Tong Xiao

Jingbo Zhu

Natural Language Processing

Neural Networks and Large Language Models

NATURAL LANGUAGE PROCESSING LAB
NORTHEASTERN UNIVERSITY

&

NIUTRANS RESEARCH

https://github.com/NiuTrans/NLPBook
https://niutrans.github.io/NLPBook

NATURAL LANGUAGE PROCESSING LAB, NORTHEASTERN UNIVERSITY
&
NIUTRANS RESEARCH

https://github.com/NiuTrans/NLPBook
https://niutrans.github.io/NLPBook

June 12, 2025

Tong Xiao and Jingbo Zhu
June, 2025

Chapter 10

Alignment

Alignment is not a new concept in NLP, but its meaning varies across different domains and over time. In traditional NLP, the term *alignment* typically refers to the tasks that link corresponding elements in two sets, such as aligning words between a Chinese sentence and an English sentence. As LLMs become increasingly important in NLP research, this term is more broadly used to refer to aligning model outputs with human expectations. The problem that alignment addresses is that the output of a model may not align with the specific goals or contexts intended by users. For example, pre-trained LLMs may not be able to follow user instructions because they were not trained to do so. Another example is that LLMs may generate harmful content or perpetuate biases inherent in their training data. This poses new challenges in ensuring that LLM outputs are not only accurate and relevant, but also ethically sound and non-discriminatory.

Simply pre-training LLMs can result in a variety of alignment problems. Our ultimate goal is to resolve or mitigate all these problems to ensure LLMs are both accurate and safe. There is an interesting issue here: since large language models are trained on vast amounts of data, we have reason to believe that if we have sufficient data covering a variety of tasks and aligned with human preferences, pre-training could make LLMs accurate and safe enough, perhaps even eliminating the need for alignment. However, the reality is that it is nearly impossible to gather data that encompasses all tasks or adequately represents human preferences. This makes it difficult to achieve model alignment through pre-training alone, or at least, at this stage, alignment remains a very necessary and critical step in the development of LLMs.

In this chapter, we will focus on alignment methods for LLMs. We will begin by discussing the general alignment tasks. Then we will consider two widely-used approaches, known as **instruction alignment** and **human preference alignment**, respectively. The former resorts to supervised fine-tuning techniques and guides the LLMs to generate outputs that adhere closely to user instructions. On the other hand, the latter typically relies on reinforcement learning techniques, where the LLMs are trained based on feedback from humans. While these methods are motivated by different goals, they are commonly used together to develop well-aligned LLMs.

10.1 An Overview of LLM Alignment

Alignment can be achieved in several different ways. We need different methods for LLM alignment because this problem is itself complicated and multifaceted, requiring a blend of technical considerations. Here we consider three widely-used approaches to aligning LLMs.

The first approach is to fine-tune LLMs with labeled data. This approach is straightforward as it simply extends the pre-existing training of a pre-trained LLM to adapt it to specific tasks. An example of this is **supervised fine-tuning** (**SFT**), in which the LLM is further trained on a dataset comprising task-specific instructions paired with their expected outputs. The SFT dataset is generally much smaller compared to the original training set, but this data is highly specialized. The result of SFT is that the LLM can learn to execute tasks based on user instructions. These tasks can either be ones previously encountered in SFT, or new tasks similar to those. For example, by fine-tuning the LLM with a set of question-answer pairs, the model can respond to specific questions, even if not directly covered in the SFT dataset. This method proves particularly useful when it is relatively easy to describe the input-output relationships and straightforward to annotate the data.

The second approach is to fine-tune LLMs using reward models. One difficulty in alignment is that human values and expectations are complex and hard to describe. In many cases, even for humans themselves, articulating what is ethically correct or culturally appropriate can be challenging. As a result, collecting or annotating fine-tuning data is not as straightforward as it is with SFT. Moreover, aligning LLMs is not just a task of fitting data, or in other words, the limited samples annotated by humans are often insufficient to comprehensively describe these behaviors. What we really need here is to teach the model how to determine which outputs are more in line with human preferences, for example, we not only want the outputs to be technically accurate but also to align with human expectations and values. One idea is to develop a reward model analogous to a human expert. This reward model would work by rewarding the LLM whenever it generates responses that align more closely with human preferences, much like how a teacher provides feedback to a student. To obtain such a reward model, we can train a scoring function from human preference data. The trained reward model is then used as a guide to adjust and refine the LLM. This frames the LLM alignment task as a reinforcement learning task. The resulting methods, such as **reinforcement learning from human feedback** (**RLHF**), have been demonstrated to be particularly successful in adapting LLMs to follow the subtleties of human behavior and social norms.

The third approach is to perform alignment during inference rather than during training or fine-tuning. From this perspective, prompting in LLMs can also be seen as a form of alignment, but it does not involve training or fine-tuning. So we can dynamically adapt an LLM to various tasks at minimal cost. Another method to do alignment at inference time is to rescore the outputs of an LLM. For example, we could develop a scoring system to simulate human feedback on the outputs of the LLM (like a reward model) and prioritize those that receive more positive feedback.

The three methods mentioned above are typically used in sequence once the pre-training is complete: we first perform SFT, then RLHF, and then prompt the LLM in some way

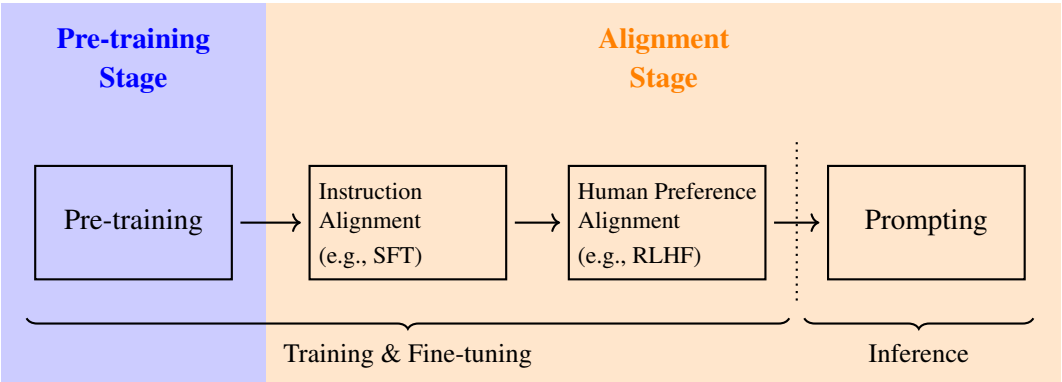


Figure 10.1: Schematic illustration of the pre-train-then-align method for developing LLMs. In the pre-training stage, we train an LLM on vast amounts of data using next token prediction. Then, in the alignment stage, we align the LLM to user instructions, intents, and preferences. This includes instruction alignment, human preference alignment, and prompting.

during inference. This roughly divides the development of LLMs into two stages — the pre-training stage and the alignment stage. Figure 10.1 shows an illustration of this. Since prompting techniques have been intensively discussed in the previous chapter, we will focus on fine-tuning-based alignment methods in the rest of this chapter.

10.2 Instruction Alignment

One feature of LLMs is that they can follow the prompts provided by users to perform various tasks. In many applications, a prompt consists of a simple instruction and user input, and we want the LLM to follow this instruction to perform the task correctly. This ability of LLMs is also called the instruction-following ability. For example, below is a prompt where we want the LLM to extract key points and provide a concise summary for a lengthy article.

Instruction	Summarize this text in three sentences.
Input	Daylight Savings Time (DST) - the process of moving clocks forward by one hour in the summer - was started in Germany in 1916. During World War One it was a way to save ...
Output	____

This task requires the LLM to understand the instruction “Summarize this text in three sentences” and perform the summarization accordingly. However, LLMs are typically trained for next-token prediction rather than for generating outputs that follow instructions. Applying a pre-trained LLM to the above example would likely result in the model continuing to write the input article instead of summarizing the main points. The goal of instruction alignment

(or **instruction fine-tuning**) is to tune the LLM to accurately respond to user instructions and intentions. The rest of this section will discuss some issues related to instruction alignment, including fine-tuning LLMs to follow instructions, generating or collecting instruction data, and generalizing instruction alignment.

10.2.1 Supervised Fine-tuning

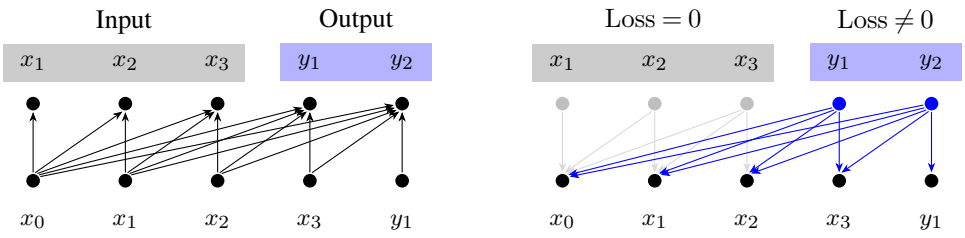
One straightforward approach to adapting LLMs to follow instructions is to fine-tune these models using annotated input-output pairs [Ouyang et al., 2022; Wei et al., 2022]. Unlike standard language model training, here we do not wish to maximize the probability of generating a complete sequence, but rather maximize the probability of generating the rest of the sequence given its prefix (i.e., generating the output given the input). This approach makes instruction fine-tuning a bit different from pre-training. Let $\mathbf{x} = x_0 ... x_m$ be an input sequence (e.g., instruction + user input) and $\mathbf{y} = y_1 ... y_n$ be the corresponding output sequence. The SFT data is a collection of such input-output pairs (denoted by S), where each output is the correct response for the corresponding input instruction. For example, below is an SFT dataset

$\mathbf{x}$ (instruction + user input)	$\mathbf{y}$ (output)
Summarize the following article. Article: In recent years, solar energy has seen unprecedented growth, becoming the fastest-growing ...	{∗summary∗}
Analyze the sentiment of the following review. Review: I absolutely loved the new dining experience. The food was divine and the service was impeccable.	Positive
Translate the following sentence into French. Sentence: practice indeed helps.	La pratique aide effectivement.
Extract the main financial figures from the following earnings report. Report: The company reported a revenue of $10 million in the first quarter with a profit margin of 15% ...	Revenue: $10 million, Profit Margin: 15%
Classify the following email as spam or not spam. Text: Congratulations! You've won a $500 gift card. Click here to claim now.	Spam
Provide a solution to the following technical issue. Issue: my computer is running slow and often freezes.	First, check for ...

where the instructions are highlighted. This dataset contains instructions and the corresponding outputs for several different NLP problems, and so we can fine-tune an LLM to handle multiple tasks simultaneously.

