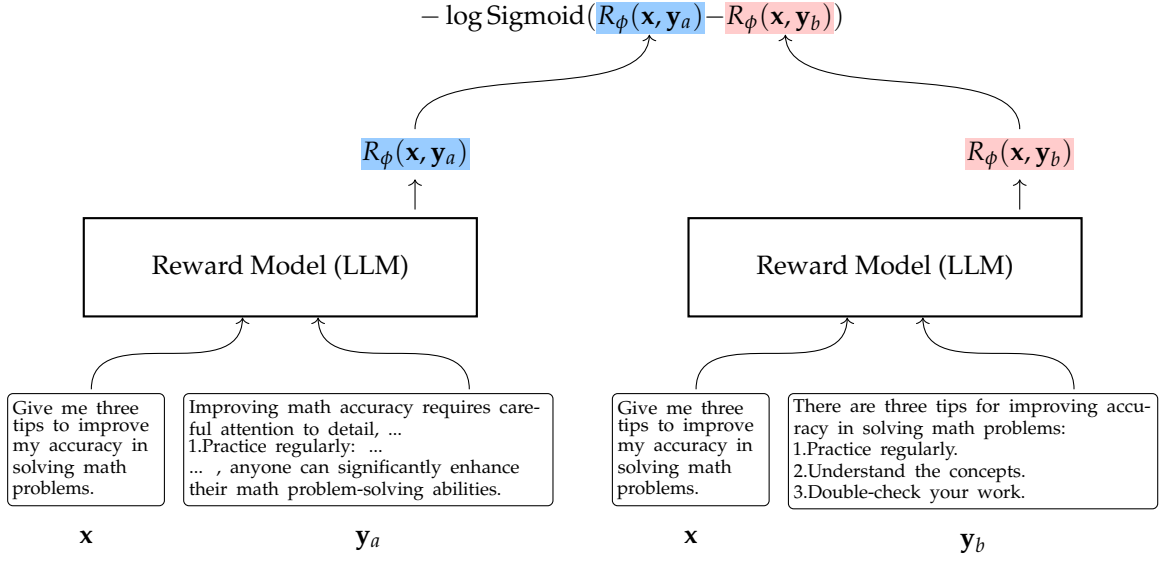


Figure 10: An example of loss computation using the Bradley-Terry model. Given an input x and two outputs (y_a, y_b) , where $y_a \succ y_b$, we initially obtain rewards $R(x, y_a)$ and $R(x, y_b)$ for two sequences seq_{x, y_a} and seq_{x, y_b} . These rewards are subsequently used to compute the loss as described in Eq. (??). Note that, for efficiency, both sequences are typically processed in a single batch during one forward pass to simultaneously obtain $R(x, y_a)$ and $R(x, y_b)$. In fact, this training approach is also common in Siamese networks (?).

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

$$\begin{aligned} \mathcal{L}_r(\phi) &= -\mathbb{E}_{(x, y_a, y_b) \sim \mathcal{D}_r} [\log \Pr_\phi(y_a \succ y_b | x)] \\ &= -\mathbb{E}_{(x, y_a, y_b) \sim \mathcal{D}_r} [\log \text{Sigmoid}(R_\phi(x, y_a) - R_\phi(x, y_b))] \end{aligned} \quad (30)$$

where (x, y_a, 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

$$\hat{\phi} = \arg \min_{\phi} \mathcal{L}_r(\phi) \quad (31)$$

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 ?? . After the training of the reward model, we can apply the trained reward model $r_{\hat{\phi}}(\cdot)$ 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

$$r_t = \begin{cases} 0 & \text{if } t < T \\ R_\phi(x, y) & \text{if } t = T \end{cases} \quad (32)$$

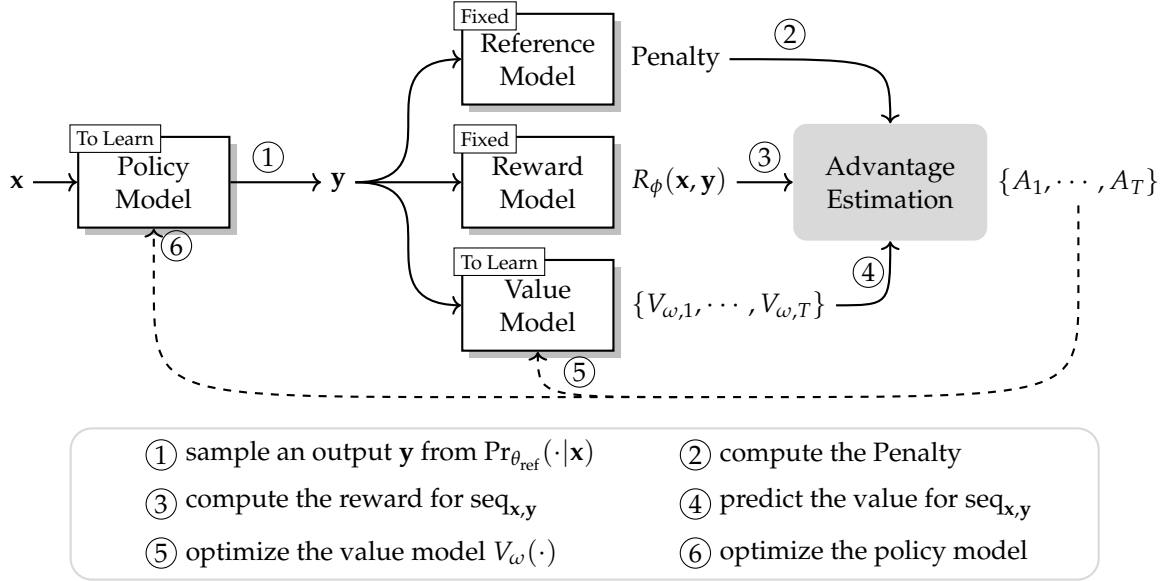


Figure 11: Illustration of PPO-based optimization workflow. Given an optimized reward model and a reference model, we proceed to train both the policy and the value model. At each prediction step, we compute the sum of the PPO-based loss and update the policy parameters. This requires access to the reward 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 (?) or train a process reward model by annotating process preference data (?). 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 $\Pr_{\theta_{\text{ref}}}(\cdot)$ 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 ??, 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 model⁷. 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 (?). 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 (?). 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 (?) and rule-based rewards (?), which can maintain state-of-the-art performance without requiring high computational and time costs.
- RL is often an unstable produce. 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.

4 Improved Reinforcement Learning for LLMs

In the previous section, we introduced some improvements to RL, such as importance sampling and reward baseline techniques. However, directly applying them to train LLMs still presents numerous challenges. In this section, we will delve deeper into the improvements for using RL to train LLMs.

⁷During 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 (?).

4.1 Advanced Reward Models

Reward models are a fundamental component in RL, as they define what a policy model optimizes for. Therefore, the quality of the reward model training directly influences the effectiveness of the LLM optimization during RLHF. A poorly trained reward model can lead to suboptimal LLM, while a well-optimized reward model ensures that the LLM is effectively aligned with the desired behaviours and objectives. Here we will introduce several methods for obtaining advanced reward models. Our discussion will be relatively general, and since the reward model is widely used in many RL problems, such as robot planning and control. This broad applicability makes it straightforward to adapt the methods discussed here to other related applications.

4.1.1 Automatic Preference Data Generation

Although learning from human preferences is an effective and popular method for aligning LLMs, annotating preference data is costly. Relying on human feedback not only faces scalability limitations but may also introduce bias, as human feedback is inherently subjective. Consequently, AI-based feedback methods offer a promising solution to address these scalability and consistency issues, avoiding the limitations associated with human annotators.

One simple method is to generate preference data using LLM. Given a set of inputs, we first use an LLM to generate pairs of outputs. Then, we prompt the LLM to label the preference between each pair of outputs, along with its corresponding input. Below is an example of prompting the LLM to generate a preference label for a pair of outputs.

Consider a homework assistant scenario in which a student types an input. You will review two outputs to this input. Please indicate which output you prefer. A good output should be informative, accurate, and well-articulated. It should directly address the student's question, provide thorough explanations or solutions, and maintain an encouraging tone.

Input:

Give me three tips to improve my accuracy in solving math problems.

Output A:

Improving math accuracy requires careful attention to detail, strategic problem-solving techniques, and consistent practice. Here are three tips:

1.Practice regularly:

- Consistent practice builds fluency and familiarity with different problem types...

...

By combining your advice with these additional tips, anyone can significantly enhance their math problem-solving abilities.

Output B:

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

1.Practice regularly.

2.Understand the concepts.

3.Double-check your work.

Output A is preferred.

Once we collect such preference labels, we can use them, along with the output pair and input, to train the reward model. Of course, these labels are not entirely accurate either. Recent studies have shown that AI-based feedback often exhibits a location bias problem, making it more likely to prefer the output at the

x Give me three tips to improve my accuracy in solving math problems.

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

y 1.Practice regularly.

2.Understand the concepts.

3.Double-check your work.

Intermediate Reward: +1

Intermediate Reward: +1

Delayed Reward: +10.6

Figure 12: An example of reward shaping. Here, length-based shaping rewards are implemented, e.g., awarding a +1 intermediate reward for every 10 tokens generated. Additionally, a delayed reward from the reward model evaluates the overall quality of the output at the end.

front (?). We can consider demonstrating a few examples or using advanced prompting techniques, such as Chain-of-Thought (CoT), to improve the labeling performance (?).

For data generation, although it is easy to scale up, it is often necessary to ensure the data is accurate and diverse. Here, the data quality and diversity issues involve not only the labeling of preferences but also the inputs and outputs of the model. Therefore, we often need to use a variety of techniques to obtain large-scale, high-quality data. For example, one can generate diverse model outputs and annotations by using different LLMs, prompts, in-context demonstrations, and so on (?). Furthermore, ? report that the variability in pairwise preference data is important for training LLMs from either human or AI feedback.

While learning from AI feedback is highly scalable and generally objective, this method is more suited to well-defined tasks where objective performance metrics are available. By contrast, learning from human feedback is more advantageous when aligning AI systems with human values, preferences, and complex real-world tasks that require an understanding of subtle or subjective context. These methods can be combined to train LLMs that benefit from both human insights and the scalability of AI feedback.

4.1.2 Reward Shaping

As discussed in Section ??, while the reward model is effective in capturing human preferences, it provides sparse rewards. These rewards, known as delayed rewards, are only at the end of the output generation process, as opposed to intermediate rewards that would be distributed continuously throughout it.

In fact, dealing with sparse rewards has long been a concern in RL, and has been one of the challenges in many practical applications. For example, robotics often needs to shape the reward function to ease optimization rather than relying solely on end-of-sequence rewards. Various methods have been developed to address this issue. One common approach is reward shaping, where the original function is modified to include intermediate rewards, thereby providing more immediate feedback. Here, the intermediate reward is often an indirect way to be able to improve this delayed reward. For example, as shown in Figure ??, setting a length-based reward as an intermediate reward encourages the generation of more content, potentially increasing the overall quality of the final output as assessed by the delayed reward from the reward model. Additional examples of reward shaping in training LLMs can be found in ?. In this way, we can obtain

$$r'_t = r_t + f(\mathbf{x}, \mathbf{y}_{<t}, y_t) \quad (33)$$

where $r'(\cdot)$ is the transformed reward, $r(\cdot)$ is the original delayed reward function, and $f(\cdot)$ is the shaping reward function that provides an intermediate reward. To ensure the optimality of the policy under the

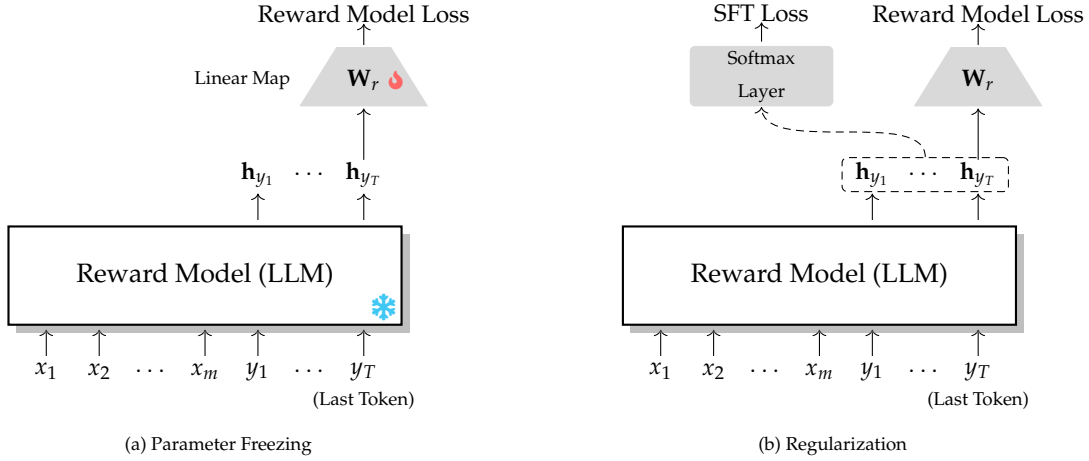


Figure 13: When training reward models, we can use parameter freezing and regularization methods to preserve the features of the LLM and thereby improve reward generalization.

transformed reward function, the shaping reward function can be given in the form

$$f(\mathbf{x}, \mathbf{y}_{<t}, y_t) = \gamma \Phi(\mathbf{x}, \mathbf{y}_{<t+1}, y_{t+1}) - \Phi(\mathbf{x}, \mathbf{y}_{<t}, y_t) \quad (34)$$

where $\Phi(\cdot)$ is called the potential value function. If we define $\Phi(\cdot)$ as the common value function and substitute Eq. (??) into Eq. (??), we obtain

$$r'_t = r_t + \gamma V_{t+1} - V_t \quad (35)$$

It is interesting to see that this function is exactly the same as the advantage function used in PPO. This relates advantage-based methods to reward shaping: the advantage is essentially a shaped reward.

In practice, an alternative method for implementing reward shaping in training LLMs involves utilizing the reward model to provide intermediate rewards based on the differences between output rewards (??). For example, at time step t , the intermediate reward r_t can be obtained by

$$r_t = R(\mathbf{x}, \mathbf{y}_{<t+1}) - R(\mathbf{x}, \mathbf{y}_{<t}) \quad (36)$$

However, in this case, it is crucial that the reward model is capable of accurately assigning rewards for partial outputs, i.e., $\{\mathbf{y}_{<2}, \mathbf{y}_{<3}, \dots, \mathbf{y}_{<T}\}$.

In addition to reward shaping, another method to address the sparse reward issue is adopting curriculum learning, where tasks are structured sequentially with gradually increasing complexity. This can help models master simpler tasks first, which prepares them for more complex challenges as their skills develop. There are many methods that can mitigate the impact of sparse rewards, such as Monte Carlo methods and intrinsic motivation. Most of these methods are general, and the discussion of them can be found in the broader literature on RL, such as ?'s book.

4.1.3 Improved Reward Generalization

As discussed in Section ??, reward models are trained using preference data and subsequently used to optimize LLMs. However, a problem arises: the data distribution during LLM optimization may differ from the distribution of the preference data, making it challenging for the reward model to generalize to unseen input-output pairs.

A well-known failure mode associated with this problem is commonly referred to as *overoptimization* or *reward hacking*, where the optimization stage improves the reward model but deteriorates the alignment

with true rewards (??). This mode occurs because the reward model may incorrectly assign high rewards to unseen input-output pairs, leading the LLM to learn and optimize for behaviours that do not truly align with the desired behaviours and objectives. For example, consider a scenario from Section ?? where the reward model is designed to favor informative and accurate outputs for a homework assistant. However, if the reward model fails to generalize to an overly verbose output and assigns a high reward to it (perhaps due to its length or certain keywords), the LLM may learn to prioritize generating a long output, which is not actually more informative but reward better according to the reward model. Consequently, the LLM becomes misaligned with its true objectives—delivering concise and relevant information—because it optimizes for the incorrect reward signal from the reward model with weak generalization.

Addressing this generalization problem is challenging, and no mature solution exists yet. The ideal approach would be to develop an oracle reward model that perfectly captures the true objectives of the task and generalizes across all input-output pairs during LLM optimization. However, creating such a model is extremely difficult due to the complexity of the real-world environment, as well as the challenge of collecting sufficient preference data. Instead, a more practical approach is to combine multiple reward models, improving generalization and providing more correct rewards (?).

Given a set of reward models, combining them is straightforward, and in some cases, we can simply treat this problem as an ensemble learning problem. A simple yet common approach is to average the outputs of these models to obtain a more precise reward estimation:

$$R_{\text{combine}}(\mathbf{x}, \mathbf{y}) = \frac{1}{K} \sum_{k=1}^K w_k \cdot R_k(\mathbf{x}, \mathbf{y}) \quad (37)$$

where $R_k(\cdot)$ is the k -th reward model in the ensemble, w_k is the weight of $r_k(\cdot)$, and K is the number of reward models. This combined reward can then be used to supervise the training of a policy. In fact, there are many ways to combine different models; for example, one can make predictions using Bayesian model averaging or develop a fusion network to learn to combine the predictions from different models. Alternatively, one can frame this task as a multi-objective optimization problem and use multiple reward models to train the policy simultaneously. These methods have been intensively discussed in the literature on optimization and machine learning (??).