In SFT, we aim to maximize the probability of the correct output given the input. Consider an LLM with pre-trained parameters $\hat{\theta}$. The fine-tuning objective can then be formulated as:

$$\tilde{\theta} = \arg\max_{\hat{\theta}^+} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \log \Pr_{\hat{\theta}^+}(\mathbf{y}|\mathbf{x}) \tag{10.1}$$



(a) Forward Pass (b) Backward Pass

Figure 10.2: Illustration of supervised fine-tuning for LLMs. We concatenate the input and the output into a single sequence. During the forward pass, we run the LLM as usual. During the backward pass, we compute the loss only for the output part and simply set the loss for the input part to 0.

where $\tilde{\theta}$ denotes the parameters optimized via fine-tuning, 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. But the reader should keep in mind that the fine-tuning starts from the pre-trained parameters rather than randomly initialized parameters.

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 $\mathbf{y}_{<i}$:

$$\log \Pr_\theta(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^{n} \log \Pr_\theta(y_i|\mathbf{x}, \mathbf{y}_{<i}) \tag{10.2}$$

This formulation is equivalent to minimizing the cross-entropy loss.

Note that minimizing the conditional log-probability $\log \Pr_\theta(\mathbf{y}|\mathbf{x})$ is not a standard language model training problem. If we concatenate $\mathbf{x}$ and $\mathbf{y}$ as a single sequence, a more general form of language modeling is based on the joint log-probability $\log \Pr_\theta(\mathbf{x}, \mathbf{y})$, that is, we minimize the loss over all tokens of the sequence $\text{seq}_{\mathbf{x},\mathbf{y}} = [\mathbf{x}, \mathbf{y}]$. We can write the probability of this sequence using the chain rule

$$\begin{aligned} \log \Pr_\theta(\text{seq}_{\mathbf{x},\mathbf{y}}) &= \log \Pr_\theta(\mathbf{x}, \mathbf{y}) \\ &= \underbrace{\log \Pr_\theta(\mathbf{x})}_{\text{set to 0}} + \underbrace{\log \Pr_\theta(\mathbf{y}|\mathbf{x})}_{\text{loss computation}} \end{aligned} \tag{10.3}$$

There are two terms on the right-hand side of the equation. We can simply set the first term $\log \Pr_\theta(\mathbf{x})$ to 0, focusing solely on the second term $\log \Pr_\theta(\mathbf{y}|\mathbf{x})$ for loss computation. As a result, the training can be implemented using standard LLMs. For the sequence $\text{seq}_{\mathbf{x},\mathbf{y}}$, we first run the forward pass as usual. Then, during the backward pass, we force the loss corresponding to $\mathbf{x}$ to be zero. Figure 10.2 shows an illustration of this process.

By taking $\log \Pr_\theta(\text{seq}_{\mathbf{x},\mathbf{y}})$ as the objective function, we can describe SFT using a regular

form of language model training:

$$\tilde{\theta} \quad = \quad \arg\max_{\theta} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \log \Pr_{\theta}(\text{seq}_{\mathbf{x},\mathbf{y}}) \tag{10.4}$$

The problem we considered above is fundamentally a **single-round prediction** problem, where the LLM generates a response based on a single input without any further interaction or feedback from the user. The input is processed, and the output is generated in one go. This is typical in scenarios where a single question is asked, and a single answer is provided, with no follow-up questions or clarifications. However, in practice, we sometimes have to handle multi-round prediction problems, for example, an LLM engages in a dialogue over multiple turns. In this setting, the LLM not only generates responses based on the initial input but also incorporates subsequent inputs that might refine or expand on earlier interactions. For example, we can use the LLM to act as a healthcare assistant chatbot and have a conversation with the user, like this

User	I've been feeling very tired lately.
Chatbot	I'm sorry to hear that. Besides feeling tired, have you noticed any other symptoms?
User	Yes, I'm also experiencing headaches frequently.
Chatbot	How long have these symptoms been going on?
User	About a week now.
Chatbot	It might be good to check in with a healthcare professional. Would you like help setting up an appointment?
User	Yes, please. Can it be after work hours?
Chatbot	Sure, I can arrange that. There are slots available next Wednesday and Thursday after 5 PM. Which day works better for you?
	...

In this task, there are several rounds of conversation, each involving the generation of a response based on the user's request or question and the conversational history. Suppose we have K rounds of conversation, denoted by $\{\mathbf{x}^1, \mathbf{y}^1, \mathbf{x}^2, \mathbf{y}^2, ..., \mathbf{x}^K, \mathbf{y}^K\}$. Here $\mathbf{x}^k$ and $\mathbf{y}^k$ denote the user request and the response, respectively, for each round k. The log-probability of generating the response can be written as $\log \Pr_{\theta}(\mathbf{y}^k | \mathbf{x}^1, \mathbf{y}^1, ..., \mathbf{x}^k)$. Our goal is then to maximize the sum of these log-probabilities

$$\tilde{\theta} \quad = \quad \arg\max_{\theta} \sum_{k=1}^{K} \log \Pr_{\theta}(\mathbf{y}^k | \mathbf{x}^1, \mathbf{y}^1, ..., \mathbf{x}^k) \tag{10.5}$$

A straightforward implementation of this involves calculating the conditional probability for each k. However, it requires running the LLM K times, each time with an increased conversational history to make predictions. A more efficient method is to perform loss computation of all responses in a single run of the LLM. To do this, we represent the conversation as a sequence $\text{seq}_{\mathbf{x}^1,\mathbf{y}^1,...,\mathbf{x}^K,\mathbf{y}^K} = [\mathbf{x}^1,\mathbf{y}^1,...,\mathbf{x}^K,\mathbf{y}^K]$ (or seq for short). The log-probability of this sequence is given by

$$\begin{aligned}\log \Pr_\theta(\text{seq}) &= \log \Pr_\theta(\mathbf{x}^1,\mathbf{y}^1,...,\mathbf{x}^K,\mathbf{y}^K) \\ &= \underbrace{\log \Pr_\theta(\mathbf{x}^1)}_{\text{set to 0}} + \underbrace{\log \Pr_\theta(\mathbf{y}^1|\mathbf{x}^1)}_{\text{loss computation}} + \cdots + \\ &\quad \underbrace{\log \Pr_\theta(\mathbf{x}^K|\mathbf{x}^1,\mathbf{y}^1,...,\mathbf{y}^{K-1})}_{\text{set to 0}} + \\ &\quad \underbrace{\log \Pr_\theta(\mathbf{y}^K|\mathbf{x}^1,\mathbf{y}^1,...,\mathbf{x}^K)}_{\text{loss computation}} \end{aligned} \tag{10.6}$$

The trick here is that we ignore the loss for generating user inputs (i.e., $\log \Pr_\theta(\mathbf{x}^1)$,..., $\log \Pr_\theta(\mathbf{x}^K|\mathbf{x}^1,\mathbf{y}^1,...,\mathbf{y}^{K-1})$), as illustrated in Figure 10.3. Hence we only compute the probabilities of generating the responses given their conversational histories, in other words, the value on the right-hand side of Eq. (10.6) is actually equal to the value on the right-hand side of Eq. (10.5). As with Eq. (10.4), the training of this multi-round prediction model can be achieved by maximizing the log likelihood over a training dataset $\mathcal{D}$:

$$\tilde{\theta} = \arg\max_\theta \sum_{\text{seq}\in\mathcal{D}} \log \Pr_\theta(\text{seq}) \tag{10.7}$$

While implementing the SFT methods introduced above seems trivial as they are fundamentally the same as regular language model training, there are still issues that need to be considered in practice. For example,

- SFT requires labeled data. This makes SFT quite different from pre-training, where raw text is used as training data and is readily available. As in other supervised machine learning problems, data annotation and selection in SFT are not simple tasks. In general, we wish to develop SFT data that is both substantial in quantity and high in quality, and this data should be highly relevant to the tasks the LLM will perform. On the other hand, there is a need to fine-tune LLMs with less data to minimize computational and data construction costs. Often, the quality of LLMs is highly dependent on the data used in SFT. Thus, such data must be carefully developed and examined. As we will see in later subsections, SFT can be more efficient and effective through more advanced techniques for data construction.
- SFT is still computationally expensive for LLMs due to their large size. As a result, maintaining and updating such models is resource-intensive. For example, applying gradient updates to billions of parameters within an LLM requires significant computational power and memory. This often requires high-performance computing environments,

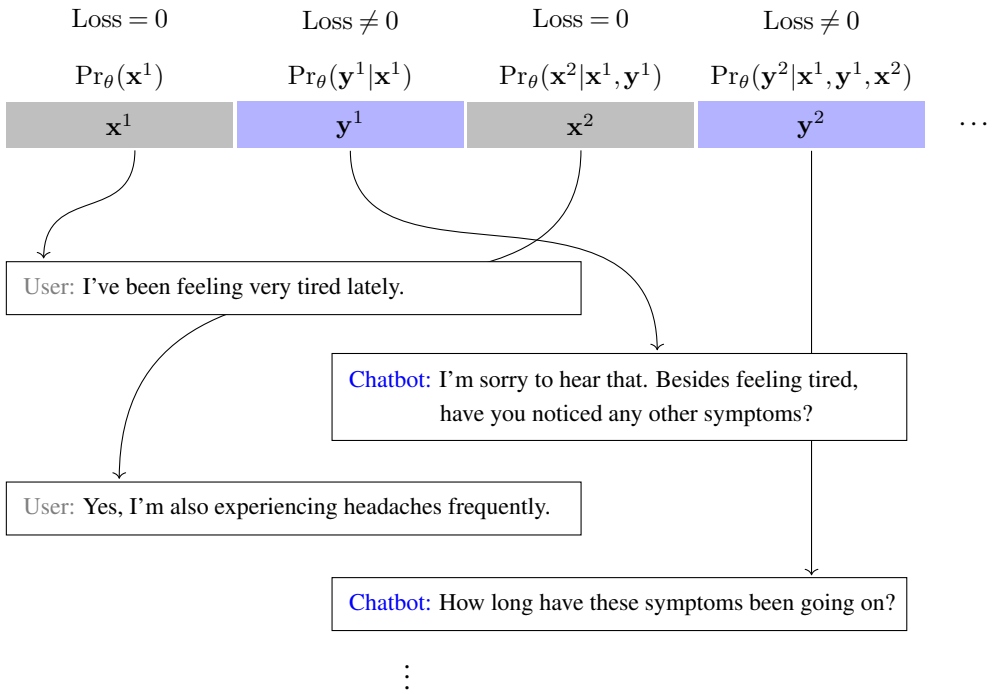


Figure 10.3: Illustration of supervised fine-tuning for conversational models. Here the LLM acts as a chatbot to respond to each request based on the conversational history. The conversation progresses by alternating between the user and the chatbot. In SFT, we treat the entire conversation as a sequence, just like in standard LLMs, but compute the loss only for the responses of the LLM.

which are costly to operate. To address these challenges, various optimization strategies, such as pruning, quantization, and the use of more efficient training algorithms, have been explored. In particular, there has been significant interest in parameter-efficient fine-tuning methods which are designed to maintain state-of-the-art performance without the need for extensive computational resources. We have seen in Chapter 9 that applying techniques like soft prompts can make the fine-tuning process more efficient. For further discussion on parameter-efficient methods, the reader can refer to related papers on this issue [Houlsby et al., 2019; Hu et al., 2022; Han et al., 2024].

- SFT can be regarded as a post-training step following pre-training. It is a separate training phase designed to preserve the advantages of the initial pre-training while incorporating new adjustments. This may seem paradoxical because updating a pre-trained LLM with further data potentially causes the model to forget some of its prior knowledge. Imagine a scenario where we have a large amount of SFT data and extensively fine-tune the LLM. In this case, the LLM could overfit the data, which in turn may reduce generalization performance or cause catastrophic forgetting. A common strategy to mitigate this issue is to employ regularization and early stopping techniques. Another

practical approach is to use a smaller learning rate to gently adjust the weights of the LLM. In addition, fine-tuning with data from diverse sources and problem domains can also be beneficial. Nevertheless, in practice, the SFT step is often carefully examined and requires substantial engineering and experimental efforts to optimize.

10.2.2 Fine-tuning Data Acquisition

Fine-tuning data is so important that much recent work in LLM has focused on developing various datasets for instruction fine-tuning. As with most work in machine learning, there are generally two approaches to data acquisition — manual data generation and automatic data generation.

1. Manually Generated Data

One straightforward method is to recruit human annotators to create input-output pairs for the tasks of interest. Unlike data annotation in conventional NLP, such as text classification, where annotators simply assign labels to collected texts according to guidelines, creating fine-tuning data for LLMs requires more steps and effort, making it thus more challenging. Suppose we want to obtain fine-tuning data for the English-to-Chinese machine translation task. The first step is to write a prompt template to describe the task and format the problem clearly. For example,

Instruction	Translate the text from English to Chinese.
User Input	$\{*\text{text}*\}$
Output	$\underline{\{*\text{translation}*\}}$

Then, we collect pairs of source and target texts (i.e., Chinese texts and the corresponding translations), and replace the variables $\{*\text{text}*\}$ and $\{*\text{translation}*\}$ to generate the fine-tuning samples. For example, given a pair of English and Chinese sentences

$$\begin{array}{ccc} \text{How's the weather today?} & \rightarrow & \text{今天天气怎么样?} \\ \{*\text{text}*\} & & \{*\text{translation}*\} \end{array}$$

we can generate a fine-tuning sample using the prompt template, like this

Instruction	Translate the text from English to Chinese.
User Input	How's the weather today?
Output	$\underline{\text{今天天气怎么样?}}$

That is,

$$\begin{aligned}\mathbf{x} &= \text{Translate the text from English to Chinese.\textbackslash n How's the weather today?}\\ \mathbf{y} &= \text{今天天气怎么样?}\end{aligned}$$

We can use this $(\mathbf{x}, \mathbf{y})$ pair to fine-tune the LLM, as described in the previous subsection.

One difficulty here is that there are many, many different ways to write prompt templates for the same task, and different people may produce prompt templates with varying qualities and complexities. Sometimes, we may write prompt templates with overly complex or verbose instructions. Sometimes, we may not even know exactly what the target task is and how to describe it. A widely-adopted strategy is to create prompt templates for existing NLP tasks, given that there have been so many well-established NLP problems and benchmarks [Bach et al., 2022; Wang et al., 2022; Mishra et al., 2022]. In this case, annotators can be given the original task description and many examples. Then, they can use their own ways to express how to prompt the LLM to perform the tasks. Note that, while such a method can ease the process of creating and writing prompts, we still need annotation frameworks and crowdsourcing systems to manage the work and conduct quality control. For example, we generally need to design annotation guidelines and a unified format for writing prompt templates, especially when many annotators are contributing to the same task. One advantage of inducing prompts from existing NLP tasks is that, once the prompt templates have been developed, it is easy to generate prompts using the annotated samples in the original tasks. For example, given a bilingual dataset for English-to-Chinese translation, we can easily create a number of fine-tuning examples by filling the slots in the above template with the sentence pairs in this dataset.

Another approach is to directly use the naturally existing data available on the internet. A common example is by collecting question-and-answer pairs from QA websites to fine-tune LLMs for open-domain QA tasks [Joshi et al., 2017]. Many benchmarks in QA are built in this way because there are so many types of questions that it is impossible to think of them all by a small group of people. Instead, using data from those websites can ensure that the LLM fine-tuning data is at a good or acceptable level in terms of quantity and quality.

In addition to employing existing resources, another straightforward way to develop a fine-tuning dataset is to crowdsource the data. A simple approach is to allow users to input any question, after which responses are either manually given or automatically generated by an LLM and then manually annotated and corrected. It is thus possible to capture real user behavior and consequently gather inputs and outputs for a large number of "new" problems that traditional NLP tasks do not cover.

An issue related to the construction of the fine-tuning datasets is that we usually want the data to be as diverse as possible. Many studies have found that increasing the diversity of fine-tuning data can improve the robustness and generalization ability of LLMs. For this reason, there has been considerable interest in involving more diverse prompts and tasks in LLM fine-tunining datasets. We will provide further discussion on the generalization of fine-tuning in Section 10.2.4.

2. Automatically Generated Data

One limitation of manual data generation is that the quality and diversity largely depend on human experience and creativity. Therefore, if we want LLMs to handle a broad range of tasks, that is, to effectively execute any instruction, relying on human-annotated data for LLM fine-tuning is often inefficient. Moreover, the coverage of such data can be limited, and the data may even contain biases introduced by the annotators themselves. An alternative approach is to generate data automatically. For example, we can collect a number of questions through crowdsourcing, and employ a well-tuned LLM to generate answers to the questions. These question-answer pairs are then used as fine-tuning samples as usual. This method, though very simple, has been extensively applied to generate large-scale fine-tuning data for LLMs.

The above way of producing synthetic fine-tuning data is similar to those used in data augmentation for NLP. If we have an LLM, we can produce a prediction in response to any input. Repeating this process for different inputs allows us to create a sufficient number of fine-tuning samples. Such a method is particularly useful for fine-tuning new LLMs using a well-tuned LLM. However, one disadvantage of this approach is that it relies on human-crafted or collected inputs for data generation, which may turn out to be inappropriate for generalizing LLMs. In many LLM applications, a significant challenge arises from the broad range of users' questions and requests, many of which are not covered in existing NLP tasks and datasets. In these cases, it becomes necessary to generate not only the predictions but also the inputs themselves.

Here we consider **self-instruct** as an example to illustrate how to generate LLM fine-tuning samples [Wang et al., 2023c; Honovich et al., 2023]. The idea is that we can prompt an LLM to create a new instruction by learning from other instructions. Given this instruction, the LLM can then fill in other fields (such as the user input) and produce the predictions. Figure 10.4 shows a schematic illustration of self-instruct. Here we give a brief outline of the key steps involved.

- The self-instruct algorithm maintains a pool of tasks. Initially it contains a number of seed hand-crafted tasks, each with an instruction and input-output sample. As the algorithm proceeds, LLM-generated instructions and samples will be added to this pool.
- At each step, a small number of instructions are drawn from the instruction pool. For example, we can randomly select a few human-written instructions and a few LLM-generated instructions to ensure diversity.