On the other hand, to improve the generalization of reward models, it is important to prevent overfitting to preference data. Reward models are typically trained on an SFT LLM, which has a strong generalization capability. However, training the LLM with a large amount of preference data could lead to overfitting, which may ultimately reduce generalization performance. A common strategy to mitigate this issue is to employ a parameter freezing technique, which helps preserve the original features of the LLM while learning the reward model. Another practical approach is to add a regularization term to the preference learning loss function, which helps regulate the features of the LLM (?). For example, we can use a simple SFT loss as regularization by adding a term that maximizes the probability of the preferred output $\mathbf{y}_a$ to Eq. (??):

$$\mathcal{L}_{\text{reg}}(\phi) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} [\log \Pr_{\phi}(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x}) + \alpha \log(\Pr_{\theta}(\mathbf{y}_a | \mathbf{x}))] \quad (38)$$

where α is a balancing factor, we further illustrate the freezing parameter and regularization methods in Figure ?. Note that the regularization term applies only to the LLM. That is, when optimizing this term, only the parameters of the LLM are updated, while the parameters of the reward linear map remain fixed. Therefore, in this equation, the parameter associated with the regularization term is θ , which specifically refers to the parameters of the LLM.

4.1.4 Generative Reward Models

Reward models are typically trained as discriminative models to assign numerical rewards to outputs and classify them as preferred or dispreferred. However, this method does not leverage the text-generation

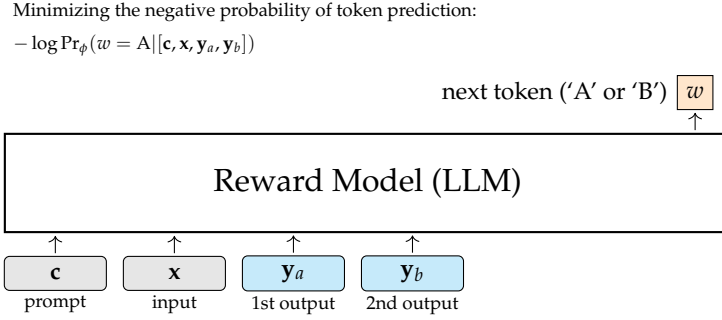


Figure 14: In generative reward models, we use an LLM to predict the label token given a prompt, an input, and a pair of outputs. This model can be trained in the same way as standard LLMs.

capabilities for which LLMs are fundamentally designed (??). For example, the discriminative reward model can not perform CoT reasoning. To address this, an LLM can alternatively be employed as a reward model, thus endowing it with the ability to engage in text generation and reasoning, as depicted in Figure ???. This model works as follows. First, we input a prompt c , along with the tuple (x, y_a, y_b) , to the LLM. The prompt is a description of the task, as demonstrated in the example below.

You are given two outputs to an input. Evaluate which output is better based on quality, relevance, and clarity. If the first output is better, return 'A'. If the second output is better, return 'B'.

Then, the LLM predicts subsequent tokens based on this input sequence. Let w be the label token predicted by the LLM. If $w = A$, it indicates a preference for y_a over y_b ; if $w = B$, then y_b is preferred. Note that here y_a and y_b do not have a pre-defined preference relationship as described in Section ???. Their relationship is instead represented by the label token.

The loss function can be defined as the log-probability of predicting 'A':

$$\mathcal{L}_{\text{gen}}(\theta) = -\mathbb{E}_{(c, x, y_a, y_b) \sim \mathcal{D}_r} [\log \Pr_{\theta}(w = A | \mathbf{s})] \quad (39)$$

where $\mathbf{s}$ denotes the string $[c, x, y_a, y_b]$ ⁸, and $\pi_{\phi}(\cdot)$ denotes the probability of token prediction by the LLM.

Although we discuss methods for training a generative reward model here, an interesting question arises: how can we use the generative reward model in RLHF? We provide some guidance as follows. Specifically, when applying this generative reward model to provide a reward for an input-output pair (x', y') , we can generate a reference output y_{ref} by using the LLM, for example, through greedy search, and concatenate x' , y' and y_{ref} into $\mathbf{s}' = [c, x', y', y_{\text{ref}}]$. Additionally, to mitigate the positional bias problem (?), we can introduce an alternative input order by transposing the positions of output, i.e., presenting y_{ref} before y' , to construct a secondary input string $\mathbf{s}'_T = [c, x', y_{\text{ref}}, y']$. The reward for (x', y') is thus defined as the log-probability that y' is preferred over y_{ref} :

$$r_{\phi}(x', y') = \frac{\Pr_{\theta}(w = A | \mathbf{s}') + \Pr_{\theta}(w = B | \mathbf{s}'_T)}{2} \quad (40)$$

where the reward ranges from 0 to 1.

⁸In this work, we will use $\mathbf{s}$ interchangeably to refer to either the tuple of a training sample or a string representing that tuple.

To further improve the generative reward model, we can label the relevant explanation to enable the generative reward model to produce a CoT rationale (??). In this case, we use a prompt $\mathbf{c}$ with a CoT rationale generation instruction, as shown below.

You are given two outputs to a user input. Evaluate which output is better based on quality, relevance, and clarity. If the first output is better, return ‘A’. If the second output is better, return ‘B’. Note that, before presenting the evaluation results, you should provide a detailed analytical process and reasoning steps.

We can then define a new loss function for training the generative reward model as follows:

$$\mathcal{L}_{\text{gen}}(\theta) = -\mathbb{E}_{(\mathbf{c}, \mathbf{x}, \mathbf{y}_a, \mathbf{y}_b, \mathbf{rat}) \sim \mathcal{D}_r} [\log \Pr_{\theta}(\mathbf{rat}|\mathbf{s}) + \log \Pr_{\theta}(w = \mathbf{A}|\mathbf{s}, \mathbf{rat})] \quad (41)$$

where $\mathbf{rat}$ is the labeled CoT rationale for generating a preference between $\mathbf{y}_a$ and $\mathbf{y}_b$. In practice, we can obtain the evaluation results by prompting a standard LLM (as known as LLM-as-a-judge), as demonstrated in Section ?? . However, this approach typically underperforms compared to LLMs trained with preference data (??). One reason is that standard LLMs are not fine-tuned specifically for the reward modeling task, and as a result, they may not fully capture the nuanced decision-making process that aligns better with human preferences. Also, LLMs trained with preference data learn to prioritize outputs based on feedback, enhancing their capability to evaluate outputs according to the desired behaviors and objectives.

4.2 Better Advantage Estimation

Accurate advantage estimation is critical in RL, particularly when optimizing the policy model to truly reflect the potential benefits of different actions (or tokens in training LLMs). In this subsection, we will delve into methods for improving advantage estimation, providing more accurate advantages in the process of optimizing the policy model.

4.2.1 Temporal Difference-based Advantage Estimation

Let us first review the Monte Carlo-based advantage estimation. As discussed in Section ?? , outputs are sampled and their values computed. The advantage can be obtained by comparing the actual received rewards with the predicted values at time step t :

$$A_t = \sum_{k=t}^T r_k - V_t \quad (42)$$

While the Monte Carlo-based estimation of advantage provides a relatively stable training process, there are two main challenges associated with it:

- Sampling an entire output for each input is necessary. In some cases where immediate rewards are available, it could be more efficient to update the policy model with partial output. Unfortunately, when using Eq. (??) for advantage estimation, this becomes unfeasible. This is because we must compute the term $\sum_{k=t}^T r_k$, which requires the rewards of the entire output.
- This method still results in high variance. Since a single sample operation might be influenced by random factors, the estimated results may not be stable, as illustrated in Figure ??.

To address these challenges, an alternative approach is the temporal difference-based advantage estimation (?), which allows for estimating the advantage using incomplete outputs. More specifically, this method computes the difference between the predicted values at consecutive time steps (i.e., current and next stapes), adjusting the advantage estimate based on new information as it becomes available, thus reducing

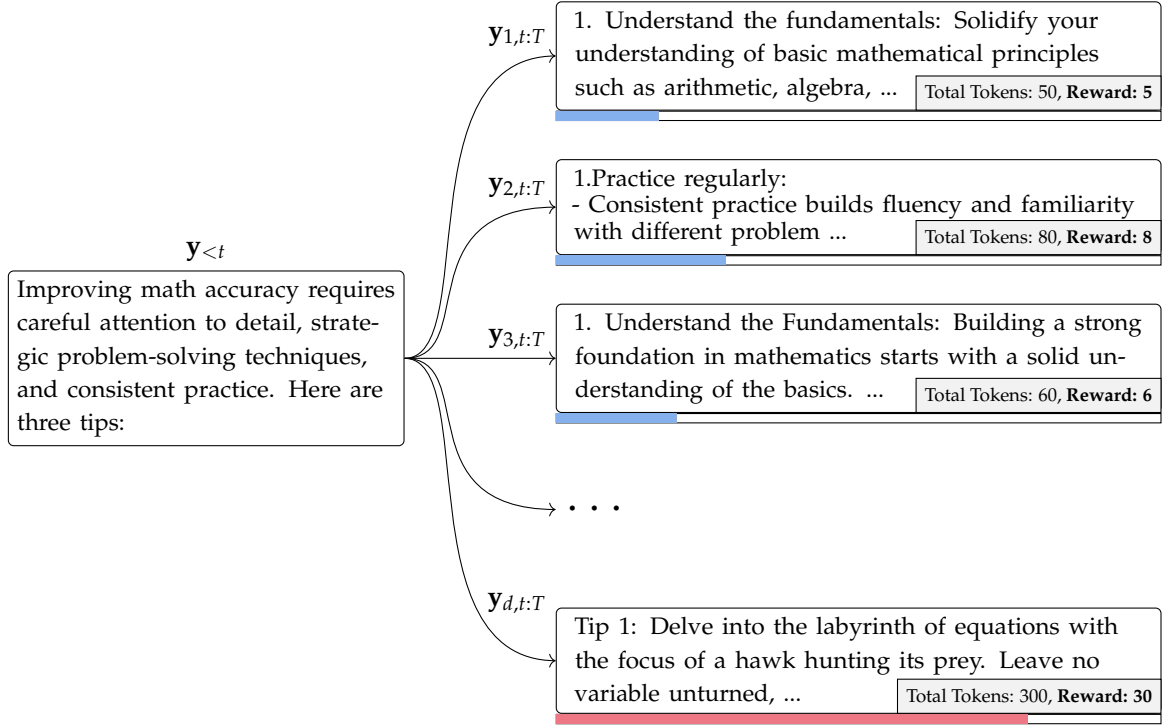


Figure 15: Illustration of the issue of high variance in Monte Carlo-based advantage estimation, using a length-based reward function as described in Eq. (??). Typically, if the value model is well-trained, it would predict an average V_t of 6 at time step t , based on most outputs having around 60 tokens in length. However, for outlier outputs like $y_{d,t:T}$, which might extend up to 300 tokens, the advantage estimation can exhibit high variance. For instance, the advantage for such an outlier could be computed as $V_t(\mathbf{x}, y_{d,<t}, y_{d,t}) = \sum_{k=t}^T r_k - V_t = 30 - 6 = 24$, a stark contrast to another output, say $y_{1,t:T}$, which is closer to the average length, where the advantage might be $V_t(\mathbf{x}, y_{1,<t}, y_{1,t}) = -1$. This discrepancy leads to high gradient variance, potentially destabilizing the learning process.

the dependency on the entire output. In a practical implementation, the temporal difference-based advantage estimation can be given by

$$A_t^{\text{TD}} = r_t + \gamma V_{t+1} - V_t \quad (43)$$

By focusing on the current and next steps, temporal difference-based advantage estimation minimizes the influence of distant future events. This leads to lower variance in optimizing the policy model and improves the stability of policy training.

4.2.2 Generalized Advantage Estimation

While temporal difference-based estimation effectively reduces variance, it can also increase estimation bias due to its heavy reliance on the predictions of the value model. By contrast, one effective method is generalized advantage estimation (GAE), which is one of the most popular advantage estimation methods used in LLMs and other fields (?). This method mainly refines the advantage estimation by using exponentially weighted averages of temporal difference advantages across multiple future steps. More specifically, consider a bias δ_t in the temporal difference at time step t , defined as

$$\delta_t = r_t + \gamma V_{t+1} - V_t \quad (44)$$

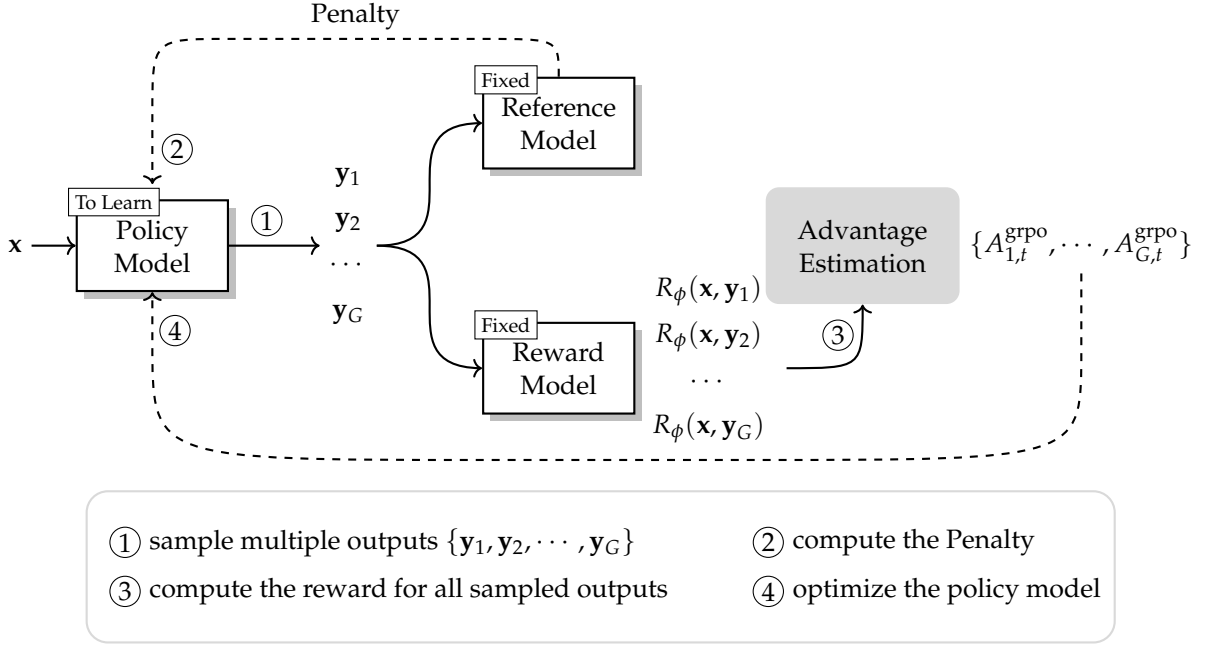


Figure 16: Group relative policy optimization. Compared to PPO, it foregoes the value model by estimating advantages directly from the rewards of a group of outputs. This method can simplify the training process and reduce the computational resources.

GAE introduces parameters that dynamically balance the trade-off between bias and variance. This basic idea is to use the weighted sum of multi-step temporal difference bias as an estimation of the advantage, combining the advantages of Monte Carlo (high variance and low bias) and temporal difference methods (low variance and high bias). This combination can be expressed as

$$A_t^{gae} = \sum_{n=1}^{T-t} (\gamma\lambda)^n \delta_{t+n} \quad (45)$$

We can recursively develop the above model:

$$A_t^{gae} = \delta_t + \gamma\lambda A_{t+1}^{gae} \quad (46)$$

This recursive formulation shows that the current advantage estimate incorporates not only the immediate temporal difference error at time t but also the estimated advantages of future time steps, weighted and adjusted by the parameters γ and λ . This method effectively combines the strengths of both Monte Carlo and temporal difference methods, leading to a more stable and efficient training process.

4.2.3 Group Relative Policy Optimization

Although the methods discussed in the previous subsections effectively estimate advantage during the learning process, they all rely on a value model that is typically trained concurrently with the policy model. This dependence not only complicates the training process but also requires significant computational resources when applied to training LLMs. Given this, a natural idea arises: could we use more lightweight methods to compute the advantage without relying on the value model? By doing so, we could forego the value model component, thereby improving efficiency.