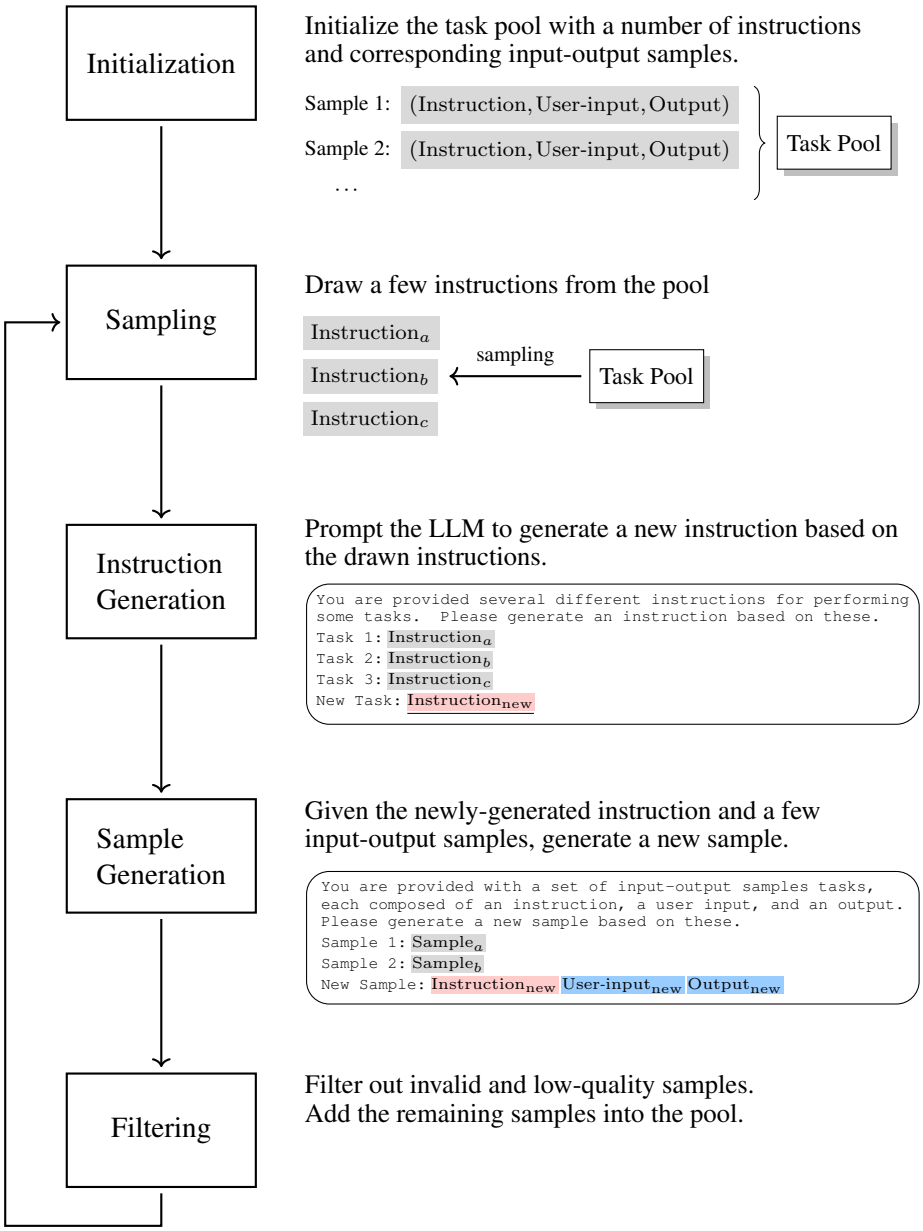


Figure 10.4: Illustration of self-instruct [Wang et al., 2023a]. This method maintains a pool of instructions and corresponding input-output samples. Initially, the pool contains a number of hand-crafted instructions and samples. Each time, we draw a few instructions from the pool. An LLM is then prompted to generate new instructions and samples based on those drawn. Finally, the newly-generated instructions and samples are filtered and added to the pool.

- The selected instructions are then used as demonstration examples. Thus, the LLM can in-context learn from these examples and produce a new instruction. Below is an example template for prompting the LLM.

> You are provided several different instructions for performing some tasks. Please generate an instruction based on these.
>
> Task 1: {instruction1}
>
> Task 2: {instruction2}
>
> Task 3: {instruction3}
>
> Task 4: {instruction4}
>
> New Task: ____

- Given the generated instruction, the LLM is then prompted to complete the sample by filling in the remaining input fields and generating the corresponding output. Below is a prompt template.

> You are provided with a set of input-output samples, each composed of an instruction, a user input, and an output. Please generate a new sample based on these.
>
> Sample 1: {instruction1}
>
> Input: {user-input1}
>
> Output: {output1}
>
> Sample 2: {instruction2}
>
> Input: {user-input2}
>
> Output: {output2}
>
> New Sample: {new-instruction}
>
> ____

- This newly-generated sample is examined by some heuristic rules (such as filtering out samples or instructions that are similar to those already in the pool). If it passes, the sample and instruction are added to the pool.

This generation process can be repeated many times to obtain a sufficient number of fine-tuning samples. Note that, above, we just show simple prompt templates for generating instruction and fine-tuning samples. Of course, we can develop better templates to generate more diverse and accurate instruction and fine-tuning samples. For example, for certain tasks like text classification, the LLM may tend to produce biased predictions, for example, most generated samples belong to a single class. In such cases, we can adjust the order of generation of different fields. More specifically, we can specify the output (i.e., the class) with some prior, and prompt the LLM to generate user input given both the instruction and the output. This

method resembles **input inversion**, where the LLM generates the input based on the specified output [Longpre et al., 2023].

Using LLM-generated instructions and fine-tuning samples has been a common method for developing LLMs, especially given that manually developing such data is so expensive that most research groups cannot afford it. In several well-tuned LLMs, their fine-tuning datasets include a certain amount of synthetic data, which has proved useful [Ouyang et al., 2022; Taori et al., 2023; Chiang et al., 2023]. There have been further studies on synthetic data generation for LLM fine-tuning. For example, one can generate more diverse instructions by introducing evolutionary algorithms [Xu et al., 2024], or use synthetic data as supervision signals in a more advanced fine-tuning process [Chen et al., 2024b]. More recently, there has also been considerable interest in using synthetic data in the pre-training stage [Gunasekar et al., 2023; Allal et al., 2024].

In many applications, a real-world scenario is that, given a task, we can collect or annotate a relatively small amount of fine-tuning data, for example, we can recruit experts to create questions for QA tasks in a specific domain. But the quantity and diversity of this data are in general not sufficient. In this case, we can use self-instruct techniques to generate more diverse question-answer pairs, and thus augment the fine-tuning data. This provides a way of bootstrapping the LLM starting from a seed set of fine-tuning samples. Note that using self-generated data is a common practice and has long been applied in NLP. For example, this approach has been successfully used in parsing and machine translation [Charniak, 1997; Sennrich et al., 2016].

10.2.3 Fine-tuning with Less Data

With the increasing prominence of instruction fine-tuning, there has been a surge in demand for large-scale, high-quality fine-tuning data. For example, the FLAN fine-tuning dataset, which is compiled from 1,836 tasks, contains 15 million samples [Longpre et al., 2023]. Fine-tuning LLMs with such large datasets is typically a computationally expensive task, especially given that updating the large number of parameters in LLMs is resource-intensive. One approach for mitigating this issue is to explore efficient model training methods, for example, one can use parameter-efficient methods to update only a small portion of the model. However, many fine-tuning datasets contain a large amount of synthetic data, where errors and biases are still inevitable.

Another approach to efficient fine-tuning is to consider only the most relevant and impactful examples for fine-tuning. We can thus reduce the amount of data that needs to be processed while still maintaining the quality of the model updates. There are several methods to achieve this. For example, Zhou et al. [2023] built an instruction-following dataset containing only 1,000 samples by carefully crafting the prompts and collecting samples from a variety of NLP tasks. They showed that the LLaMa 65B model fine-tuned with this dataset could be competitive with or even better than models with much more fine-tuning effort. This suggests that LLMs can be adapted to respond to diverse tasks without necessarily needing fine-tuning on all types of instruction-following data. Chen et al. [2024a] developed a system based on the GPT-3.5 model to assess the quality of each instruction-following sample. Therefore,

they could select high-quality samples from existing datasets, showing better fine-tuning performance with fewer fine-tuning samples. Researchers have also developed methods to either select or filter out data using heuristics [Zhao et al., 2024; Ge et al., 2024], or to prioritize data that more significantly influences the fine-tuning process [Xia et al., 2024]. In fact, most of these methods can be seen as instances of larger families of data selection and filtering methods. And it is often the case that using higher quality (but maybe less) data is beneficial for training NLP models.

The discoveries in instruction fine-tuning somewhat differ from traditional views in NLP: the ability of models to handle complex problems can be activated with a small amount of annotated data, rather than requiring massive amounts of supervised data for extensive training. One possible explanation is that the ability of generating correct responses given instructions has been learned during pre-training, but such instruction-response mappings are not with high probabilities during inference. Fine-tuning can slightly adjust the models to get them to follow instructions, requiring significantly less training effort than pre-training. This is closely related to what is known as the **superficial alignment hypothesis**, which suggests that learning primarily occurs during pre-training, and the subsequent fine-tuning or alignment phase does not significantly contribute to the underlying knowledge base of an LLM [Zhou et al., 2023]. Since the core abilities and knowledge of the model are already established from pre-training, effective fine-tuning for alignment with user needs can be achieved with relatively small training fine-tuning effort. This implies the possibility of fine-tuning LLMs with very little data. In another direction, it may not be necessary to restrict fine-tuning to paired instruction-response data. For example, Hewitt et al. [2024] found that instruction-following can be implicitly achieved by fine-tuning LLMs only on responses, without corresponding instructions.

A concept related to the discussion here is sample efficiency. A machine learning method is called **sample efficient** if it can learn effectively from a small number of training examples. In this sense, instruction fine-tuning is sample efficient compared with pre-training. From the perspective of machine learning, sample-efficient methods can be seen as efficient ways to sample the space of data, and are advantageous as they make optimal use of scarce data. Therefore, sampling-based learning techniques, such as many reinforcement learning algorithms, can benefit from these sample efficient approaches. For example, in human preference alignment, we can either efficiently sample preference data via reward models [Liu et al., 2024] or improve the sampling efficiency in policy learning [Wang et al., 2024].

10.2.4 Instruction Generalization

In many machine learning and NLP problems, training a model to generalize is a fundamental goal. For example, in text classification, we expect our model to correctly classify new texts that were not seen during training. However, generalization poses additional challenges in instruction fine-tuning. We expect instruction-fine-tuned LLMs to not only generate appropriate responses for different inputs within a task but also to accurately perform various tasks as described by different instructions. To illustrate this issue, consider an LLM $\Pr(\mathbf{y}|\mathbf{c},\mathbf{z})$, where $\mathbf{c}$ is an instruction, $\mathbf{z}$ is a user input, and $\mathbf{y}$ is the corresponding model output (i.e., the

response). Suppose that the performance of this model is evaluated in terms of a metric, written as $\text{Performance}(\Pr(\mathbf{y}|\mathbf{c},\mathbf{z}))$ or $\mathrm{P}(\mathbf{c},\mathbf{z},\mathbf{y})$ for short. Informally, when we say this model can generalize within a given task (indicated by the instruction $\mathbf{c}^*$), we mean that there may be a value ϵ such that the average performance on new inputs is above this value:

$$\frac{1}{|\mathcal{Z}|}\sum_{\mathbf{z}'\in\mathcal{Z}}\mathrm{P}(\mathbf{c}^*,\mathbf{z}',\mathbf{y}')>\epsilon \tag{10.8}$$

where $\mathcal{Z}$ is the set of new inputs, and $\mathbf{z}'$ and $\mathbf{y}'$ are an input in this set and the corresponding output, respectively.

Likewise, we can say that this model can generalize across tasks if the average performance over all instruction-input pairs is above some ϵ:

$$\frac{1}{|\mathcal{D}|}\sum_{(\mathbf{c}',\mathbf{z}')\in\mathcal{D}}\mathrm{P}(\mathbf{c}',\mathbf{z}',\mathbf{y}')>\epsilon \tag{10.9}$$

where $\mathcal{D}$ is the set of new instruction-input pairs.

Here, we need to deal with variations in two dimensions: instruction and user input. This makes the generalization problem very complex, because, intuitively, a model needs to learn from a vast number of tasks and different input-output pairs associated with each task to achieve good generalization. As we have discussed several times in this book, achieving such generalization incurs much lower cost than pre-training. In general, fine-tuning LLMs with instruction-response data to some extent can lead to models yielding instruction following on new tasks. Nevertheless, it is typically believed that certain efforts are still needed to adapt LLMs to make them understand and execute instructions broadly.

One way to generalize instruction fine-tuning is to increase the diversity of the fine-tuning data. In earlier studies on instruction fine-tuning, researchers developed many datasets, covering a wide variety of NLP tasks and different instructions for each task [Wang et al., 2022; Sanh et al., 2022; Longpre et al., 2023]. By transforming these tasks into a unified format, one can fine-tune an LLM with a sufficiently large number of samples, for example, there have been several instruction fine-tuning datasets that involve over 100 NLP tasks and 1M samples. However, these early datasets mostly focus on existing academic problems, but not those that users want to deal with in real-world applications. Much recent work has shifted focus to addressing new and more practical problems. For example, there has been considerable interest in constructing datasets that contain large and complicated demonstrations and responses from SOTA models to real user queries [Wang et al., 2023b; Teknium, 2023].

Perhaps the use of large and diverse fine-tuning datasets has its origins in attempts to scale LLMs in different dimensions. Indeed, scaling laws have been used broadly to motivate the development of a wide range of different instruction-fine-tuned LLMs. And it is reasonable to scale instruction fine-tuning to make an LLM follow broad instructions. From the perspective of LLM alignment, however, scaling instruction fine-tuning might not be efficient to achieve generalization.

One problem is that instruction fine-tuning relies on supervised learning that learns to

generalize and perform tasks based on instruction-response mappings. However, such an approach does not capture subtle or complex human preferences (e.g., tone, style, or subjective quality) because these are hard to encode as explicit instruction-response data. Moreover, the generalization performance is bounded by the diversity and quality of the instruction-response dataset. Given these limitations, we would instead like to employ preference models as an additional fine-tuning step following instruction fine-tuning, so the LLMs can generalize further (see Section 10.3).

Another view is that some instruction-response mappings may already be learned during pre-training, and so the pre-trained LLMs have encoded such mappings. However, since we often do not know exactly what data is used in the pre-training, it is hard to judge whether we need to learn such mappings in the fine-tuning. A related question is whether out-of-distribution generalization is primarily achieved during pre-training or fine-tuning. While directly answering this question is beyond the scope of this chapter, it has been shown that pre-training on large and diverse datasets is effective in improving out-of-distribution performance [Hendrycks et al., 2020; Radford et al., 2021; Gunasekar et al., 2023]. This raises an interesting problem: if an LLM has been well pre-trained at scale, fine-tuning may not be as essential for out-of-distribution generalization, since the model may have already encountered sufficient distributional variation. This prompts researchers to fine-tune LLMs with modest effort or to explore new methods to achieve instruction-following. As discussed in the previous subsection, for example, instruction following can be yielded by fine-tuning on a small number of carefully selected instruction-response pairs [Zhou et al., 2023], or even by using methods that are not explicitly designed to do so [Kung and Peng, 2023].

The above discussion provides two different strategies: one requires scaling up fine-tuning datasets for larger diversity, the other requires small but necessary fine-tuning datasets for efficient LLM adaptation. However, in practice, involving diverse instructions often helps. In many cases, we need to adapt our LLM for specific purposes. But the LLM, which has possibly encoded broad instruction-following mappings during pre-training, might tend to behave as a general-purpose instruction executor even with modest fine-tuning. An interesting phenomenon is that when fine-tuning on math data, the resulting LLM might not specialize in math outputs. Instead, this model might respond normally to general instructions, for example, it could generate poetry if instructed to do so [Hewitt, 2024]. This is not a bad thing, but it shows that LLMs may not easily change their nature of following general instructions. In this case, additional adaptations with more diverse data may help adjust the way the LLM follows instructions, particularly for those tasks we aim to address.

10.2.5 Using Weak Models to Improve Strong Models

So far we have explored a variety of instruction fine-tuning methods based on labeled data. One of the limitations of many such methods is that they require the data to be annotated by humans or generated by strong LLMs, which can provide accurate supervision signals in fine-tuning. However, in many cases, the LLM we have in hand is already strong (or at least is advantageous in specific aspects of problem solving), and thus it is not easy to find a superior model for supervision. Even for human experts, when the problem becomes complex,

providing correct and detailed answers might be difficult, or sometimes infeasible. For example, when faced with an extremely long document, the experts would find it challenging to identify any inconsistencies, subtle biases, or missing key points without conducting an exhaustive and time-consuming review.

One may ask at this point: can we use weak LLMs to supervise strong LLMs? This seems to be a significant challenge, but it may reflect a future scenario where we need to supervise AI systems that are smarter than humans or any other AI systems [Burns et al., 2023b]. The problem of using smaller, less complex models to improve the training of larger, more complex models is also called the **weak-to-strong generalization** problem. While there have not been mature approaches to weak-to-strong generalization, using smaller models to assist stronger models has indeed proven useful in several areas of LLMs.

For instruction fine-tuning, one of the simplest ways of applying weak LLMs is to use these models to generate synthetic fine-tuning data. Suppose we have a collection of inputs X, where each input includes an instruction and a user input if necessary. For each $\mathbf{x} \in X$, we use a weak LLM $\Pr^w(\cdot)$ to generate a prediction $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \Pr^w(\mathbf{y}|\mathbf{x})$. Then, the strong LLM $\Pr^s_\theta(\cdot)$ can be trained on these generated predictions (see Eq. (10.1)):

$$\tilde{\theta} \quad = \quad \arg\max_{\theta} \sum_{\mathbf{x} \in X} \log \Pr^s_\theta(\hat{\mathbf{y}}|\mathbf{x}) \tag{10.10}$$

where θ is the model parameters.

The above form transforms the fine-tuning problem into a knowledge distillation problem, in other words, we distill knowledge from the weak model to the strong model. Consequently, we can employ various knowledge distillation methods to achieve this goal. However, explaining weak-to-strong fine-tuning from the perspective of knowledge distillation is not straightforward. A major concern is that the strong model may merely imitate or overfit the errors of the weak model and fail to generalize. For example, the fine-tuned strong model still cannot solve difficult problems that the weak model cannot accurately predict. Fortunately, preliminary experiments in this line of research have shown positive and promising results. For example, Burns et al. [2023a] found that fine-tuning the strong pre-trained GPT-4 model with GPT-2-level supervision could improve generalization across several NLP tasks. To measure how the weak model improves the generalization of the strong model, we define the following terms:

- **Weak Performance** ($\mathrm{P}_{\mathrm{weak}}$). This is the test-set performance of the weak model, which can be regarded as the baseline performance.
- **Weak-to-strong Performance**($\mathrm{P}_{\mathrm{weak}\to\mathrm{strong}}$). This is the test-set performance of the strong model that is fine-tuned with the weak model.
- **Strong Ceiling Performance** ($\mathrm{P}_{\mathrm{ceiling}}$). This is the test-set performance of the strong model that is fine-tuned with ground truth data. For example, we fine-tune the strong model with human-annotated predictions and take the resulting model as a ceiling.

Then, the **performance gap recovered** (**PGR**) can be defined as

$$\text{PGR} \quad = \quad \max\left\{0, \frac{\text{P}_{\text{weak}\to\text{strong}} - \text{P}_{\text{weak}}}{\text{P}_{\text{ceiling}} - \text{P}_{\text{weak}}}\right\} \tag{10.11}$$

This metric measures how much of the performance gap between the ceiling model and the weak model can be recovered by the weak-to-strong model. A PGR of 1 indicates that the weak-to-strong fine-tuning can completely closes the performance gap, whereas a PGR of 0 indicates no improvement. In Burns et al. [2023a]'s work, it is shown that PGR can be around 0.8 on 22 NLP classification tasks. It should be noted that, while the potential of weak-to-strong fine-tuning is promising, achieving substantial weak-to-strong generalization remains a challenging goal that needs further investigation [Aschenbrenner, 2024].

Fine-tuning LLMs with weak supervision is just one choice for using small models to improve large models. Although this section primarily focuses on fine-tuning LLMs, we also mention other methods here to give a more complete discussion (see Figure 10.5 for illustrations of these methods).