In fact, this is entirely feasible, and one example of such an approach is ?’s approach, called group relative policy optimization or GRPO for short. In the GRPO approach, the advantage is computed based on the

relative rewards of the outputs within each group. More specifically, given an input $\mathbf{x}$, GRPO samples a group of outputs $\mathbf{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_G\}$. Then, the rewards for each output are computed using a reward model or a reward function, denoted as $\mathbf{R} = \{R(\mathbf{x}, \mathbf{y}_1), R(\mathbf{x}, \mathbf{y}_2), \dots, R(\mathbf{x}, \mathbf{y}_G)\}$. The advantage of the i -th output can be computed through a relative reward in comparison to the other outputs:

$$A_{i,t}^{\text{grpo}} = \frac{R(\mathbf{x}, \mathbf{y}_i) - \text{Mean}(\mathbf{R})}{\text{Std}(\mathbf{R})} \quad (47)$$

where the $\text{Mean}(\cdot)$ and $\text{Std}(\cdot)$ represent the mean and standard deviation functions, respectively. Note that the advantage computation used here differs slightly from that in PPO. In this case, this computed advantage is applied to each time step, whereas in the traditional PPO, we separately compute an advantage for each time step. At this point, we can also use the process reward model to supplement the advantage computation, allowing us to compute an advantage specific to each time step. A simple way to achieve this is by implementing Eq. (??) at each time step, where the rewards are computed using the process reward model. As a result, the objective for GRPO can be defined according to Eq. (??):

$$\mathcal{L}_g(\theta) = -\mathbb{E}_{\mathbf{x} \sim \mathcal{S}_x} \frac{1}{G} \sum_{i=1}^G \left[\sum_{t=1}^T \text{Clip} \left(\frac{\Pr_{\theta}(y_{i,t} | \mathbf{x}, \mathbf{y}_{i,<t})}{\Pr_{\theta_{\text{ref}}}(y_{i,t} | \mathbf{x}, \mathbf{y}_{i,<t})} A_{i,t}^{\text{grpo}} \right) - \beta \text{Penalty} \right] \quad (48)$$

In addition to optimizing the advantage computation, GRPO modifies the penalty term to enhance the optimization objective further. However, since our focus here is on advantage estimation, we will not delve into its details. Interested readers can refer to the GRPO paper for further details. The workflow of GRPO is illustrated in Figure ??, showcasing how these modifications integrate into the training process.

Moreover, there are other methods like GRPO that design advantage estimation without relying on a value model for training LLMs, such as those discussed in ? and ?.

4.3 Efficient Reinforcement Learning Methods

Efficiency is a critical consideration for many practical applications of RL, and it becomes even more significant in the context of LLMs due to their vast number of parameters. For example, we might wish to train LLMs using RL, given memory and time constraints. In practice, the efficiency of RL is not a single issue but encompasses a wide range of challenges. While these challenges can be categorized in various ways in the existing literature, we focus on two fundamental aspects: *time* and *space* efficiency, which are commonly considered in efficiency-related problems. For these two efficiency problems, we aim to achieve the desired RL performance with the least amount of time and memory required.

In this section, we will not discuss all the issues related to the efficiency of RL, which is an extensive topic. Instead, we will focus on the commonly used efficient methods for training LLMs with RL. Some of these methods refine the sampling process, while others aim to eliminate certain components, such as the reward model, and utilize alternative lightweight methods in their place. Nonetheless, although these efficient methods are suggested for training LLMs, they are rather general and can be utilized in other RL scenarios.

4.3.1 Dynamic Sampling

As mentioned in Section ??, training LLMs with RL typically requires sampling for each sample $\mathbf{x}$ in $\mathcal{S}_x$, which introduces significant computational time overhead. This becomes particularly challenging in large-scale RL scenarios, where thousands of training steps need to be conducted, such as in DeepSeek-R1 (?).

From the perspective of LLM inference, there are many methods to reduce the time overhead of sampling: since the sampling process is implemented by LLM inference, we have reason to believe that any method that accelerates inference can also be applied to reduce the time required for sampling. Examples include KV-Cache (?), quantization (?), and speculative decoding (?). However, in this section, we will not discuss

these methods in detail, as there are numerous approaches, each addressing different aspects of LLM inference. We refer the interested readers to these papers for more details. Instead, we will focus on one particular method that aims to reduce sampling overhead from the perspective of optimizing RL for training LLMs.

Before discussing specific methods, let us first review the purpose of sampling. Given a sample $\mathbf{x}$, the goal of sampling is to explore a better output $\mathbf{y}$ from the policy model. This process allows us to evaluate different possible outputs in order to find those that lead to improved outcomes, enabling the model to make more informed decisions and optimize its performance on the state. Unlike typical RL scenarios, where policy models may start with limited information or random initialization, the policy model in the context of LLMs is usually quite advanced. These LLMs often begin as pre-trained models equipped with substantial knowledge acquired through pre-training and SFT. As a result, for certain samples, we can consider that the policy model may already perform well without the need for further exploration and optimization. This can be viewed through the lens of the classic RL dilemma of exploration versus exploitation (?). In such cases, exploration may be less necessary, and exploiting the knowledge already embedded in the pre-trained model could be more effective. Thus, a balance must be struck between exploring new possibilities and leveraging the pre-existing knowledge of the policy model to maximize efficiency.

To achieve this balance between exploration and exploitation, one simple and straightforward approach to improving the efficiency of sampling is to identify the samples that are required to explore and perform sampling only for those samples. Given an input-only dataset $\mathcal{S}_x = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$, where N denotes the size of the dataset, we can use a reward model to evaluate the need for exploration by the policy model. First, we generate $\hat{\mathbf{y}}$ for each $\mathbf{x}$ using greedy search. Then, by taking $\mathbf{x}$ and $\hat{\mathbf{y}}$ as input to the reward model, the reward for the k -th sample $\mathbf{x}_k$ is computed as $R_\phi(\mathbf{x}_k, \hat{\mathbf{y}}_k)$.

At this point, we can directly use the reward value to determine whether the policy model requires further exploration and optimization for the sample. Specifically, we set a threshold r_{upper} . If the reward for a sample exceeds r_{upper} , we can consider that no further sampling and optimization is required. Otherwise, exploration and optimization are required. This approach enables selective sampling and optimization, improving the efficiency of the overall process by focusing resources on the most promising samples that are likely to benefit from further exploration (?). A challenge in implementation is determining the value of r_{upper} . Typically, we set it as the average reward:

$$r_{\text{upper}} = \frac{1}{N} \sum_{i=1}^N R_\phi(\mathbf{x}_i, \hat{\mathbf{y}}_i) \quad (49)$$

A similar dynamic sampling strategy, known as prioritized sampling, has proven effective in large-scale RL scenarios (?). However, we must consider another important factor in dynamic sampling. For samples with very low rewards, the policy model may lack the capacity to learn and optimize effectively. Excessive exploration of such samples can be inefficient. Consequently, we can introduce an additional threshold r_{lower} to eliminate unnecessary exploration of these low-reward samples, thereby enhancing efficiency. In this case, the policy model can concentrate its exploration and optimization efforts on a select group of samples, avoiding wasted sampling time on those that are unlikely to explore a more optimal output (?).

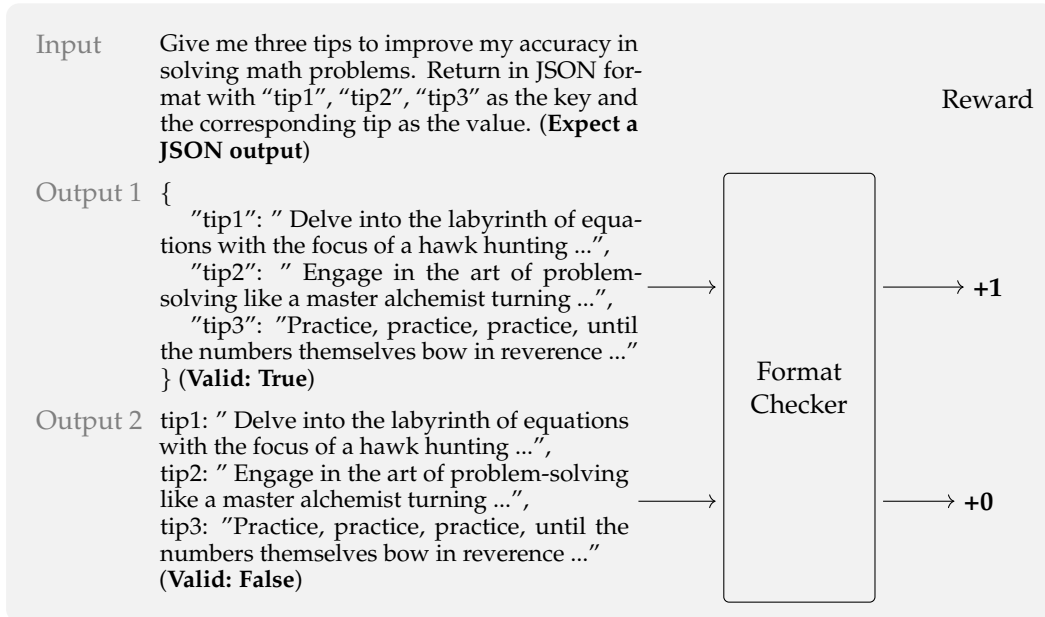
A concept closely related to the discussion here is sample efficiency. This topic has been widely discussed in recent literature. For example, some research on knowledge distillation seeks to select a smaller subset of samples to achieve more optimal performance (?). More recently, much work on instruction sample selection in SFT aims to improve SFT efficiency by learning from a small number of instruction samples (?).

4.3.2 Lightweight Reward Methods

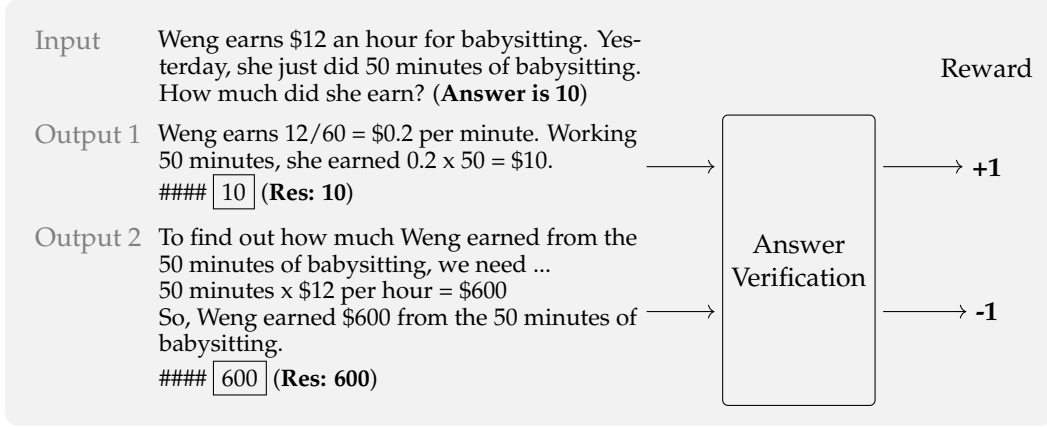
An additional computational overhead in training LLMs with RL is the frequent need to call the reward model to compute rewards. Furthermore, to ensure the reliability and generalizability of the reward model, we often scale it to a large size (?), which increases the computational time for reward computations.

One approach for mitigating this issue is to explore rule-based rewards, for example, as discussed in Section ??, the length of the outputs could be utilized as a reward. Instead of relying on large-scale, resource-intensive models, rule-based rewards assign rewards using predefined and task-specific rules. These rules, often derived from expert knowledge or task-specific requirements, are computationally inexpensive and provide fast, effective feedback to the policy model. By replacing or complementing traditional reward models with rule-based approaches, we can reduce both the time and computational cost associated with reward computations while still offering meaningful guidance for the learning process. This approach is also based on the understanding that, in some tasks, human preferences are simple to describe and do not require the complexity of training a reward model. Instead, we can use rules to express these preferences efficiently.

Here, in addition to the previously mentioned length-based rewards, we further demonstrate two examples of rule-based rewards. First, considering the output format, we can design rewards that encourage the policy model to generate outputs adhering to specific formats. For example, in response to the input "Give me three tips to improve my accuracy in solving math problems," we can assign rewards based on whether the output can be parsed as JSON correctly and include "tip1", "tip2", and "tip3" as the keys.



Similarly, in mathematical reasoning tasks, where the desired outcome is a correct final answer, we can design a rule to verify the correctness of the answer and provide rewards based on whether the result is correct (??).



In addition to improving computational efficiency, another benefit of using rule-based rewards is their stability. Compared to reward models, rule-based rewards are less prone to issues such as reward hacking. As the rules are predefined and not subject to the same complex training process as traditional models, the risk of misalignment between the optimization objectives of the policy model and the oracle rewards is reduced. This stability makes rule-based rewards a more predictable and reliable approach, ensuring that the learning process of the policy model is less influenced by prediction errors or biases that may arise from the reward model. Furthermore, recent research has shown that employing rule-based rewards can significantly enhance the reasoning capabilities of LLMs through RL, underscoring its practical effectiveness (??).

Another lightweight reward method is to consider achieving a lightweight reward model with fewer parameters, which would reduce both computational time and resource usage. There are several methods to achieve this. For example, given that reward models fundamentally utilize an LLM, many efficient methods, such as Mixture of Experts (MoE) (?) and model pruning (?), originally developed for LLMs can be adapted to the reward model to improve its efficiency. Furthermore, knowledge distillation techniques offer a pathway to construct a smaller reward model that learns to emulate the behaviour of a larger one (?). These approaches not only improve the efficiency of the reward model but also maintain its ability to deliver accurate rewards.

4.4 Direct Preference Optimization

Although learning reward models is a standard step in RL, it makes the entire training process much more complex than supervised training. Training a reliable reward model is itself not an easy task and a poorly trained reward model can greatly affect the outcome of policy learning. We now consider an alternative method, called direct preference optimization (DPO), which simplifies the training framework by eliminating the need to explicitly model rewards (?). This method directly optimizes the policy model based on user preferences rather than developing a separate reward model. As a result, we can achieve human preference alignment in a supervised learning-like fashion. Figure ?? shows a comparison of the standard PPO method and the DPO method.

DPO simplifies the RL process for LLMs by utilizing a straightforward cross-entropy loss, which streamlines the learning process and enhances its stability. Rather than delving into the detailed derivation of the DPO loss function, we will discuss its optimization objective from the perspective of optimizing an LLM. The loss function derived from a Bradley-Terry reward model can be given by

$$\mathcal{L}_{\text{dpo}}(\theta) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r} \left[\log \text{Sigmoid} \left(\beta \log \frac{\Pr_{\theta}(\mathbf{y}_a | \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_a | \mathbf{x})} - \beta \log \frac{\Pr_{\theta}(\mathbf{y}_b | \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_b | \mathbf{x})} \right) \right] \quad (50)$$

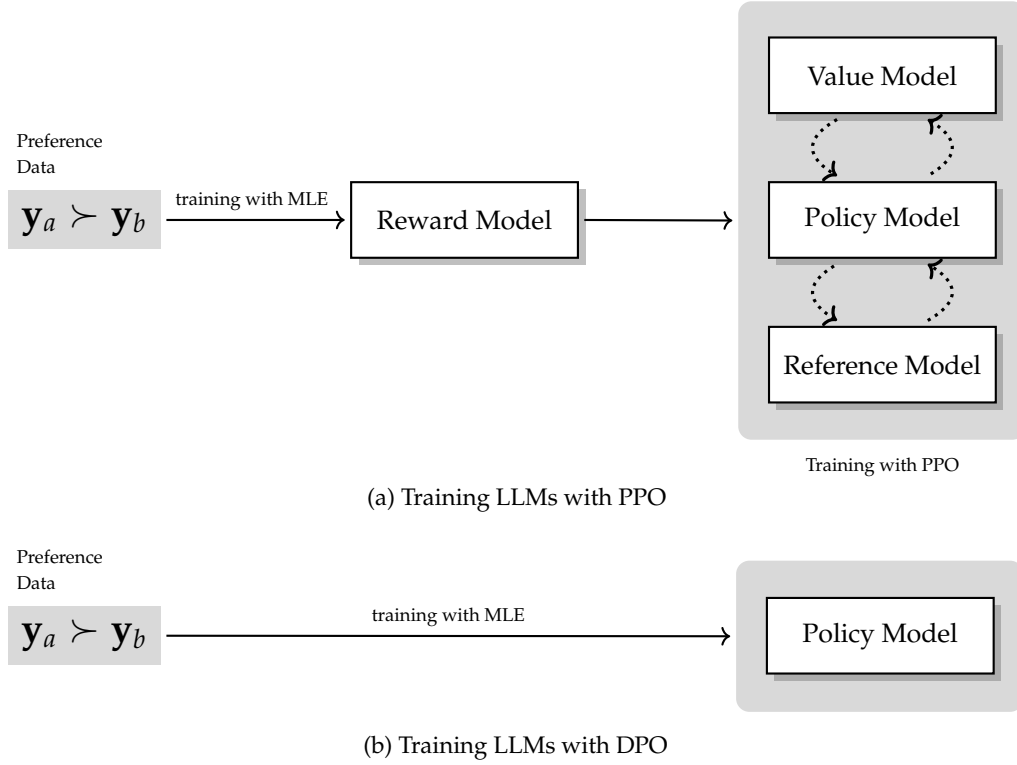


Figure 17: Standard PPO vs. DPO in training LLMs. In PPO, the human preference data is used to train a reward model, which is then employed to train the policy and the value function. In DPO, the use of human preference data is more direct, and the policy is trained on this data without the need for reward model training.