- Instead of using small models to generate synthetic data, it is also straightforward to incorporate knowledge distillation loss based on these models. For example, a simple loss function that measures the difference between the small and large models can be defined as:

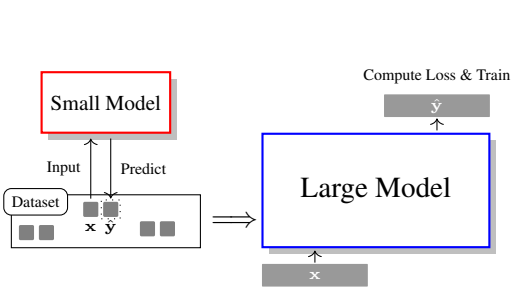
$$\text{Loss}_{\text{kd}} \quad = \quad \text{KL}(\Pr^{w}(\cdot|\mathbf{x}) \,||\, \Pr^{s}_{\theta}(\cdot|\mathbf{x})) \tag{10.12}$$

  Then, we can add this loss to the original loss of language modeling, and yield the following training objective

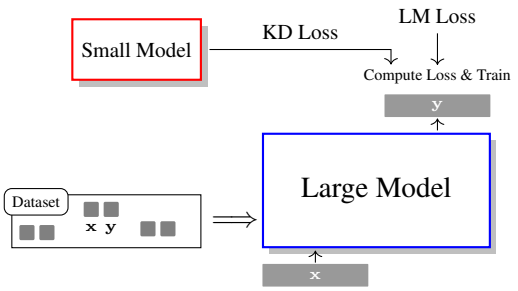
$$\tilde{\theta} \quad = \quad \arg\max_{\theta} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \log \Pr^{s}_{\theta}(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Loss}_{\text{kd}} \tag{10.13}$$

  where $\mathcal{D}$ is the set of input and output pairs, and λ is the coefficient of the interpolation. This method can be employed in either the pre-training or fine-tuning phase. We can adjust λ to control how much the small model influences the training. For example, we can gradually decrease λ to make the training rely more on the original language modeling loss as the large model becomes more capable.

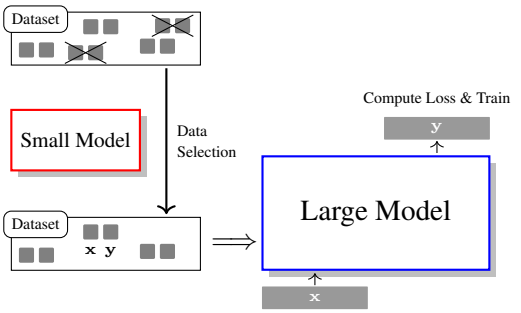
- Another approach to involving small models in LLM pre-training and fine-tuning is to use them to do data selection or filtering. Given a sequence, we can compute the likelihood or cross-entropy using a small model. These quantities can then be used as criteria for selecting or filtering data. For example, sequences with low likelihood or high cross-entropy might be excluded from the training set, as they are less aligned with the small model's learned distribution. Conversely, sequences with high likelihood or low cross-entropy can be prioritized, ensuring that the training focuses on more relevant or high-quality data.
- Ensemble learning is a simple and effective way to build a strong model by combining



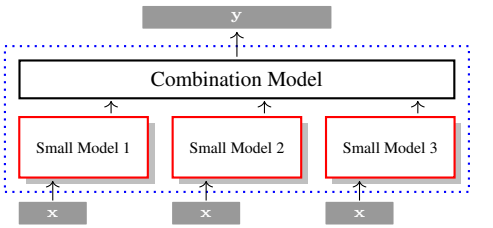
(a) Fine-tuning on data generated by a small model (weak-to-strong generalization)



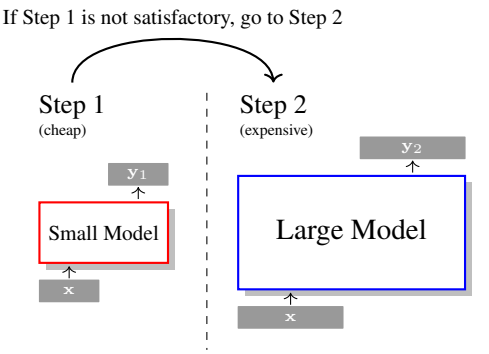
(b) Fine-tuning with KD Loss from a small model (weak-to-strong generalization)



(c) Data selection with a small model



(d) Ensemble of multiple small models



(e) Cascading (at inference time)

Figure 10.5: Illustrations of using small models to improve large models in LLMs. One approach involves using smaller models for the fine-tuning or pre-training of larger models. This includes generating synthetic data (a), incorporating auxiliary loss (b), and selecting appropriate data (c). Another approach involves combining small models and large models. This includes learning a strong model by aggregating multiple small models (d), and cascading small models with large models (e).

multiple weak models. Applying this technique to LLMs is straightforward. We can aggregate distributions predicted by multiple small models or specialized submodels,

and derive the final prediction from the aggregated results. This aggregation can be done using methods such as majority voting, weighted averaging, or stacking.

- Small models can also be employed at inference time to improve overall efficiency. Suppose we have a large model that is slow but more accurate, and a small model that is fast but less accurate. In model cascading, the small model first processes the input data, quickly generating preliminary results. If these results meet certain pre-defined criteria, they can be directly used. However, if the initial results are not sufficiently good, the input is then passed to the larger, more accurate model to produce a better result. This approach significantly reduces computational costs and latency, as the small model can effectively handle many inputs without access to the large model.

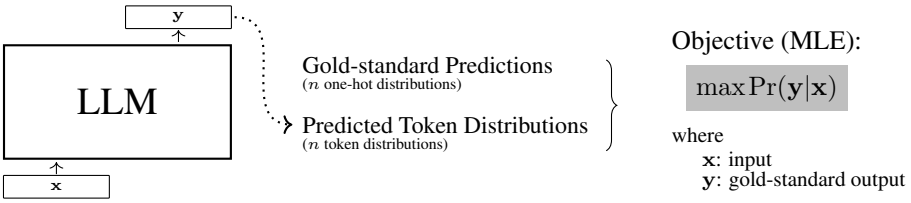
10.3 Human Preference Alignment: RLHF

So far in this chapter, we have focused on fine-tuning LLMs using input-output paired labeled data. This approach allows us to adapt LLMs for instruction-following via supervised learning. In many applications, however, LLMs are required not only to follow instructions but also to act in ways that are more aligned with human values and preferences. Consider a scenario where a user asks an LLM how to hack into a computer system. If the LLM is not appropriately aligned, it may respond by providing details on how to perform this illegal activity. Instead, a more desirable response might be to advise the user against engaging in illegal activities and offer a general overview of the consequences of such actions. The difficulty in achieving this is that the ethical nuances and contextual considerations required for an LLM to respond appropriately in such scenarios are not always straightforward to encode into a fine-tuning dataset. What's even more challenging is that, often, humans themselves cannot precisely express their own preferences.

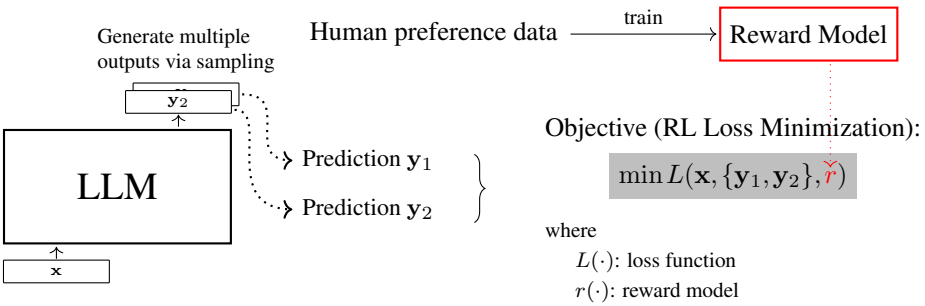
In this section, we discuss an alternative LLM fine-tuning method, called reinforcement learning from human feedback or RLHF for short [Christiano et al., 2017; Stiennon et al., 2020]. The basic idea behind RLHF is that LLMs can learn from comparisons of model outputs using reward models (see Figure 10.6). To do this, we can recruit human experts who indicate their preferences between pairs of outputs generated by the LLM. This preference data is used to train a reward model that can predict the perceived quality of LLM outputs. Once trained, the reward model provides feedback by assigning scores to new outputs that the LLM generates in response to the inputs. The LLM uses these scores to update its parameters through reinforcement learning algorithms. In the rest of this section, we will first introduce the basic knowledge of reinforcement learning to facilitate the discussion, and then discuss methods for training reward models and aligning LLMs with these models.

10.3.1 Basics of Reinforcement Learning

We begin by looking at some basic concepts of reinforcement learning. Note that the notation used here slightly differs from that used in the previous sections and chapters because we want to make our description more consistent with those in the reinforcement learning literature.



(a) Supervised fine-tuning (maximizing the prediction probability given the input)



(b) Reinforcement Learning from Human Feedback

Figure 10.6: Supervised fine-tuning vs. reinforcement learning from human feedback. In supervised fine-tuning, we optimize the LLM by maximizing the probability of the prediction given the input. In reinforcement learning from human feedback, we first train a reward model on human preference data (on each pair of predictions, evaluators are asked to choose which one they prefer). Then, we use this reward model to supervise the LLM during fine-tuning.

Nevertheless, we will show how this notation corresponds to the language modeling notation. The reader who is already familiar with reinforcement learning techniques may skip or skim this subsection.

A general reinforcement learning framework describes how an agent interacts with a dynamic environment. This interaction is modeled as a sequence of actions taken by the agent in response to the state of the environment. At each time step, the agent observes the current state, chooses an action based on its policy, performs the action, and then receives feedback from the environment in the form of a reward and a new state. This sequence of observe-act-receive feedback is repeated until the agent achieves its goal.

A reinforcement learning system involves several components:

- **Agent**. This is the learner or decision-maker in reinforcement learning. In the context of LLMs, it can be seen as the LLM itself.
- **Environment**. This includes everything external to the agent with which the agent interacts. But the environment in LLMs is less about a physical or virtual space and more about the framework within which the agent (e.g., an LLM) receives feedback and

learns.

- **State** (s). A state represents the current situation of the environment. Given a sequence of tokens for language modeling, a state at a time step can be viewed as the tokens we observed so far, that is, the context tokens we take to predict the next token. For example, we can define $(\mathbf{x}, \mathbf{y}_{<t})$ as the state when predicting the next token at the time step t.
- **Action** (a). Actions represent possible decisions the agent can make. We can see them as possible predicted tokens in the vocabulary.
- **Reward** (R). The reward is the feedback from the environment that evaluates the success of an action. For example, $r(s, a, s')$ denotes the reward the agent receives for taking the action a at the state s and moving to the next state s'. If the state-action sequence is given, we can denote the reward at the time step t as $r_t = r(s_t, a_t, s_{t+1})$. Also note that if the decision-making process is deterministic, we can omit s_{t+1} because it can be determined by s_t and a_t. In such cases, we can use $r(s_t, a_t)$ as shorthand for $r(s_t, a_t, s_{t+1})$.
- **Policy** (π). For an LLM, a policy is defined as the probability distribution over the tokens that the LLM predicts, given the preceding context tokens. Formally, this can be expressed as

$$\pi(a|s) = \Pr(y_t|\mathbf{x}, \mathbf{y}_{<t}) \tag{10.14}$$

where a corresponds to the token y_t, and s corresponds to the context $(\mathbf{x}, \mathbf{y}_{<t})$. Figure 10.7 illustrates how an LLM can be treated as a policy in the reinforcement learning framework.