This loss function utilizes the implicit reward for DPO training, which replaces the need for an external reward model. The implicit reward is computed as the log-ratio of probabilities:

$$R(\mathbf{x}, \mathbf{y}) = \beta \frac{\Pr_{\theta}(\mathbf{y}|\mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})} \quad (51)$$

Here, we can consider that the DPO directly models human feedback through the relative likelihood of outputs under the policy and reference models. This method provides a more straightforward approach than traditional methods that use a separate reward model and then apply RL algorithms like PPO to adjust the probabilities of sampled outputs. In this way, we can further understand the optimization objective of the DPO by leveraging the hypothesis space as mentioned in Section ?? . In practice, as illustrated in Figure ?? , we can see the DPO as reranking in the hypothesis space through the Bradley-Terry model approach. Specifically, this method increases the probabilities of preferred outputs in preference data while decreasing those of dispreferred ones. Similar to TRPO, the DPO also incorporates a penalty derived from the reference model, which ensures that updates remain within a trusted behaviour space for the policy model.

However, there are two sides to every coin. While DPO simplifies the RL process, it also introduces limitations. Notably, since DPO eliminates the exploration phase during training, its potential peak performance is generally considered lower compared to RL-based methods like PPO, which incorporate exploration to discover more effective policies. In other words, the performance of DPO is inherently bounded by the labeled preferred outputs in the preference data. To address this limitation, current

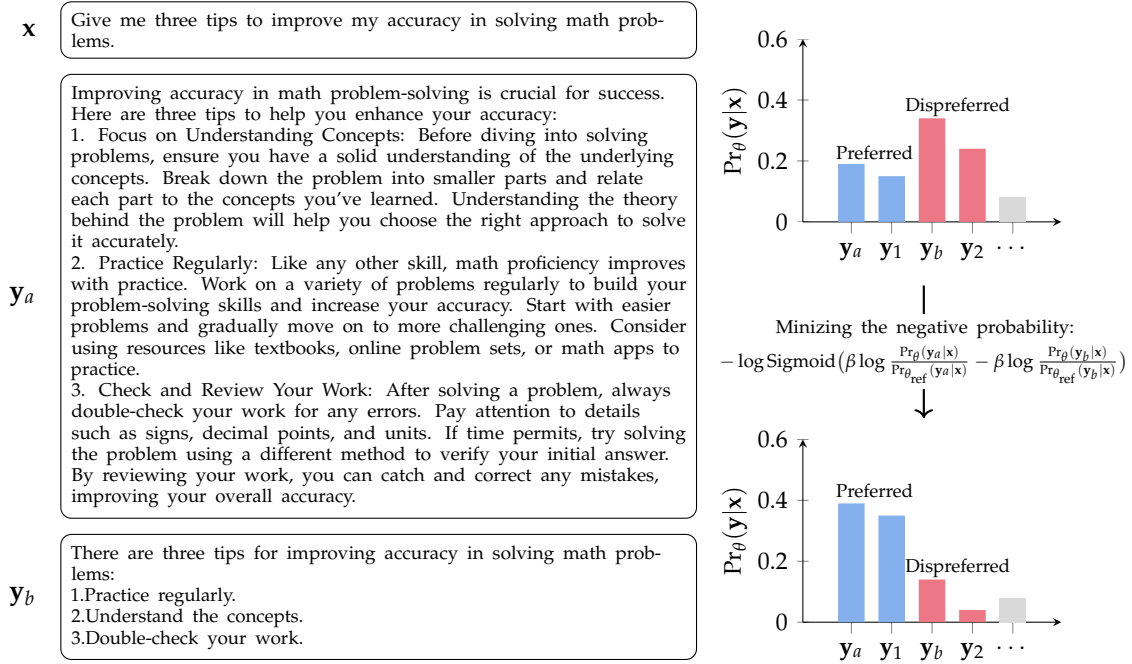


Figure 18: Illustration of the optimization objective of DPO from a hypothesis space perspective. The right-top chart shows the probability distributions over different hypotheses before optimization, where y_a is the preferred output and y_b is the dispreferred output. The right-bottom chart shows the probabilities after applying DPO, where the likelihood of the preferred output increases and that of dispreferred outputs decreases. Note that during this optimization process, outputs similar to the preferred output y_a (such as y_1) will obtain an increased probability, while those similar to the less preferred output y_b (such as y_2) will experience a decreased probability, due to the generalization of the model.

approaches focus on ensuring high-quality preferred outputs, either by employing advanced LLMs like GPT-4 or through human labeling (??).

Another limitation associated with DPO is over-optimization. Since DPO employs the Bradley-Terry model for modeling preferences, it can suffer from over-optimization, such as length exploitation, where a longer output might be mistakenly deemed as more aligned with human preferences (??). Many efforts have been made to address this issue and propose different variants of DPO, as detailed in Table ?? . Note that here we only provide an introduction to their optimization objectives. For more discussions on these variants, interested readers can refer to the related papers.

5 Reinforcement Learning for LLM Reasoning

So far, our discussion has mainly focused on various aspects of using and improving RL for training LLMs. The methods mentioned can be easily adapted to a wild of scenarios where the correctness of an output can be examined by checking whether the desired result is included. For example, in the task of calculating a mathematical expression, a reward model can provide positive feedback if the answer is correct and negative feedback if the answer is wrong. However, in many problems that require complex reasoning, simply examining the correctness of the final answer is insufficient for learning. Imagine a student who is only given the final answer to a challenging math problem. Knowing whether the final answer is right or wrong does not help the student figure out where they went wrong and how to calculate the correct answer. A better approach would be to guide the student with a step-by-step breakdown of the problem-solving process and encourage understanding of the underlying concepts and logic behind these steps. To address

Method	Objective
IPO (?)	$\left(\log \frac{\Pr_\theta(\mathbf{y}_a \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_a \mathbf{x})} - \log \frac{\Pr_\theta(\mathbf{y}_b \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_b \mathbf{x})} - \frac{1}{2\tau} \right)^2$
CPO (?)	$-\log \text{Sigmoid}(\beta \log \Pr_\theta(\mathbf{y}_a \mathbf{x}) - \beta \log \Pr_\theta(\mathbf{y}_b \mathbf{x})) - \lambda \log \Pr_\theta(\mathbf{y}_a \mathbf{x})$
KTO (?)	$-\lambda_a \text{Sigmoid} \left(\beta \log \frac{\Pr_\theta(\mathbf{y}_a \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_a \mathbf{x})} - z_{\theta_{\text{ref}}} \right) + \lambda_b \text{Sigmoid} \left(z_{\theta_{\text{ref}}} - \beta \log \frac{\Pr_\theta(\mathbf{y}_b \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_b \mathbf{x})} \right)$ where $z_{\theta_{\text{ref}}} = \mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim \mathcal{S}} [\beta \text{KL}(\Pr_\theta(\mathbf{y} \mathbf{x}) \Pr_{\theta_{\text{ref}}}(\mathbf{y} \mathbf{x}))]$
ORPO (?)	$-\log p_\theta(\mathbf{y}_a \mathbf{x}) - \lambda \log \text{Sigmoid} \left(\log \frac{p_\theta(\mathbf{y}_a \mathbf{x})}{1-p_\theta(\mathbf{y}_a \mathbf{x})} - \log \frac{p_\theta(\mathbf{y}_b \mathbf{x})}{1-p_\theta(\mathbf{y}_b \mathbf{x})} \right)$ where $p_\theta(\mathbf{y} \mathbf{x}) = e^{\frac{1}{ \mathbf{y} } \log \Pr_\theta(\mathbf{y} \mathbf{x})}$
R-DPO (?)	$-\log \text{Sigmoid} \left(\beta \log \frac{\Pr_\theta(\mathbf{y}_a \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_a \mathbf{x})} - \beta \log \frac{\Pr_\theta(\mathbf{y}_b \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_b \mathbf{x})} + (\alpha \mathbf{y}_a - \alpha \mathbf{y}_b) \right)$
PCDPO (?)	$-\log \text{Sigmoid}(\Delta^* - \Delta_{\Pr_\theta}) - \log \text{Sigmoid} \Delta_{\Pr_\theta}$, where $\Delta_{\Pr_\theta} = \beta \log \frac{\Pr_\theta(\mathbf{y}_a \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_a \mathbf{x})} - \beta \log \frac{\Pr_\theta(\mathbf{y}_b \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_b \mathbf{x})}$, $\Delta^* = \frac{\beta_1}{\text{Sim}(\mathbf{y}_a, \mathbf{y}_b \mathbf{x}) + \beta_2} + \beta_3$
SimPO (?)	$-\log \text{Sigmoid} \left(\frac{\beta}{ \mathbf{y}_a } \log \Pr_\theta(\mathbf{y}_a \mathbf{x}) - \frac{\beta}{ \mathbf{y}_b } \log \Pr_\theta(\mathbf{y}_b \mathbf{x}) - \gamma \right)$
ODPO (?)	$-\log \text{Sigmoid} \left(\beta \log \frac{\Pr_\theta(\mathbf{y}_a \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_a \mathbf{x})} - \beta \log \frac{\Pr_\theta(\mathbf{y}_b \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_b \mathbf{x})} - \delta(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \right)$
Cal-DPO (?)	$-\log \text{Sigmoid}(\Delta_{\Pr_\theta}) + \lambda \mathcal{L}_{\text{cal}}$ where $\Delta_{\Pr_\theta} = \beta \log \frac{\Pr_\theta(\mathbf{y}_a \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_a \mathbf{x})} - \beta \log \frac{\Pr_\theta(\mathbf{y}_b \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_b \mathbf{x})}$, $\mathcal{L}_{\text{cal}}$ calibrates implicit rewards.
TDPO (?)	$-\log \text{Sigmoid} \left(\sum_{t=1}^{T_a} \beta \log \frac{\Pr_\theta(\mathbf{y}_{a,t} \mathbf{x}, \mathbf{y}_{a,<t})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_{a,t} \mathbf{x}, \mathbf{y}_{a,<t})} - \sum_{t=1}^{T_b} \beta \log \frac{\Pr_\theta(\mathbf{y}_{b,t} \mathbf{x}, \mathbf{y}_{b,<t})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_{b,t} \mathbf{x}, \mathbf{y}_{b,<t})} \right) + \lambda \sum_t \text{KL}_t$ where KL_t denotes the token-level KL regularization term.
AOT (?)	$\mathcal{L}_{\text{OT}}(\{r_\theta(\mathbf{x}, \mathbf{y}_a)\}, \{r_\theta(\mathbf{x}, \mathbf{y}_b)\})$ where $r_\theta(\mathbf{x}, \mathbf{y}) = \beta \log \frac{\Pr_\theta(\mathbf{y} \mathbf{x})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y} \mathbf{x})}$, and $\mathcal{L}_{\text{OT}}$ aligns preference distributions via optimal transport.
D ² PO (?)	$-\log \text{Sigmoid} \left(\sum_{t=0}^T \gamma^t \beta \log \frac{\Pr_\theta(\mathbf{y}_a^t \mathbf{x}, \mathbf{y}_{a,<t})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_{a,t} \mathbf{x}, \mathbf{y}_{a,<t})} - \sum_{t=0}^T \gamma^t \beta \log \frac{\Pr_\theta(\mathbf{y}_b^t \mathbf{x}, \mathbf{y}_{b,<t})}{\Pr_{\theta_{\text{ref}}}(\mathbf{y}_{b,t} \mathbf{x}, \mathbf{y}_{b,<t})} \right)$

Table 1: DPO variants and their optimization objectives.

this, researchers also explore the potential of RL to teach LLMs not just to generate correct answers but to develop and present coherent, step-by-step reasoning that aligns with human cognitive processes. In this regard, RL has previously demonstrated its effectiveness in training neural networks for complex planning and reasoning within game environments, as evidenced by notable successes such as AlphaGo (?) and AlphaStar (?). Given these advancements and the inherently interactive nature of problem-solving, it is also natural to consider the application of RL to LLM reasoning.

In this section, we delve deeper into the application of RL to enhance the reasoning capabilities of the LLM, a topic that has recently garnered significant attention. We begin by giving a general introduction to test-time scaling that fully unleashes the reasoning potential of LLMs, including best-of- N sampling, step-by-step verification, and Monte Carlo Tree Search. We then discuss two critical issues: how to scale RL effectively, and how to iterate the RL process to enhance the reasoning capabilities of LLMs. Note that the following methods are primarily illustrated through the mathematical reasoning problem, they are applicable to a broad range of decision-making problems.

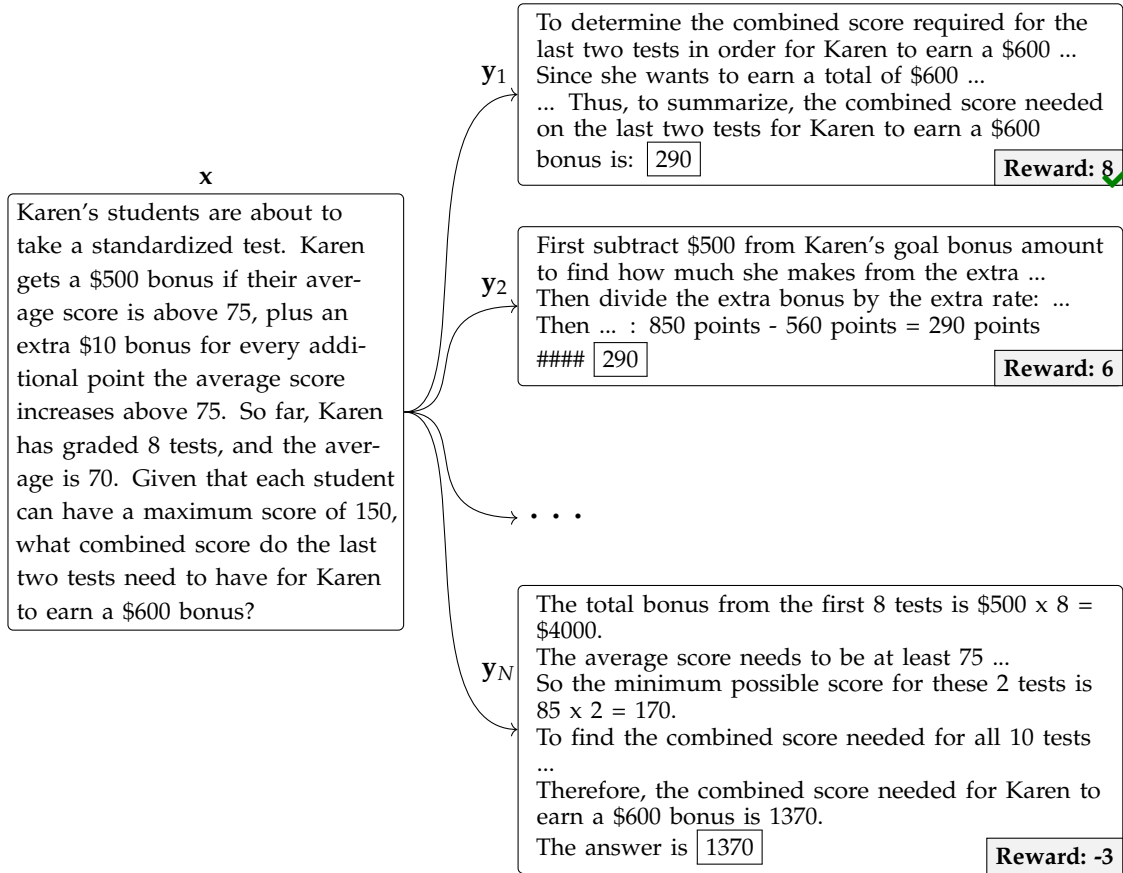


Figure 19: Best-of- n sampling. Given an input, multiple candidate outputs are sampled, and a reward model is used to select the best output. A majority vote can also determine the final output in scenarios without a reward model. For example, in this figure, most of the outputs suggest an answer of 290. Therefore, we can consider any one of the outputs resulting in 290 to be the final output.

5.1 Test-time Scaling

Initially, we review the fundamental objective of RL: to maximize the rewards obtained during output generation. This goal has been extensively achieved through various training-time optimization techniques, such as policy gradient or PPO algorithms. Beyond training, recent research has illuminated the benefits of scaling up test-time computation, a process also known as test-time scaling, which can significantly enhance the maximization of rewards, especially in the reasoning task. In a practical implementation, one simple approach to achieve test-time scaling is the use of prompting techniques. For example, appending the phrase “Let’s think step-by-step.” to the input can effectively stimulate the LLM to engage in a more detailed reasoning process during the generation. While this approach can improve reasoning accuracy, it often leads to suboptimal performance because the model is not inherently trained to understand the most beneficial reasoning processes. To address this issue, we can perform the test-time scaling with guidance from a reward model, as discussed in the following subsections.

5.1.1 Best-of- N Sampling

One approach to test-time scaling using a reward model involves sampling multiple reasoning paths given input and then using the reward model to select the best one from N alternative outputs generated by the LLM, called best-of- N sampling (BoN sampling). We can consider BoN sampling a reranking technique. In

fact, reranking methods have been a prevalent technique in NLP, particularly in machine translation, where they have been employed for a long time to enhance output quality by selecting the most appropriate translation from a set of candidates. Additionally, this method often functions as a simple model ensemble approach. In such cases, different outputs generated by various models can be reranked according to a specified metric.