- **Value Function** (V and Q). A **state-value function** (or value function, for short) assesses the expected discounted return (i.e., accumulated rewards) for an agent starting from a particular state s and following a specific policy π. It is defined as:

$$\begin{aligned} V(s) &= \mathbb{E}\Big[r(s_0, a_0, s_1) + \gamma r(s_1, a_1, s_2) + \gamma^2 r(s_2, a_2, s_3) + \cdots \;\Big|\; s_0 = s, \pi\Big] \\ &= \mathbb{E}\Big[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots \;\Big|\; s_0 = s, \pi\Big] \\ &= \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r_t \;\Big|\; s_0 = s, \pi\Big] \end{aligned} \tag{10.15}$$

where $\gamma \in [0, 1]$ is the discount factor that adjusts the importance of future rewards, $s_0 = s$ indicates that the agent starts with the state s, and the expectation $\mathbb{E}$ is performed over all possible trajectories (i.e., state-action sequences). Similarly, an **action-value function** (or **Q-value function**) measures the expected return starting from a state s taking an action a and thereafter following a policy π, given by

$$Q(s, a) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r_t \;\Big|\; s_0 = s, a_0 = a, \pi\Big] \tag{10.16}$$

where $a_0 = a$ indicates that the action taken at the initial state is a.

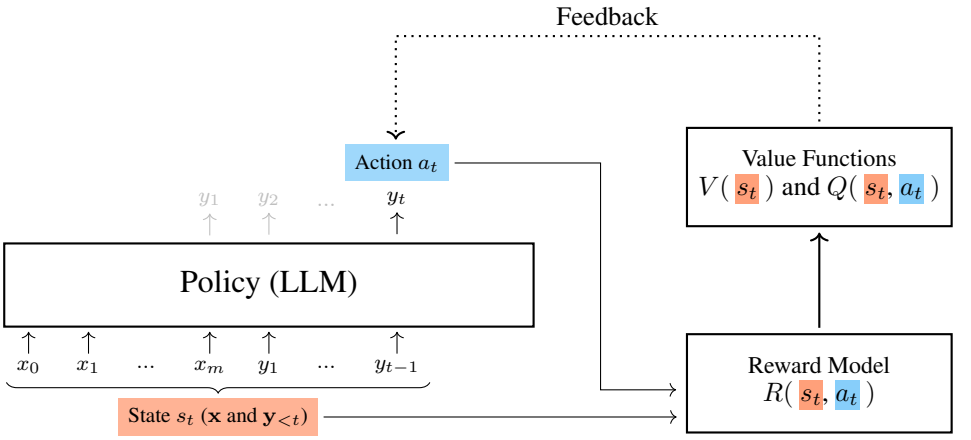


Figure 10.7: LLM as policy in reinforcement learning. At each step t, the LLM predicts a token y_t given the model input $\mathbf{x}$ and the previously-generated tokens $\mathbf{y}_{<t}$. This process can be framed as a reinforcement learning problem, where y_t serves as the action, $(\mathbf{x}, \mathbf{y}_{<t})$ as the state, and the predicted distribution $\Pr(y_t|\mathbf{x}, \mathbf{y}_{<t})$ as the policy. Once y_t is predicted, the LLM inputs both $(\mathbf{x}, \mathbf{y}_{<t})$ and y_t to the reward model, which evaluates how effectively the chosen token contributes to achieving the desired textual outcome. This evaluation generates reward scores which are used to compute the value functions $V(s_t)$ and $Q(s_t, a_t)$. These functions then provide feedback to the LLM and guide the policy training.

The goal of reinforcement learning is to learn a policy that maximizes the **cumulative reward** (or **return**) the agent receives over the long run. Given a state-action sequence $\tau = \{(s_1, a_1), ..., (s_T, a_T)\}$[1], the cumulative reward over this sequence can be written as

$$R(\tau) = \sum_{t=1}^{T} r_t \tag{10.17}$$

The expectation of this cumulative reward over a space of state-action sequences is given in the form

$$\begin{aligned} J(\theta) &= \mathop{\mathbb{E}}_{\tau \sim \mathcal{D}} \Big[R(\tau) \,\Big|\, \pi_\theta \Big] \\ &= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) R(\tau) \\ &= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) \sum_{t=1}^{T} r_t \end{aligned} \tag{10.18}$$

where $\tau \sim \mathcal{D}$ indicates that τ is drawn from the state-action sequence space $\mathcal{D}$, and the subscript

[1]We assume the state-action sequence begins with s_1 and a_1, rather than s_0 and a_0, to align with the notation commonly used in this chapter, where the prediction $\mathbf{y}$ typically starts from y_1. Of course, it is also common to denote a state-action sequence as $\{(s_0, a_0), ..., (s_T, a_T)\}$ or $\{(s_0, a_0), ..., (s_{T-1}, a_{T-1})\}$ in the literature. But this variation in notation does not affect the discussion of the models presented here.

θ indicates the parameters of the policy. $J(\theta)$ is also called the **performance function**.

Then the training objective is to maximize $J(\theta)$:

$$\tilde{\theta} = \arg\max_{\theta} J(\theta) \tag{10.19}$$

Now, we have a simple reinforcement learning approach: 1) we sample a number of state-action sequences; then, 2) we evaluate each sequence using the performance function; then, 3) we update the model to maximize this performance function. If we take Eq. (10.18) and use gradient descent to optimize the policy, this approach would constitutes a form of policy gradient methods [Williams, 1992].

Note that in many NLP problems, such as machine translation, rewards are typically sparse. For instance, a reward is only received at the end of a complete sentence. This means that $r_t = 0$ for all $t < T$, and r_t is non-zero only when $t = T$. Ideally, one might prefer feedback to be immediate and frequent (dense), and thus the training of the policy can be easier and more efficient. While several methods have been proposed to address sparse rewards, such as reward shaping, we will continue in our discussion to assume a sparse reward setup, where the reward is available only upon completing the prediction.

The model described in Eqs. (10.17-10.19) establishes a basic form of reinforcement learning, and many variants and improvements of this model have been developed. Before showing those more sophisticated models, let us take a moment to interpret the objective function $J(\theta)$ from the perspective of policy gradient. In gradient descent, we need to compute the gradient of $J(\theta)$ with respect to θ:

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta} &= \frac{\partial \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) R(\tau)}{\partial \theta} \\
&= \sum_{\tau \in \mathcal{D}} \frac{\partial \Pr_\theta(\tau)}{\partial \theta} R(\tau) \\
&= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) \frac{\partial \Pr_\theta(\tau)/\partial \theta}{\Pr_\theta(\tau)} R(\tau) \\
&= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) \frac{\partial \log \Pr_\theta(\tau)}{\partial \theta} R(\tau)
\end{aligned} \tag{10.20}$$

In some cases, we will assume that every sequence in $\mathcal{D}$ is equally probable (i.e., $\Pr_\theta(\tau) = 1/|\mathcal{D}|$). In this case we can simplify Eq. (10.20) and need only consider the terms $\frac{\partial \log \Pr_\theta(\tau)}{\partial \theta}$ and $R(\tau)$:

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} \sum_{\tau \in \mathcal{D}} \frac{\partial \log \Pr_\theta(\tau)}{\partial \theta} R(\tau) \tag{10.21}$$

One advantage of this result is that $R(\tau)$ does not need to be differentiable, which means that we can use any type of reward function in reinforcement learning.

By treating the generation of the sequence τ as a Markov decision process, we can further derive $\frac{\partial \log \Pr_\theta(\tau)}{\partial \theta}$, and obtain:

$$\begin{aligned}\frac{\partial \log \Pr_\theta(\tau)}{\partial \theta} &= \frac{\partial}{\partial \theta} \log \prod_{t=1}^{T} \pi_\theta(a_t|s_t) \Pr(s_{t+1}|s_t, a_t) \\ &= \frac{\partial}{\partial \theta} \sum_{t=1}^{T} \underbrace{\log \pi_\theta(a_t|s_t)}_{\text{policy}} + \frac{\partial}{\partial \theta} \sum_{t=1}^{T} \underbrace{\log \Pr(s_{t+1}|s_t, a_t)}_{\text{dynamics}} \end{aligned} \tag{10.22}$$

where the gradient is decomposed into two parts: the policy gradient and the dynamics gradient. The policy component, $\log \pi_\theta(a_t|s_t)$, determines the log-probability of taking action a_t given state s_t, and it is parameterized by θ. The dynamics component, $\log \Pr(s_{t+1}|s_t, a_t)$, represents the log-probability of transitioning to state s_{t+1} from state s_t after taking action a_t. In typical reinforcement learning settings, the dynamics are not directly influenced by the policy parameters θ, and thus, their derivatives are often zero. In this case, therefore, Eq. (10.22) can be simplified to:

$$\frac{\partial \log \Pr_\theta(\tau)}{\partial \theta} = \frac{\partial}{\partial \theta} \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \tag{10.23}$$

In other words, we only concentrate on optimizing the policy without concerning ourselves with the underlying dynamics.

Substituting Eq. (10.23) into Eq. (10.21), and expanding $R(\tau)$, we then obtain

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \Big(\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \sum_{t=1}^{T} r_t \Big) \tag{10.24}$$

While this policy gradient approach is straightforward, it suffers from the problem that the variance of the estimated gradients can be very high, making the learning process noisy and inefficient. One reason for this high variance problem is that rewards can vary greatly across different steps or scenarios. Imagine that in a sequence of action decisions, the reward model tends to assign small rewards to good actions (e.g., $R_t = 2$) and large penalties to poor actions (e.g., $R_t = -50$). Such varying reward scales for good and poor actions can result in a very low total reward for the entire sequence, even if it includes good actions.