As illustrated in Figure ??, in the BoN sampling, we first sample N different outputs $\{y_1, y_2, \dots, y_N\}$ for the input x :

$$\{y_1, \dots, y_N\} = \arg\text{TopN}_y [\Pr_\theta(y|x)] \quad (52)$$

where the $\arg\text{TopN}$ operation returns the top- N outputs that maximize the function $\Pr(y|x)$. These outputs can be sampled in various ways, depending on the search algorithm used by the model (e.g., nucleus sampling or beam search). Once the N -best output candidates are sampled, the reward model is used to evaluate and select the best one:

$$y_{\text{best}} = \max\{R_\phi(x, y_1), \dots, R_\phi(x, y_N)\} \quad (53)$$

This process not only identifies the output with the highest reward but also makes it a direct evaluation of the effectiveness of the reward model. For example, the agreement between the reward model and human judgments can be assessed by judging whether y_{best} matches the best output selected by humans. Given its speed and cost-efficiency compared to more complex evaluation methods such as those used in PPO, this approach is widely used for evaluating the performance of reward models (??). Nevertheless, in scenarios without a reward model, we can also employ the majority vote approach to determine the final output of the reasoning task. The basic steps involve first identifying the answer that most reasoning paths converge upon and then considering any of the outputs leading to this answer as the final output.

It is worth noting that the result of BoN sampling is also influenced by the diversity of the N -best list. This is a common issue with most reranking methods. Typically, we wish the N -best output candidates to be relatively high quality but sufficiently different from each other. In many text generation systems, the N -best outputs are very similar, often differing by just one or two words. The diversity issue is even more challenging in LLMs, as the N -best outputs sampled by an LLM can differ in their wordings. Yet, their semantic meanings are often quite similar. In practice, one can adjust the model hyperparameters and/or adopt different LLMs to generate more diverse output candidates for reranking. Nevertheless, as with many practical systems, we need to make a trade-off between selecting high-quality candidates and ensuring sufficient variation in the sampled outputs.

BoN sampling can also be used to train LLMs. A closely related method is rejection sampling. In this method, we first select the “best” outputs from the N -best lists via the reward model and then take these selected outputs to fine-tune the LLM. In this way, we can introduce human preferences into the training of LLMs via a much simpler approach than PPO. Many LLMs have adopted rejection sampling for fine-tuning (??).

5.1.2 Step-by-step Verification

While BoN sampling is effective for scaling test-time computation, it is inefficient because the model must generate the entire reasoning path before evaluating its quality. Addressing this inefficiency, one approach is step-by-step verification, which evaluates the quality of each step while generating a reasoning path⁹. This allows us to identify the potential of a path at intermediate steps early and further reduce computational resources by abandoning unpromising paths. Note that in this approach, traditional reward

⁹We typically evaluate each reasoning step from multiple dimensions. A primary consideration is the presence of errors, assessing whether the intermediate step contains any inaccuracies. Furthermore, we evaluate the advantage conferred by the step, which examines the potential of the reasoning step to facilitate superior outcomes in subsequent steps (?).

models, which are designed to evaluate the entire output, are inadequate. Instead, we need to develop a process reward model to evaluate each step in the reasoning path.

We can collect or generate reasoning paths corresponding to problems from existing datasets to train a process reward model. Human experts then annotate each step in these paths for correctness. These annotations can be used to train LLMs or as rewards in reward modeling directly. However, in practice, richer annotations are often introduced (?). In addition to the *correct* and *incorrect* labels, a step can also be labeled as *neutral* to indicate that while the step may be technically correct, it might still be problematic within the overall reasoning process. Additionally, an automatic process annotation framework can be utilized, which relies on generating multiple entire reasoning paths based on the current step and using the accuracy of these paths to serve as the quality annotation of the step (?).

Given a set of step-level annotated reasoning paths and corresponding inputs, we can train a reward model to provide a reward for each step in the reasoning process. The reward model can be treated as a classification model. So its architecture can be an LLM with a Softmax layer stacked on top, akin to the architecture depicted in Figure ?? . Here, consider an reasoning path including n_s steps, represented as $\mathbf{y} = \{\bar{\mathbf{y}}_1, \dots, \bar{\mathbf{y}}_{n_s}\}$. At each step k , the process reward model takes both the problem description, denoted by $\mathbf{x}$, and the reasoning steps generated so far, denoted by $\bar{\mathbf{y}}$, as inputs. It then outputs a probability distribution over the set of labels *correct*, *incorrect*, or *correct*, *incorrect*, *neutral*, to evaluate the reasoning at that point. This model can trained by a casual classification loss, e.g., Sigmoid Cross-entropy. Once trained, the reward model can be used to evaluate reasoning paths by assessing the correctness of each step. A simple method to use log-probabilities of classification to define the reward of each reasoning step, for example, the reward of the k -th reasoning step can be given by

$$R_\phi(\mathbf{x}, \bar{\mathbf{y}}_{\leq k}) = \Pr_\phi(\text{correct}|\mathbf{x}, \bar{\mathbf{y}}_{\leq k}) \quad (54)$$

where $\Pr_\phi(\text{correct}|\mathbf{x}, \bar{\mathbf{y}}_{\leq k})$ denotes the probability of the *correct* label generated by the reward model. The reward score $R_\phi(\mathbf{x}, \mathbf{y})$ can then be used to select the best step while generating a reasoning path. Additionally, as discussed in Section ??, there is the option to train a generative process reward model that further improves this performance on evaluating the reasoning step, also called generative verifier in the literature (?). Note that in practice, the process reward model serves not only to provide rewards for test-time scaling but also to train the model using RL, e.g., the rewards from this model can be employed as shaping rewards.

As illustrated in Figure ??, step-by-step verification can be conducted using a simple greedy search method. Specifically, multiple candidate reasoning steps are sampled at each step, and the process reward model is employed to select the best one. This selected step then serves as the foundation for generating the subsequent step, continuing this process until the entire reasoning path is obtained. In this way, any search method can be applied to improve test-time scaling. For example, we can expand the search space with beam search, retaining multiple promising reasoning steps at each step. Furthermore, the reasoning steps can be refined through the self-refinement technique at each step using verification feedback, such as rewards, to enhance accuracy (?).

5.1.3 Monte Carlo Tree Search

In this subsection, we discuss the use of Monte Carlo Tree Search (MCTS), a popular search method in step-by-step verification. In practice, MCTS is not a new method but is well-established across various domains. Notably, it typically serves as an effective alternative to traditional RL methods, particularly in environments characterized by large or intricate state spaces, where conventional RL algorithms may falter. For example, within an RL framework, MCTS can aid in planning by simulating diverse actions to maximize rewards, as demonstrated by its success in complex game environments (?). In LLM reasoning, MCTS leverages randomness and structured tree search to probe potential reasoning paths, thereby expanding the search space in an efficient way. More specifically, as illustrated in Figure ??, MCTS repeatedly cycles through the following four stages to explore potential reasoning paths:

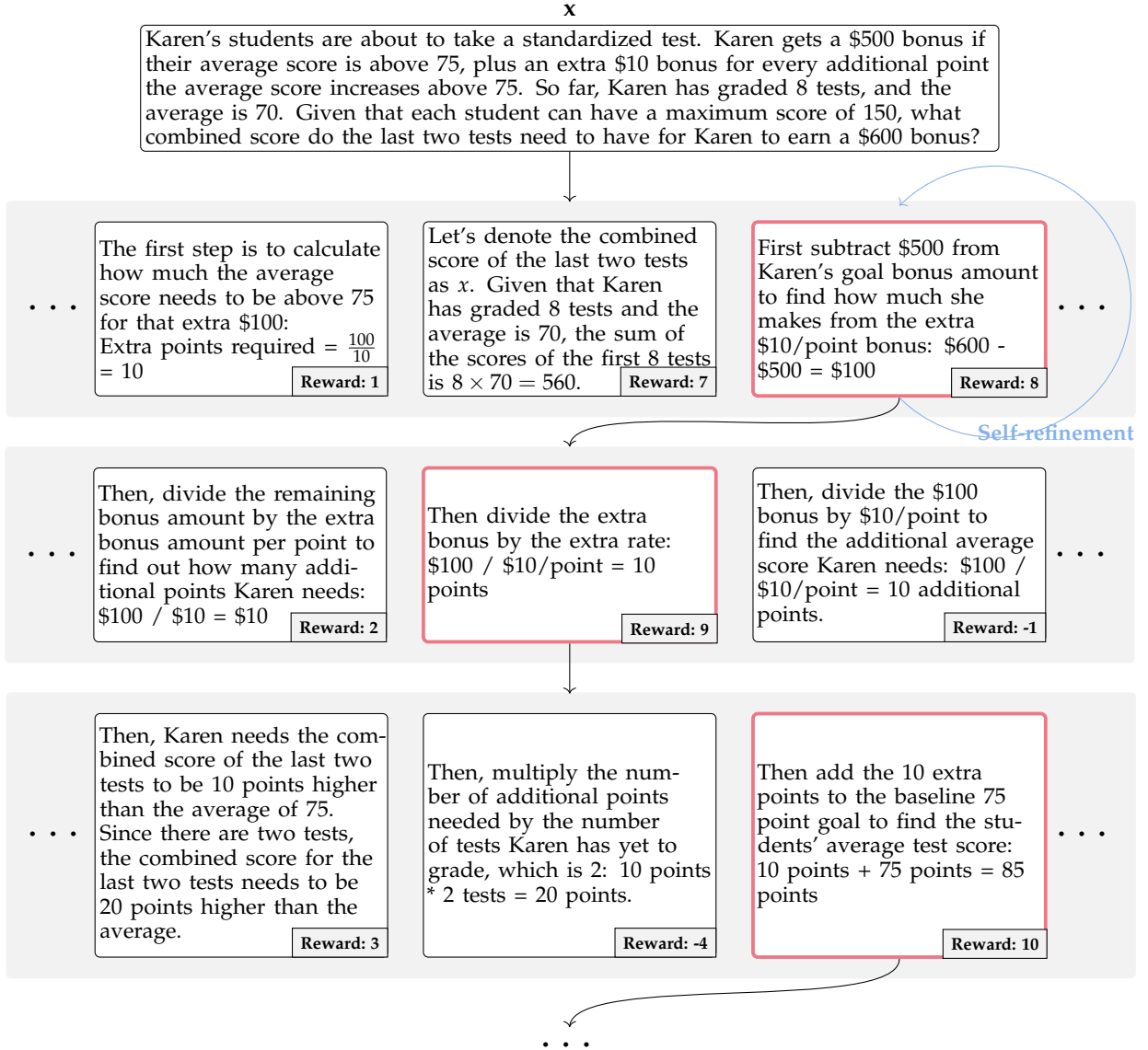


Figure 20: Step-by-step verification with greedy search. This process can be readily extended to beam search by retaining the top steps at each step from all candidate paths for further scaling up test-time computation. As the blue line indicates, the reasoning steps can be refined using verification feedback, such as rewards, to enhance accuracy.

- **Selection.** This process begins at the root node (i.e., an input), where the algorithm selects promising child nodes (i.e., reasoning steps) based on specific selection strategies, such as the Upper Confidence Bound (UCT). More specifically, at each step k , among the sampled candidate reasoning steps, we aim to select the one that maximizes the UCT objective:

$$\bar{\mathbf{y}}_k^* = \arg \max_{\bar{\mathbf{y}}_k} (\bar{R}(\bar{\mathbf{y}}_k) + c \sqrt{\frac{\ln N_p}{N_{\bar{\mathbf{y}}_k}}}) \quad (55)$$

where N_p denotes the total number of times the parent node has been visited, $N_{\bar{\mathbf{y}}_k}$ is the number of visits to the child node $\bar{\mathbf{y}}_k$, and c is a constant that balances the trade-off between exploitation of known good paths and exploration of new paths. $\bar{R}(\cdot)$ denotes the reward of a reasoning step.

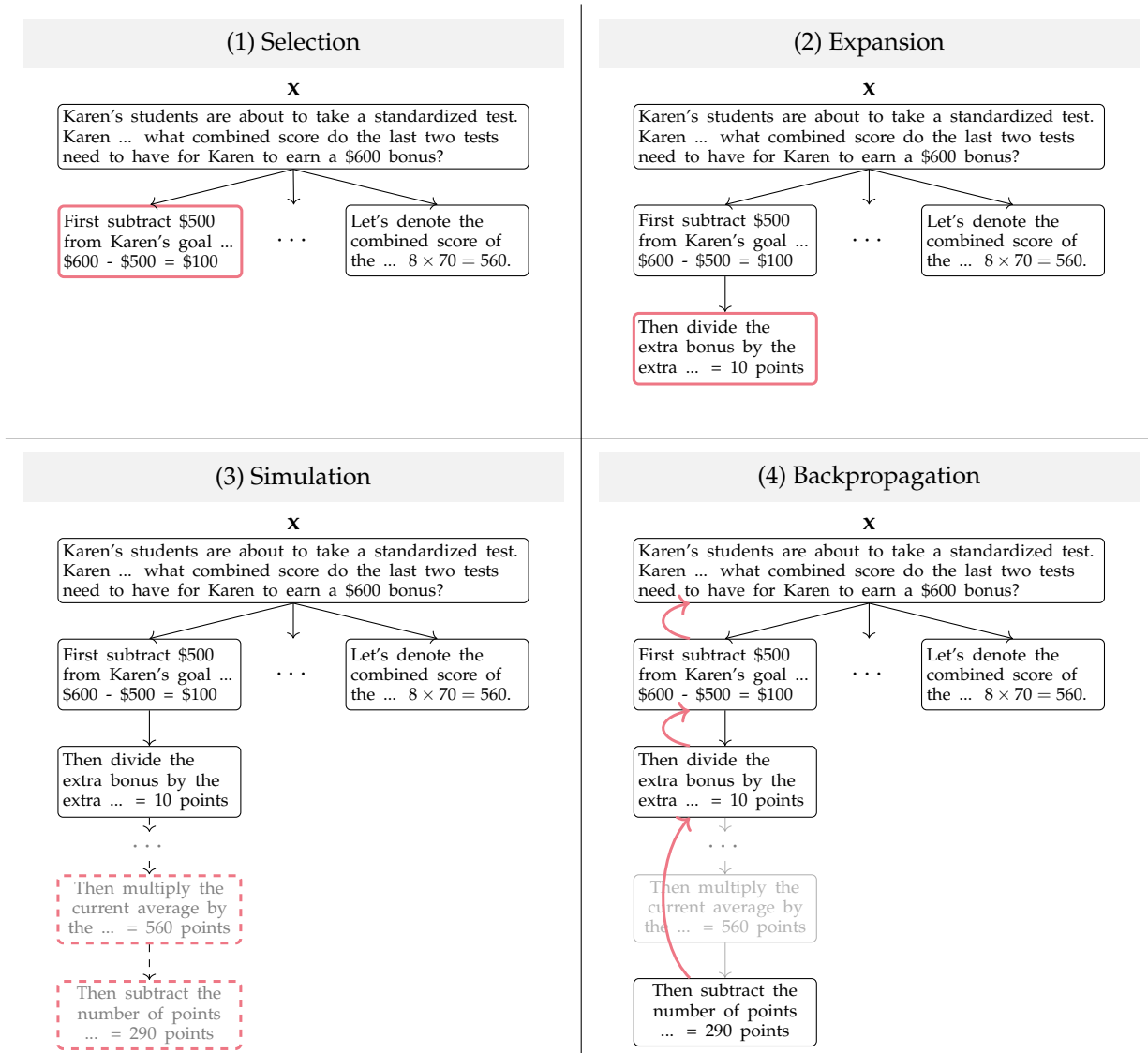


Figure 21: Illustration of step-by-step verification with MCTS. The process consists of four phases: (1) Selection, where the algorithm navigates the decision tree to select the most promising node based on strategies like UCT; (2) Expansion, where new potential nodes are added to the tree for further exploration; (3) Simulation, where each new node is evaluated by simulating possible outcomes (i.e., reasoning paths); and (4) Backpropagation, where the outcomes from the simulations are used to refine node values, such as UCT values, thus enhancing the decision-making for subsequent explorations.

This reward can be static, set by a process reward model, or dynamic, continuously updated based on the delayed rewards obtained from simulations.

- **Expansion.** Once a leaf node (i.e., a newly generated reasoning step based on previously selected steps) is encountered and it does not represent a terminal state of the reasoning process, the algorithm expands this node by adding it as a new child node.
- **Simulation.** The process performs simulations (or rollouts) for each new node added during the expansion stage. These simulated paths help evaluate the effectiveness of the reasoning steps

initiated from the node. For example, the final answer obtained from the simulation can be compared with the correct answer to derive a delayed reward.

- **Backpropagation.** The process updates the UCT values of prior nodes using the results of these simulations. Key updates include the visitation counts, N_p and $N_{\bar{y}_k}$ in Eq. (??), reflecting how frequently each node has been explored. Additionally, this stage allows for the incorporation of delayed rewards to adjust the $\bar{R}(\bar{y}_k)$ values.

5.2 Large-scale Reinforcement Learning

While test-time scaling is instrumental in exploring more effective reasoning paths, its potential is restricted if the model has not learned to generate high-quality reasoning paths. In other words, in practice, test-time scaling struggles to achieve desired outcomes without foundational reasoning ability, no matter how extensive it is (??). Consequently, there has been a growing research interest in training LLMs specifically to develop reasoning abilities that mimic human-like processes. A straightforward approach is to use annotated reasoning data to train the LLM through SFT. However, a significant challenge in this approach is the high cost of annotations, especially since annotating detailed step-by-step reasoning paths is significantly more cost-intensive than annotating SFT data in other tasks like machine translation and summarization.

Another more advanced approach is to utilize large-scale RL, transitioning from human-annotated to self-reinforced learning processes. Unlike the RL used as a fine-tuning method discussed in Section ??, which typically involves training for a few dozen or hundreds of steps at a small scale, this approach employs it on a larger scale with rewards to break free from the limitation of supervised reasoning data. This approach has enabled the development of robust reasoning models, such as OpenAI-o1 (?), DeepSeek-R1 (?), and Kimi-1.5 (?). Notably, DeepSeek-R1-Zero was able to develop a robust reasoning model by applying large-scale RL to a pre-trained LLM without the need for any annotated reasoning data. However, despite its successes, large-scale RL is not easy and introduces unique challenges that are not present at smaller scales, as follows.

One is that large-scale RL requires a highly generalizable reward model. As the scale of training increases, the model is trained on broader data, necessitating a reward model capable of effectively generalizing across this varied data. On the other hand, the extensive scope of training introduces significant variability in the model, leading to considerable changes in sampling behaviours. Consequently, it is crucial to ensure that the reward model possesses robust generalization capabilities to prevent overfitting and maintain the effectiveness of the learning process. There are several methods to achieve this. For example, ? incorporated rule-based rewards, as discussed in Section ??, such as format checking and answer verification in reasoning scenarios, which can provide a stable reward throughout the learning process. This suggests that in certain RL scenarios, prioritizing rule-based rewards may be beneficial if they can effectively describe human preferences. ? introduced a self-rewarding framework that dynamically updates the reward model based on the currently optimized policy model so that this reward model can effectively evaluate the behaviours from the current policy model.

Another is that large-scale RL might be constrained by the reference model. To prevent the policy model from deviating too far from its initial state, an SFT LLM is typically employed as the reference model, adding a penalty that restricts updates to ensure the policy model operates within a desirable behaviour region. However, reliance on a static reference model can limit the adaptability and innovation potential of the policy model in large-scale RL. A common strategy to mitigate this issue is to periodically update the reference model to better align with the improving capabilities and knowledge of the policy model, thus striking a balance between keeping stability and fostering adaptiveness during the training process (?).

5.3 Iterative Reinforcement Learning

Training LLMs with RL usually follows a two-phase approach: training a pre-trained LLM with SFT and further training with an RL algorithm applied to the SFT LLM. However, using large-scale RL in such a two-phase approach to train LLMs in reasoning may lead to significant knowledge forgetting as the model continuously adjusts to fit the reward model. For example, as mentioned in DeepSeek-R1 (?), directly applying large-scale RL can achieve the desired reasoning outcomes, but often at the cost of reduced readability in the reasoning process. To address these challenges, we can utilize an iterative RL approach to continuously enhance the various capabilities of the LLM. Here we consider DeepSeek-R1 as an example to illustrate how to perform an iterative RL. The idea is to split the RL process into multiple phases, each designed to enhance different capabilities using varied rewards, to develop a robust reasoning model that is able to generate clearer and more comprehensible reasoning paths. Figure ?? provides a schematic illustration of the iterative RL process. Here we give a brief outline of each phase involved.

- The iterative RL aims to train an LLM that generates a human-like reasoning process. Initially, it involves collecting high-quality data from various reasoning tasks, such as mathematical problem-solving and code generation. This data is then used to fine-tune a pre-trained LLM as a cold start. This phase is designed to equip the LLM with basic reasoning capabilities, preventing linguistically confused or nonsensical reasoning paths from being sampled during the following RL phase.
- After the cold start phase, large-scale RL is employed on the reasoning task. During this phase, rule-based rewards (i.e., format checking and answer verification) are utilized to optimize the reasoning process continuously. By doing so, the model learns longer and more reasonable reasoning paths, progressively enhancing its capacity to tackle complex reasoning tasks.
- While the model has learned how to generate reasoning paths through large-scale RL, it often produces outputs with poor readability. This can mainly be attributed to the large-scale RL phase, which does not focus on optimizing the readability of the reasoning paths but instead solely on format correctness and answer accuracy. To address this issue, the third phase involves using rejection sampling to further enhance the readability of outputs. More specifically, multiple outputs are sampled from the model trained in the previous phase across various tasks such as mathematical reasoning, code generation, and question answering. A reward model then selects the best outputs, as discussed in Section ?. These selected outputs are used to train the model using SFT, which aims to refine its performance by aligning it more closely with human-like reasoning.
- In the final phase, multi-task RL is implemented to ensure broader generalization and maintain high performance across tasks beyond reasoning. The process involves training the model using RL on various tasks against a general reward model.

An interesting issue arises with this design of iterative RL: why is RL aimed at enhancing reasoning capabilities placed at the initial phase rather than at the end? This is partly because the rewards for the reasoning task, which are inherently more stable as described in Section ?, are well-suited for large-scale RL compared to traditional reward models. Furthermore, the reasoning capabilities could be applicable across many tasks, allowing for large-scale learning to boost performance in subsequent multi-task learning. We consider that this foundational enhancement in reasoning capabilities may contribute to the observed robust performance across various tasks, even without the use of labeled data in later phases. In this case, large-scale RL could be viewed as an approach to continued pre-training.

Since the optimization objective of each phase is different, the design of iterative RL can also be analyzed and understood from the perspective of multi-objective optimization (?). In the context of multi-objective optimization, iterative RL for LLMs can be likened to interactive methods where the solution process is iterative, and preferences are actively defined and refined by the decision-maker during the search for the most preferred solutions (?). More specifically, in this scenario, RL can be seen as a decision-maker, iteratively refining and enhancing the different capabilities of the LLM across different phases.

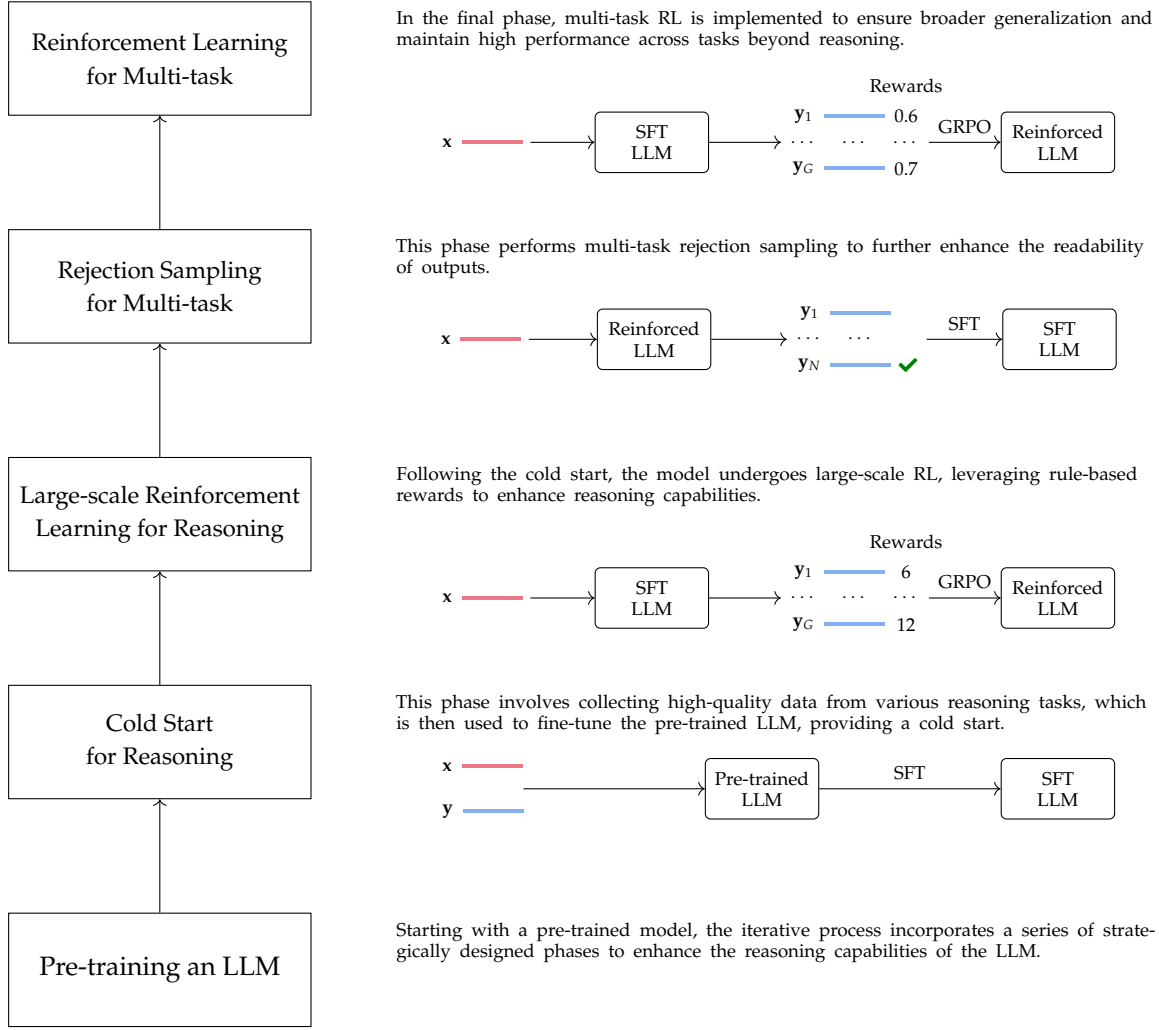


Figure 22: Illustration of iterative RL in DeepSeek-R1 (?). This method maintains multi-phase RL to develop a robust reasoning LLM with a GRPO algorithm. Initially, a pre-trained LLM is fine-tuned using a small set of high-quality labeled reasoning data with SFT as a cold start. Then, large-scale RL is applied specifically to enhance reasoning capabilities. After that, the model undergoes reject sampling across multiple tasks to refine output quality. In the final phase, the model is further trained using RL to enhance generalization across various tasks.

6 Agentic Reinforcement Learning

Using RL to train agents, often referred to as *agentic RL*, is not a new concept. In classical machine learning, agentic RL typically involves training a domain-specific decision model from scratch. For example, in tasks such as playing complex games like Go (??) or controlling robotic locomotion (?), the goal is to explore a structured environment and learn an optimal policy starting from random initialization.

With the rise of LLMs as core components of autonomous systems, RL has been increasingly used to adapt these models for sequential decision-making in dynamic and open-ended environments. In this setting, agentic RL addresses a key limitation of LLMs: although they encode extensive world knowledge, their outputs are not inherently aligned with long-horizon task requirements. For example, a supervised fine-tuned LLM may correctly generate a script for calling an external API, but it cannot reliably self-correct

when the environment returns unexpected errors. This creates new challenges for enabling LLM-based agents to plan autonomously, adapt to environmental feedback, and safely improve their strategies through iterative interaction.

In this section, we focus on the emerging paradigm of agentic RL for LLM-based agents. We begin by discussing how RL builds and enhances core agentic capabilities, specifically focusing on long-horizon planning and tool-integrated reasoning. Then we consider training-free methods that apply RL principles to optimize external modules such as memory updates, skill optimization, and trajectory refinement without altering the internal model weights. It is worth noting that while these approaches depart from traditional parametric updates, they fundamentally adopt the RL modeling paradigm to formulate and optimize sequential decision-making. Finally, we introduce the crucial role of designing better environments, which serve as the foundational testbeds for training agents.

6.1 Building Agent Capabilities

LLM-based agents extend LLMs from passive response generators into interactive decision-making systems. Rather than producing a single answer, an agent must repeatedly observe its environment, decide what to do next, and execute an action. This interaction loop allows the agent to solve tasks that require multiple steps, external resources, and adaptation during execution. The underlying LLM already provides several important capabilities, including instruction understanding, reasoning, code generation, and broad world knowledge. These capabilities allow the agent to interpret complex goals and propose plausible intermediate steps. However, they do not automatically guarantee reliable behavior in an interactive environment. An agent must also determine how to decompose a task, choose among alternative actions, coordinate external tools, and recover when an earlier decision leads to an unexpected result.

These agentic capabilities are difficult to learn from static input-output pairs alone. The quality of an individual decision often depends on its effect on later interactions and the final task outcome. A locally plausible action may lead to failure several steps later, while an initially unsuccessful attempt may provide useful information for a better strategy. RL is well suited to this setting because it optimizes complete interaction trajectories through environmental feedback and delayed rewards (??). It enables the agent to explore different behaviors, compare their consequences, and gradually improve its decision-making policy.

In this subsection, we focus on two core capabilities in agentic RL: *planning* and *tool use*. Planning allows the agent to organize a complex goal into a sequence of executable steps. Tool use allows it to access external information, perform reliable computation, and take actions beyond the capabilities stored in the model parameters. Together, these capabilities form the basis for effective interaction in given environments.

6.1.1 Planning

An agent’s autonomy refers to its ability to independently determine the steps required to complete a given task, even when these steps are not explicitly specified in the input. This capability requires the agent to infer the necessary procedure from the task description, decompose the goal into executable subgoals, and decide when and how to invoke external tools. Such autonomy is often reflected in the generation of a structured plan that guides the execution trajectory. For example, given a specific task description, the agent may produce an actionable plan with tool-use decisions as follows:

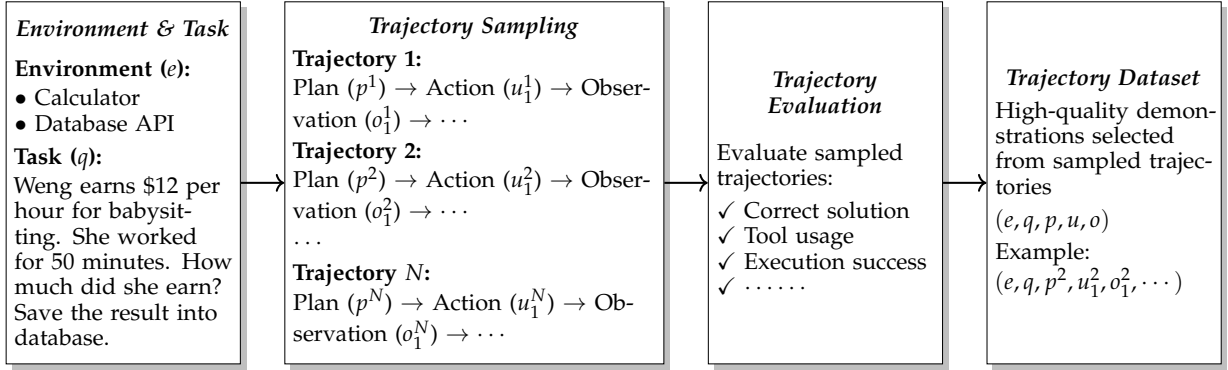


Figure 23: Overview of SFT data construction for improving agent planning.

Input	<i>Environment:</i> Calculator, Database API <i>Task:</i> Weng earns \$12 per hour for babysitting. Yesterday, she babysat for 50 minutes. How much did she earn? Please save the result into the database.
Output	1. Extract the relevant numerical values and identify the required computation. 2. Call the calculator tool to evaluate the expression $12 \times (50/60)$. 3. Format the computed result into a valid JSON schema. 4. Invoke the <code>update_record</code> API to save the final earnings into the database. Next, I will execute the plan step by step.

Unlike single-turn response generation, agent planning requires a sequence of interdependent decisions, where early choices can affect subsequent tool use, environmental feedback, and final task success. Improving planning capability therefore requires not only teaching the model to generate reasonable intermediate steps, but also enabling it to adjust its behavior based on execution outcomes. To this end, researchers have explored various approaches to achieving this goal. Among them, one promising approach is RL, as it naturally formulates agent planning as sequential decision-making driven by environmental feedback and trajectory-level rewards. In practice, SFT provides a cold start by allowing the model to imitate high-quality planning trajectories, while RL further refines the agent through interaction-based feedback and trajectory-level optimization.

6.1.1.1 Supervised Planning Data

Before applying RL, SFT is commonly adopted to provide a cold start for LLM-based agents by teaching them basic planning and interaction behaviors (??). As illustrated in Figure ??, constructing SFT data for agent planning typically involves three stages: (1) preparing diverse environments and planning tasks; (2) synthesizing expert-level trajectories, consisting of action-observation sequences, through agent-environment interactions; and (3) selecting high-quality trajectories based on predefined evaluation criteria. Specifically, expert trajectories can be generated by leveraging state-of-the-art LLMs as expert agents and retaining reliable demonstrations according to reward signals or task success criteria. Through this process, LLMs can learn planning behaviors from demonstrations and improve their ability to generate structured execution plans.

$$\mathbf{x} = [e, q] \quad (56)$$

$$\mathbf{y} = [p, u_1, o_1, \dots, u_T, o_T] \quad (57)$$

where e denotes the environment information, including available tools and external resources, and q represents the task description provided by users. p denotes the generated plan that decomposes the task into intermediate steps. u_t and o_t represent the agent’s behavior and the corresponding observation from the environment at the t -th interaction step, respectively. Specifically, u_t denotes the executable behavior performed by the agent, such as tool invocation, while o_t denotes the feedback returned by the environment after executing u_t . Therefore, $\{u_1, o_1, \dots, u_T, o_T\}$ forms a sequential interaction trajectory between the agent and the environment. It is worth noting that although u_t and a_t in Section ?? both represent “action”, they are defined at different levels of abstraction. Specifically, u_t represents a high-level interaction step in agent planning, such as calling a specific tool, whereas a_t represents the low-level token-level action in the standard LLM-based RL formulation, corresponding to the generation of an individual token by the language model.

Similar to LLM (??), the scale and quality of trajectory data significantly influence the effectiveness of agent training. High-quality trajectories provide explicit supervision signals for agents to learn how to decompose complex tasks, select appropriate tools, and interact with dynamic environments. Therefore, recent studies have explored constructing trajectory-based instruction tuning datasets to enhance the planning capabilities of LLM-based agents.

A straightforward approach is to utilize generated trajectories for instruction tuning. However, trajectories collected from LLM-based agents may contain unsuccessful plans, ineffective tool usage, or execution failures, which can introduce noisy supervision. Therefore, existing methods typically employ filtering strategies based on task success criteria or heuristic rules to select high-quality demonstrations. For example, AgentTuning constructs AgentInstruct by collecting interaction trajectories and filtering unsuccessful samples, while FireAct investigates the impact of diverse trajectory data from multiple tasks and reasoning paradigms for improving agent fine-tuning (??).

Beyond selecting high-quality trajectories, another challenge lies in effectively representing the complex interaction processes within trajectories. Unlike conventional instruction tuning, agent trajectories contain not only final responses but also intermediate planning processes and multi-step interactions with environments. Therefore, how to organize and annotate different components of trajectories becomes crucial for enabling agents to acquire planning and execution capabilities. To address this challenge, AgentLUMOS introduces a modular training framework that explicitly separates high-level planning and low-level execution (?). Specifically, its planning module learns to generate high-level subgoals, while its grounding module learns to translate these subgoals into executable actions through external tools. Such a modular design enables agents to acquire different capabilities from structured trajectory annotations.

Furthermore, even with effective trajectory selection and representation, scaling the generation of diverse and high-quality trajectory data remains challenging. Existing approaches often rely on manually designed environments and planning tasks, which limits the diversity and coverage of collected trajectories. Designing new environments requires substantial human expertise, while constructing tasks with appropriate difficulty levels remains difficult. To address this limitation, AGENTGEN explores automatically generating diverse environments and planning tasks with LLMs, enabling large-scale synthesis of trajectory data with varying task complexity for agent training (?).

6.1.1.2 Learning to Plan with Reinforcement Learning

Although SFT can provide a useful cold start for agent planning, it mainly teaches the model to imitate offline demonstrations. This limits its ability to improve beyond the quality and coverage of the collected trajectories. In contrast, RL further optimizes planning through direct interaction with the environment, where the agent receives feedback based on the outcome of its generated plans and executed behaviors.

Under the agent planning formulation, the LLM-based agent can be viewed as a high-level policy that generates a plan and then produces a sequence of executable behaviors (?). Given the environment information e and task description q , the agent first generates a plan p and then interacts with the environment

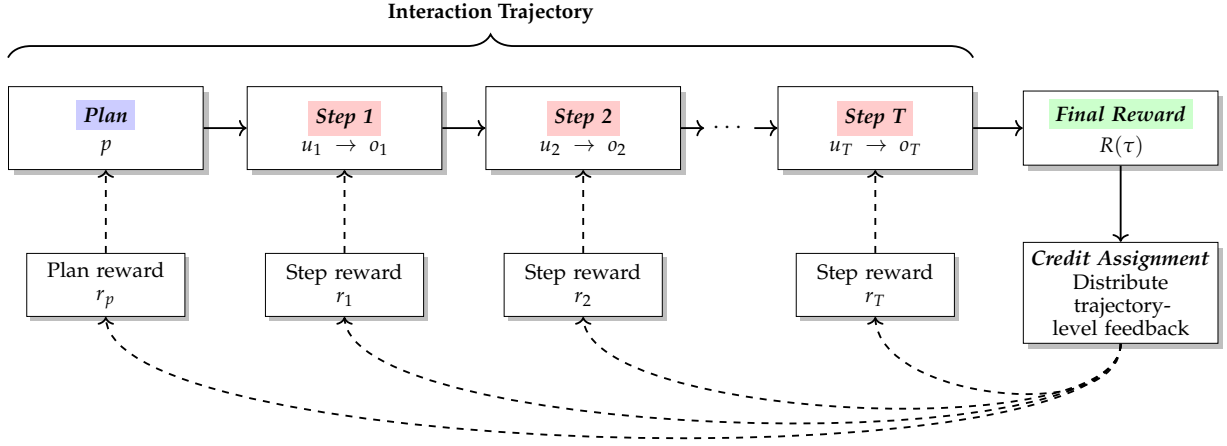


Figure 24: Illustration of credit assignment in agentic RL.

through a sequence of behaviors and observations:

$$\tau = [p, u_1, o_1, \dots, u_T, o_T] \quad (58)$$

The goal of RL is to optimize the agent policy so that the generated trajectory receives higher feedback from the environment. This objective can be written as

$$\theta^* = \arg \max_{\theta} \mathbb{E}_{\tau \sim \pi_{\theta}(\cdot|e,q)} [R(\tau)] \quad (59)$$

where π_{θ} denotes the agent policy parameterized by the LLM, and $R(\tau)$ denotes the trajectory-level reward that evaluates the overall quality of the planning process. In practice, this reward can be defined according to task success. The objective can then be optimized using standard RL algorithms introduced in Section ??, such as PPO or GRPO.

Compared with SFT, RL provides two important advantages for agent planning. First, it enables the agent to learn from execution outcomes rather than only from static demonstrations. If a plan leads to failed tool execution or poor environmental feedback, the agent can be penalized and encouraged to explore alternative behaviors. Second, RL supports trajectory-level credit assignment, allowing the model to adjust earlier planning decisions according to later outcomes. This is crucial for agent planning, where early subgoal decomposition or tool selection can substantially affect the final task success.

6.1.1.3 Credit Assignment

Credit assignment is not unique to LLM-based agents. It is a long-standing challenge in traditional RL. When rewards are delayed, the agent must decide which past actions should be reinforced or suppressed. Classical methods, such as temporal-difference learning and eligibility traces, address this issue by propagating reward information backward along the trajectory. In agentic RL, credit assignment becomes more challenging. This is because that each interaction step may involve high-level planning, tool invocation, and environmental feedback, rather than a single low-level action. Therefore, credit assignment often needs to operate at both the trajectory level and the step level.

This challenge is especially important for agent planning. In many agent tasks, the environment only provides a sparse reward after the entire trajectory is completed. However, the final success or failure may come from an earlier planning decision, an incorrect tool invocation, or an ineffective response to an intermediate observation. If we directly assign the same trajectory-level reward to all interaction steps, the supervision can become noisy. The agent may fail to identify which decisions should be reinforced and which should be suppressed.

Formally, given a trajectory $\tau = [p, u_1, o_1, \dots, u_T, o_T]$, a simple RL objective uses a trajectory-level reward $R(\tau)$ to optimize the whole planning process. However, such a reward only provides coarse feedback on the overall outcome. Therefore, credit assignment aims to decompose the trajectory-level reward into step-level signals:

$$R(\tau) \rightarrow \{r_p, r_1, \dots, r_T\} \quad (60)$$

where r_p evaluates the quality of the generated plan, and r_t evaluates the contribution of the t -th interaction step (u_t, o_t) to the final task outcome.

The mechanism of credit assignment is illustrated in Figure ?? . This decomposition enables the agent to distinguish whether failure comes from an unreasonable plan, an invalid tool call, or a poor adaptation to environmental feedback (???). For example, consider the task of computing the babysitting payment and saving the result into a database. Suppose the agent generates a reasonable plan and correctly calls the calculator, but fails to invoke the database API with the required JSON format. If we only use a final trajectory-level reward, the whole trajectory may receive a low reward, e.g., $R(\tau) = 0.3$, even though the early planning and calculation steps are correct.

For illustration, we can define rule-based step-level rewards using predefined verification criteria, such as whether the plan includes all necessary steps, whether the calculation is correct, whether the JSON schema is valid, and whether the database update succeeds. Then, we can use credit assignment to decompose the final trajectory-level feedback into step-level rewards:

$$R(\tau) = 0.3 \quad \Rightarrow \quad \{r_p = 0.4, r_1 = 1.0, r_2 = 1.0, r_3 = 0.7, r_4 = 0.0\} \quad (61)$$

where r_p indicates that the overall plan is reasonable, r_1 and r_2 indicate that the agent correctly extracts the numerical values and performs the calculation, r_3 indicates that the JSON formatting is partially correct, and r_4 indicates that the database update fails. In this way, the agent receives more precise feedback: it should preserve the correct planning.

6.1.2 Tool Use

Tool use is another fundamental capability of LLM-based agents. Tools extend the agent beyond its internal parameters. They allow the agent to access external knowledge, perform accurate computation, and execute actions in external systems. Early studies show that LLMs can use tools through prompt-based reasoning-action patterns. ReAct is a representative example, where the model alternates between reasoning steps and tool-use actions (?). However, prompt-based tool use is often unstable. It also depends heavily on the capability of the model. Thus, SFT methods train LLMs on tool-use trajectories, so that the model can learn when and how to invoke tools from demonstrations (?). Still, SFT mainly teaches imitation. It cannot directly optimize whether the agent should use a tool, which tool it should select, or how to balance tool benefit with tool cost. RL provides a natural framework for efficient tool use. We can define rewards based on task success, tool-use correctness, tool-use completeness, and tool cost (??).

6.1.2.1 Prompt-based Tool Use

Prompt-based tool is a simple approach to enable LLM-based agents to interact with external tools without updating model parameters. However, the model may not naturally know when to invoke a tool or how to organize tool-use behaviors. Thus, we usually need to provide tool descriptions and a few demonstrations in the prompt. These demonstrations show the model how to alternate between reasoning, tool invocation, and observation. At each step, the model first generates a reasoning trace, then produces a tool-use action. After the tool returns an observation, the model uses this feedback to update its next decision. For example,

Tools	<code>calculator(expression)</code> : return the numerical result of an expression. <code>update_record(json)</code> : save a JSON record into the database.
Demonstration	Task: Tom earns \$15 per hour. He worked for 2 hours. How much did he earn? Please save the result into the database. Thought: I need to compute the total payment. Action: <code>calculator(15 × 2)</code> Observation: 30 Thought: The payment is 30 dollars. I need to save it into the database. Action: <code>update_record({"earning": 30})</code> Observation: success Answer: Tom earned \$30, and the result has been saved into the database.
Current Task	Weng earns \$12 per hour for babysitting. Yesterday, she babysat for 50 minutes. How much did she earn? Please save the result into the database.
Output	Thought: I need to compute the payment based on the hourly wage and working time. Action: <code>calculator(12 × 50/60)</code> Observation: 10 Thought: The payment is 10 dollars. I need to save it into the database. Action: <code>update_record({"earning": 10})</code> Observation: success Answer: Weng earned \$10, and the result has been saved into the database.

This demonstration teaches the model a tool-use pattern. The model learns to reason about the task, invoke an external tool when necessary, read the returned observation¹⁰, and continue the execution process.

6.1.2.2 Supervised Tool-use Data

To make tool use more reliable, we can further train LLMs on tool-use trajectories with supervised fine-tuning. In this setting, each training example contains a user task, available tool descriptions, and a demonstrated tool-use process. The model learns when to invoke a tool, which tool to select, how to construct valid arguments, and how to use the returned observation. For example, we can construct a training example as follows:

x = Task: Weng earns \$12 ... Tools: `calculator(expression)`, ... , `update_record(json)`
 y = Thought: compute the payment; Action: `calculator(12 × 50 / 60)`; ...; Answer: Weng earned \$10.

With such data, SFT teaches the model to imitate the demonstrated tool-use pattern. The model can learn the format of tool invocation and the role of observations in later generation.

A common way to construct such data is to manually annotate tool-use trajectories. However, this requires annotators to know which tool should be used, how to write valid arguments, and how to judge whether the returned observation is useful. This process is costly and difficult to scale to many tools and tasks.

A representative approach to addressing this data construction problem is Toolformer, which uses a self-supervised method to generate tool-use data (?). Instead of relying on large-scale human annotations, Toolformer starts from only a few demonstrations for each API. It then lets the language model annotate a large text corpus with possible API calls. After executing these calls, Toolformer keeps only the API calls that help the model better predict future tokens. The filtered data is then used to fine-tune the model.

¹⁰We refer to the response returned by an external tool as an “observation”.

Through this process, the model learns when to call a tool, what arguments to pass, and how to incorporate the tool result into later generation.

Beyond learning to use a small set of tools, later studies further scale tool-use training to larger and more realistic API spaces. API-Bank builds a benchmark and training set for tool-augmented LLMs, where models need to plan, retrieve, and call APIs in executable environments (?). ToolLLM constructs ToolBench with more than 16,000 real-world APIs and uses automatically generated instruction-solution trajectories to train ToolLLaMA (?). Gorilla focuses on API calling at scale and fine-tunes LLaMA-based models to generate accurate API calls. It also uses a document retriever to reduce API hallucination and adapt to changing API documents (?).

6.1.2.3 Learning to Use Tools with Reinforcement Learning

Although SFT can teach LLMs to imitate tool-use demonstrations, it is still limited by the coverage and quality of offline trajectories. The model mainly learns the tool-use patterns that appear in the supervised data. As a result, it may fail to generalize to unfamiliar tools, complex tool combinations, or new interaction patterns. More importantly, SFT does not directly optimize whether a tool call is necessary, whether the selected tool is appropriate, or whether the returned observation improves the final answer. This limitation becomes more serious in multi-step tool-use scenarios, *where the model needs to decide when to call a tool, how to formulate the tool input, how to interpret the tool output, and when to stop using tools.*

To address this limitation, researchers have explored RL for tool use. Instead of only imitating fixed demonstrations, the model can interact with tools during rollout and receive feedback from task outcomes or tool execution results. Given a user task q and a set of available tools $\mathcal{T}$, the model generates a tool-use trajectory:

$$\tau = [s_1, u_1, o_1, \dots, s_T, u_T, o_T, a] \quad (62)$$

where s_t denotes the reasoning state at the t -th step, u_t denotes the model action, o_t denotes the observation returned by the tool or environment, and a denotes the final answer. Here, u_t can be either a natural language reasoning step, a tool invocation with arguments, or a final response. The goal of RL is to optimize the policy so that the generated trajectory receives a higher reward:

$$\max_{\theta} \mathbb{E}_{\tau \sim \pi_{\theta}(\cdot|q, \mathcal{T})} [R(\tau)] \quad (63)$$

In practice, this objective can be optimized with standard RL algorithms such as PPO or GRPO. Recent studies show that such outcome-driven optimization can help LLMs learn more adaptive tool-use behaviors, such as deciding when to search, when to execute code, and how to revise a tool call after receiving an error message (???).

However, RL-based tool use also introduces several unique challenges. First, the action space is more complex than ordinary text generation. A tool-use action may involve tool selection, argument generation, execution timing, and response integration. A small error in any part can lead to tool failure. Second, the reward signal is often sparse and delayed. The environment may only provide a final success signal after the whole trajectory is completed, making it difficult to identify which tool call causes success or failure. Third, tool use may introduce additional costs. Unnecessary tool calls can increase latency and computation cost, while insufficient tool use may lead to incorrect answers. Therefore, the model needs to learn not only how to use tools, but also how to use them efficiently.

A key direction is to design more informative rewards for tool use. Instead of using only a final-answer reward, we can decompose the reward into multiple components:

$$R(\tau) = R_{\text{ans}}(\tau) + \alpha R_{\text{fmt}}(\tau) + \beta R_{\text{exec}}(\tau) + \gamma R_{\text{tool}}(\tau) - \lambda C_{\text{tool}}(\tau) \quad (64)$$

where R_{ans} measures final answer correctness, R_{fmt} measures whether the model follows the required tool-use format, R_{exec} measures whether tool calls can be successfully executed, and R_{tool} measures whether the model selects appropriate tools and passes valid arguments. C_{tool} denotes the cost of tool use, such

as the number of tool calls, latency, or computational expense. The coefficients α , β , γ , and λ control the relative importance of different reward terms.

In application, this reward decomposition provides denser feedback than final-answer rewards. For example, if the model selects the correct tool but passes an invalid argument, the execution reward can penalize the invalid call while preserving credit for the correct tool selection. If the model reaches the correct answer but uses many unnecessary tool calls, the cost term can discourage inefficient behavior. ToolRL systematically studies this reward design problem and shows that fine-grained reward decomposition can make RL training more stable and effective for tool selection and tool application (?).

Another direction is to use RL to improve strategic tool use. For example, ReTool first builds a cold-start model with code-augmented reasoning traces and then applies RL with real-time code execution. During rollout, the model interleaves natural language reasoning with code execution, observes execution results, and revises its subsequent reasoning. This allows the model to discover when and how to invoke a code interpreter based on outcome feedback rather than human-written rules (?). Search-R1 and ReSearch follow a similar idea in retrieval-augmented reasoning. They train models to generate search queries during reasoning and use retrieved information to update later steps, rather than relying on fixed retrieval pipelines or hand-crafted search prompts (??).

So far, we have introduced how RL can enhance the core capabilities of LLM-based agents, including planning and tool use. Since an LLM-based agent is inherently driven by an underlying LLM, its RL training can reuse the general LLM-based RL algorithms introduced in Section ?. The main differences lie in the agentic setting. Compared with standard response optimization, agentic RL involves longer trajectories, environmental feedback, and intermediate decisions such as planning and tool invocation. Thus, the key challenges are not only the choice of RL algorithms, but also trajectory construction, reward design, and environment optimization.

6.2 Environment Design and Scaling

The previous subsections focus on optimizing the agent policy for planning and tool use. In this subsection, we turn to another important component of agentic RL: the environment that produces interaction experience. An environment is more than a testbed. It determines which actions the agent can perform, what observations it receives, how the underlying state changes after each action, and how task success is evaluated. More formally, we can represent an environment as

$$e = (\mathcal{S}, \mathcal{U}, \mathcal{O}, P_e, V_e) \quad (65)$$

where $\mathcal{S}$ is the state space, $\mathcal{U}$ is the set of available actions or tool invocations, and $\mathcal{O}$ is the observation space. The transition function P_e determines how an action changes the environment state, while the verifier V_e determines whether the resulting state satisfies the task objective. This formulation describes how the environment produces observations and feedback, rather than how the agent generates its actions.

The quality of an environment directly affects what an agent can learn. For example, an agent trained in a narrow environment may simply memorize specific APIs, task templates, or interaction patterns. An unreliable environment may return inconsistent observations or incorrect rewards, introducing noise into the learning process. Ideally, an effective environment should provide diverse tasks, executable interactions, reliable state transitions, and verifiable outcomes.

To encourage generalization, we can train the agent over a distribution of environments rather than within a single fixed environment:

$$\theta^* = \arg \max_{\theta} \mathbb{E}_{e \sim p(\mathcal{E}), q \sim p_e(\mathcal{Q})} \mathbb{E}_{\tau \sim \pi_{\theta}(\cdot | q, e)} [R_e(\tau)] \quad (66)$$

where $p(\mathcal{E})$ denotes the environment distribution, and $p_e(\mathcal{Q})$ denotes the task distribution within environment e . The interaction trajectory τ follows the definition introduced in the previous subsections. This

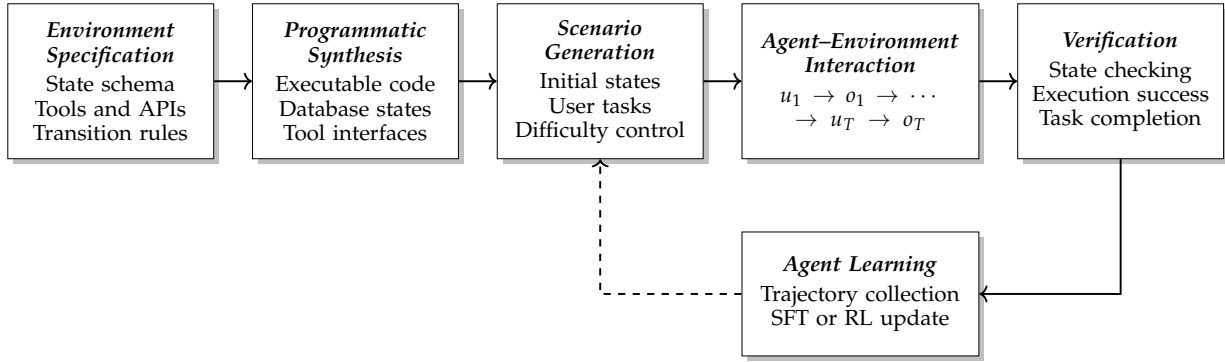


Figure 25: Overview of environment synthesis and interaction for agentic reinforcement learning.

objective shows that agent performance depends not only on policy optimization, but also on the coverage and quality of the environments used for training.

However, constructing high-quality environments manually is expensive and difficult to scale. Real-world systems may be inaccessible, costly, or unsafe for large-scale exploration. Purely language-based simulations are easier to build, but they may generate inconsistent state transitions, invalid tool outputs, or unreliable evaluations. Thus, recent studies explore procedural and programmatic environment synthesis, where executable programs and structured states are used to provide a scalable and reliable interaction experience.

This process is illustrated in Figure ?? . It typically consists of the following steps:

- **Environment Specification.** We first define the state schema¹¹, available tools and APIs, and transition rules of the environment.
- **Programmatic Synthesis.** We then implement the environment as an executable system. This process typically converts the structured specification into program code, database tables, and callable tool interfaces.
- **Scenario Generation.** Based on the environment specification, we generate diverse initial states and user tasks. The tasks should cover different goals, constraints, and difficulty levels. For example, in a shopping environment, a simple task may ask the agent to purchase a single item, while a more difficult task may require it to compare multiple products.
- **Agent-Environment Interaction.** The agent interacts with the generated environment through multiple rounds of actions and observations. At each step, the agent selects a tool and provides the required arguments. The environment executes the action and returns an observation.
- **Outcome Verification.** After the interaction is completed, a verifier checks whether the task has been successfully solved. The verifier may inspect tool execution results, database states, or other structured records.
- **Agent Learning.** Finally, the verified trajectories are used to improve the agent. On the one hand, successful trajectories can serve as demonstrations for SFT, while task-level or step-level feedback can support RL training. On the other hand, failed interactions are also valuable because they reveal weaknesses in the current policy and gaps in the task distribution. This can guide the generation of targeted scenarios that exercise the agent’s weak capabilities and provide more

¹¹A state schema specifies the structured variables used to represent the current status of an environment, together with their types and possible values. For example, in a travel-booking environment, it may define fields for available flights, user preferences, and existing reservations.

informative experience for subsequent training (??). More broadly, this process can be viewed as learning from agentic experience, which we will discuss in Section ??.

Despite this progress, several important questions remain. First, how can we scale both the number and diversity of environments? Second, how can we ensure that generated environments are executable and verifiable? Third, how can agents generalize across heterogeneous and previously unseen environments?

One direction focuses on scaling environment diversity. RandomWorld procedurally generates executable tools and compositional tasks for tool-use agents (?). Unlike static datasets that contain only predefined trajectories, these generated environments support online execution and allow agents to explore different tool combinations during training. AgentScaler further constructs heterogeneous simulated environments and separates the learning of general function-calling behaviors from domain-specific adaptation (?). At a larger scale, DeepSeek-V3.2 synthesizes diverse environments and complex tasks for agentic post-training, showing that broad interaction experience is important for long-tail generalization (?).

Another direction focuses on executability. EnvScaler first constructs environment skeletons that specify state schemas, available tools, and interaction rules. It then generates multiple task scenarios within each environment (?). Agent World Model follows a similar idea by implementing synthetic environments with executable code and database-backed states (?). Compared with purely language-based simulations, these programmatic environments produce more consistent state transitions and generate tool outputs through actual execution.

Verifiability is equally important because RL requires reliable feedback. Instead of evaluating only the textual final response, an environment can check whether the agent has produced the desired state change. Given the final environment state s_T and the task goal g , we can define a state-based reward as

$$R_e(\tau) = V_e(s_T, g) \quad (67)$$

where $V_e(\cdot)$ may be implemented using executable tests, database queries, or rule-based validators. For example, in a database operation task, the verifier can directly check whether the required record has been inserted correctly. This provides more reliable feedback than comparing the generated answer with a reference text. Both EnvScaler and Agent World Model use such mechanisms to connect agent actions with observable state changes.

Scaling environments also introduces substantial heterogeneity. Different environments may vary in task difficulty, available tools, and interaction length. These differences can make RL training unstable and cause the agent to overfit to frequently sampled or easier environments. AutoForge addresses this issue by synthesizing difficult but verifiable tasks and estimating learning signals at the environment level (?). This design reduces the influence of unstable simulated interactions and improves training across heterogeneous environments.

6.3 Learning from Agentic Experience

AI systems are gradually moving from an era dominated by human-generated data toward an era in which agents increasingly learn from their own experience (?). As discussed in Section ??, supervised data remains useful for teaching agents basic behaviors, such as planning, tool invocation, and response formatting. However, human demonstrations alone are unlikely to cover the full range of situations that an agent may encounter in complex environments.

The first approach is memory management. An agent can store useful information from previous interactions, such as successful solutions and failure patterns. When facing a new but related task, the agent can retrieve relevant memories and use them to guide its decisions. In this way, the agent does not need to solve every problem from scratch and can gradually accumulate knowledge across interactions.

The second approach is skill optimization. Individual trajectories often contain reusable procedures, such as how to search for information, recover from an invalid tool call, or complete a common sequence of

operations. The agent can abstract these procedures into higher-level skills and reuse them in future tasks. Skill optimization therefore converts low-level interaction experience into more general and efficient behaviors.

The third approach is trajectory refinement. Not all collected trajectories are optimal, even when they eventually succeed. Some may contain incorrect actions, redundant steps, or inefficient tool-use patterns. The agent can revise these trajectories by correcting failures, removing unnecessary actions, and constructing better reasoning or execution paths. Both successful and failed trajectories can provide useful signals for this refinement process.

6.3.1 *Memory Management*

6.3.2 *Skill Optimization*

6.3.3 *Trajectory Refinement*

6.4 Self-Evolving Agents

7 Multimodal Reinforcement Learning

Multimodal learning involves models that process and relate information from multiple data modalities (e.g. vision, text, speech), enabling more comprehensive understanding than single-modality systems. By combining modalities, models can make more robust predictions and capture complementary information that one modality alone might miss. Such multimodal approaches have benefits across various applications, such as image caption and image generation. Given these advantages, multimodal learning has become increasingly significant as AI systems aim to perceive and reason more like humans, who naturally integrate sight, sound, and language (?).

In the era of LLM, we can extend their capabilities into the multimodal domain by training them with diverse data types. For example, we can develop a visual language model by training an LLM with image-text pairs using SFT. Here, we return to the issue of aligning models with human preferences. Ideally, once aligned with human preferences through RL, the LLM would also generalize the target modality well by the SFT. However, the reality does not always meet expectations. Although an LLM may align well with human preferences in one aspect, it often struggles to generalize this alignment across a different modality. Thus, to align multimodal models with human preferences, we perform RL to enhance their performance further within the specific modality.

In this section, we use the visual language, speech generation, and diffusion models to discuss the application of RL in multimodal models.

7.1 Visual Language Models

While RL is commonly used to train LLMs, its application to other domains has been a prominent research topic. In multimodal language models¹², for example, a notable trend is to perform RL training to improve their trustworthiness and helpfulness. This section considers Visual Language Models (VLMs), which connect a visual encoder to an LLM through a linear projector, facilitating general-purpose visual and language understanding. VLMs are currently the most explored extension in multimodal language research, and they form the foundation for many open-source multimodal language models, such as Qwen2.5-VL (?) and LLaMA-3.2-11B-Vision (?).

¹²A multimodal language model is defined as a model that integrates an LLM with a multimodal encoder, such as CLIP (?), allowing the LLM to process inputs beyond text, such as images. Recent literature has also introduced the use of LLMs for generating outputs in non-text modalities, such as images and speech (??). However, in this section, we focus on the former definition of multimodal language models.

Figure 26: RoVRM

Training LLMs and VLMs with RL exhibits only minimal differences, primarily related to the input content. Unlike the textual input used for LLMs, the input for VLMs typically comprises a combination of one or multiple images and an instruction, denoted by $(\mathbf{I}, \mathbf{x})$, where $\mathbf{I}$ represents the input images. These images are encoded into representations that are either concatenated with instruction embeddings or integrated through cross-attention mechanisms into the LLM. In practice, this subtle difference does not significantly affect the applicability of RL algorithms to VLMs. As a result, the RL training process for LLMs, as described in Section ??, can be seamlessly adapted to train VLMs without major improvements (????).

However, training VLMs with RL is not a low-hanging fruit in practical applications. This is because it typically encounters the challenge of training a visual reward model due to the scarcity of high-quality visual preference data. One straightforward approach to address this issue is to generate visual preference data through the automatic preference data generation method described in Section ?? (?). Another alternative is a multi-stage training approach for the visual reward model, motivated by a simple idea: human preferences are well captured in text, and these preferences can be transferred across modalities (?). By leveraging textual preference data, this approach reduces the dependence on visual preference data in training a visual reward model. More specifically, as illustrated in Figure ??, we can train a visual reward model in the following three stages:

- Stage 1: pre-training with large-scale textual preference data. Given the transferability of human preferences across different modalities, we can begin by using large-scale textual preference data to pre-train the visual reward model. Note that the projector parameters are frozen without images. This stage can be considered as providing a stronger starting point for training the visual reward model, as it enables the model to pre-learn general human preferences.
- Stage 2: fine-tuning with image caption-based preference data. Pre-learned human preferences cannot be directly applied to vision tasks due to both *task gap* and *modality gap*. To bridge the task gap, we fine-tune the reward model using image caption-based preference data in this stage. The rationale behind this approach is that general textual preference data does not cover vision-specific tasks, such as “Please describe the content in this image”. We use such data to fine-tune the model, adapting it to vision-specific tasks. The projector parameters are frozen at this stage.
- Stage 3: fine-tuning with small-scale visual preference data. To further bridge the modality gap, we use visual preference data in the final stage to train the model. Note that in this phase, we also train the projector parameters.

In this process, not all preference data may align with the preferences used in subsequent phases, potentially leading to preference conflicts. We can enhance the visual reward model through data selection techniques, such as LESS (?) to address this. In fact, preference transfer is effective across modalities and has also been shown to work across different tasks and languages (??). Interested readers can refer to these papers for more detailed discussions of these topics.

As discussed in Section ??, reward reasoning models have demonstrated superior performance in reward prediction. A natural question then arises: *can reward reasoning capabilities also be transferred from text to multimodal settings?* Recent work by ? provides empirical evidence supporting this hypothesis. By following a similar multi-stage training paradigm, they show that reward reasoning capabilities can indeed be effectively transferred across modalities, further enhancing the performance of visual reward models.

Apart from preference data, another approach to improving the visual model is to integrate additional image content, such as image captions, into the reward model (?). This approach aims to achieve factually augmented reward prediction, addressing reward hacking. Specifically, in the original setup, the reward model predicts a reward based solely on the image, input, and output; that is, the reward model’s input is

$[I, x, y]$. In the factually augmented setup, the reward model also receives additional input in the form of the textual image caption C , resulting in an input of $[I, C, x, y]$. The basic idea is that the backbone of the visual reward model remains a well-trained LLM with an in-context learning ability. With this ability, we can provide additional content to help the model predict rewards more accurately.

While our discussion primarily focuses on models for visual inputs in the context of visual language models, the RL techniques are adaptable across inputs of various modalities, such as video and audio.

7.2 Speech Generation Models

7.3 Diffusion Models

8 Summary

9 Systems and Datasets

Dataset Name	Sample Size	Response Size	Modality	Feedback	
				Source	Category
HuggingFaceH4/stack-exchange-preferences	10,000	1		Human	Score
Skywork/Skywork-Reward-Preference-80K-v0.2	10,000	2		AI	Ranking
openbmb/UltraFeedback	10,000	3		Rule	
	10,000	4			
					

Table 2: datasets

System Name	Supported Modality				Supported Training Approaches
	Text	Image	Video	Audio	
TRL	✓	✓	✗	✓	Reward Modeling, PPO, GRPO, DPO, Online-DPO, etc.
OpenRLHF	✓	✗	✗	✗	Sft, Reject Sampling, PPO, GRPO, DPO, KTO, etc.
EasyR1	✓	✓	✗	✗	GRPO, Reinforce++, Remax, RLOO, etc.
veRL	✓	✗	✗	✗	GRPO, PPO, Remax, RLOO, SFT, etc.
OpenRLHF-M	✗	✓	✗	✗	PPO, GRPO, RLOO, Online-RLHF, Reject-Sampling, etc.
Align-Anything	✓	✓	✓	✓	PPO, GRPO, DPO, KTO, ORPO, etc.

Table 3: systems