One simple method for reducing the variance of the gradient is to set a baseline b and subtract it from $\sum_{t=1}^{T} r_t$, resulting in $\sum_{t=1}^{T} r_t - b$.[2] Here, the baseline can be interpreted as a reference point. By centering the rewards around this baseline, we remove systematic biases in

[2] In 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 $\sum_{t=1}^{T} r_t - b$ centers the rewards around zero (e.g., b is defined as the expected value of $\sum_{t=1}^{T} r_t$), which can lead to reduced variance in the product $\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t)(\sum_{t=1}^{T} r_t - b)$.

the reward signal, making the updates more stable and less sensitive to extreme fluctuations in individual rewards.

This policy gradient model with a baseline can be given by

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta} &= \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\frac{\partial}{\partial\theta}\Big(\sum_{t=1}^{T}\log\pi_\theta(a_t|s_t)\Big)\Big(\sum_{t=1}^{T}r_t-b\Big)\\ &= \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\frac{\partial}{\partial\theta}\Big[\sum_{t=1}^{T}\log\pi_\theta(a_t|s_t)\Big(\sum_{k=1}^{T}r_k-b\Big)\Big]\\ &= \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\frac{\partial}{\partial\theta}\Big[\sum_{t=1}^{T}\log\pi_\theta(a_t|s_t)\Big(\sum_{k=1}^{t-1}r_k+\sum_{k=t}^{T}r_k-b\Big)\Big] \end{aligned} \tag{10.25}$$

Here we write $\sum_{k=1}^{T} r_k$ as the sum of two terms $\sum_{k=1}^{t-1} r_k$ and $\sum_{k=t}^{T} r_k$ to distinguish between the rewards accrued before and after the action at time step t. Note that in Markov decision processes, the future is independent of the past given the present. Therefore, the action taken at time step t cannot influence the rewards received before t, or in other words, the rewards prior to t are already "fixed" by the time the action at t is chosen. The term $\sum_{k=1}^{t-1} r_k$ does not contribute to the gradient and can be omitted, leading to a simplified version of Eq. (10.25)

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\frac{\partial}{\partial\theta}\Big[\sum_{t=1}^{T}\log\pi_\theta(a_t|s_t)\Big(\sum_{k=t}^{T}r_k-b\Big)\Big] \tag{10.26}$$

Also note that removing $\sum_{k=t}^{T} r_k$ can further reduce the variance of the gradient.

There are many ways to define the baseline b. Here we consider the value function of the state s_t, that is, the estimated value of being in state s_t: $V(s_t) = \mathbb{E}(r_t + r_{t+1} + \cdots + r_T)$. Hence we have

$$\begin{aligned} A(s_t, a_t) &= \sum_{k=t}^{T} r_k - b \\ &= \sum_{k=t}^{T} r_k - V(s_t) \end{aligned} \tag{10.27}$$

where $\sum_{k=t}^{T} r_k$ represents the actual return received, and $V(s_t)$ represents the expected return. $A(s_t, a_t)$ (or A_t for short) is called the **advantage** at time step t, which quantifies the relative benefit of the action a_t compared to the expected value of following the policy from the state s_t onward.

By using the advantage function $A(s_t, a_t)$, the gradient of $J(\theta)$ can be written in the form

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\frac{\partial}{\partial\theta}\Big(\sum_{t=1}^{T}\log\pi_\theta(a_t|s_t)A(s_t,a_t)\Big) \tag{10.28}$$

This optimization objective corresponds to the **advantage actor-critic** (**A2C**) method in reinforcement learning [Mnih et al., 2016]. In this method, the actor aims at learning a policy. It updates the policy parameters using Eq. (10.28) to help focus more on actions that are likely to improve performance. The critic, on the other hand, updates its estimation of the value function, which is used to calculate the advantage function $A(s_t, a_t)$, thus serving as the evaluator of the policy being learned by the actor.

In the A2C method, $A(s_t, a_t)$ is typically expressed as the difference of the action-value function $Q(s_t, a_t)$ and the state-value function $V(s_t)$

$$A(s_t, a_t) \quad = \quad Q(s_t, a_t) - V(s_t) \tag{10.29}$$

At first glance, this model may seem challenging to develop because it requires two separate sub-models to calculate $Q(s_t, a_t)$ and $V(s_t)$ respectively. Fortunately, considering that $Q(s_t, a_t)$ can be defined as the return $r_t + V(s_{t+1})$, we can rewrite Eq. (10.29) as

$$A(s_t, a_t) \quad = \quad r_t + V(s_{t+1}) - V(s_t) \tag{10.30}$$

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

$$A(s_t, a_t) \quad = \quad r_t + \gamma V(s_{t+1}) - V(s_t) \tag{10.31}$$

$A(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t)$ is also called the **temporal difference** (**TD**) error. What we need is to train a critic network for the value function $V(s_t)$, and then use it to compute the advantage function[3].

Up to this point, we have spent considerable space discussing the basics of reinforcement learning, especially on how to derive the optimization objective for the A2C method. However, reinforcement learning is a vast field, and many technical details cannot be covered here. The interested reader can refer to reinforcement learning books for more details [Sutton and Barto, 2018; Szepesvári, 2010]. Nevertheless, we now have the necessary knowledge to further discuss RLHF. In the subsequent subsections, we will return to the discussion on LLM alignment, demonstrating how to use the A2C method for aligning with human preferences.

10.3.2 Training Reward Models

We have shown that reward models play a very important role in the general reinforcement learning framework and form the basis for computing value functions. We now consider the problem of training these reward models.

In RLHF, a reward model is a neural network that maps a pair of input and output token

[3]The training loss for the value network (or critic network) in A2C is generally formulated as the mean squared error between the computed return $r_t + \gamma V(s_{t+1})$ and the predicted state value $V(s_t)$. Suppose that the value network is parameterized by ω. The loss function is given by

$$\mathcal{L}_v(\omega) \quad = \quad \frac{1}{M} \sum \big(r_t + \gamma V_\omega(s_{t+1}) - V_\omega(s_t)\big)^2 \tag{10.32}$$

where M is the number of training samples, for example, for a sequence of T tokens, we can set $M = T$.

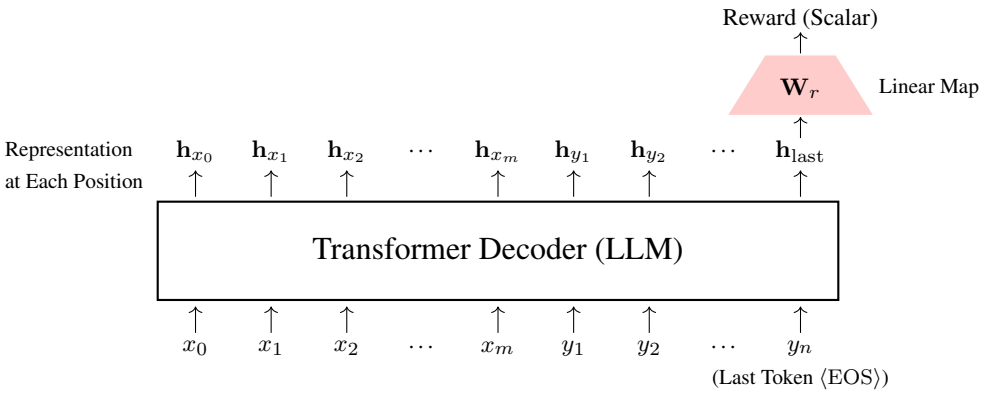


Figure 10.8: Architecture of the reward model based on Transformer. The main component of this model is still an LLM. We use the Transformer decoder as the sequence representation model. We extract the representation of the last position of the decoder as the representation of the entire sequence $[\mathbf{x}, \mathbf{y}]$. We then map this representation to a scalar through a linear transformation, which serves as the reward score for $[\mathbf{x}, \mathbf{y}]$.

sequences to a scalar. Given an input $\mathbf{x}$ and an output $\mathbf{y}$, the reward can be expressed as

$$r \quad = \quad \text{Reward}(\mathbf{x}, \mathbf{y}) \tag{10.33}$$

where $\text{Reward}(\cdot)$ is the reward model. r can be interpreted as a measure of how well the output $\mathbf{y}$ aligns with the desired behavior given the input $\mathbf{x}$. As discussed in the previous subsection, both $\mathbf{x}$ and $\mathbf{y}$ are assumed to complete texts. This means that the reward model evaluates the relationship between inputs and outputs that provide full semantic content. For example, when applying the reward model, it assigns a value of 0 (or another predetermined value) at each position t in the output sequence $\mathbf{y} = y_1 ... y_n$. Only at the final position, when $t = n$, does the reward model generate the actual reward score. To keep the notation uncluttered, we will use $r(\mathbf{x}, \mathbf{y})$ to denote the reward model $\text{Reward}(\mathbf{x}, \mathbf{y})$ from here on.

There are many ways to implement the reward model. One simple approach is to build the reward model based on a pre-trained LLM. 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 a pre-trained LLM on this sequence, as usual, and at each position, we obtain a representation from the top-most Transformer layer. Then, we take the representation at the last position (denoted by $\mathbf{h}_{\text{last}}$) and map it to a scalar via linear transformation:

$$r(\mathbf{x}, \mathbf{y}) \quad = \quad \mathbf{h}_{\text{last}} \mathbf{W}_r \tag{10.34}$$

where $\mathbf{h}_{\text{last}}$ is a d-dimensional vector, and $\mathbf{W}r$ is a $d \times 1$ linear mapping matrix. This architecture of the reward model is illustrated in Figure 10.8.

To train the reward model, the first step is to collect human feedback on a set of generated outputs. Given an input $\mathbf{x}$, we use the LLM to produce multiple candidate outputs $\{\mathbf{y}_1, ..., \mathbf{y}_N\}$.

Human feedback can be obtained in several ways:

- **Pairwise Comparison** (**Pairwise Ranking**). Given two different outputs, human experts select which one is better.
- **Rating**. Human experts provide a score or rating to 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.

Here we consider pairwise comparison feedback as it is one of the simplest and most common forms of human feedback used in RLHF. In this setting, each time, two outputs $(\mathbf{y}_a, \mathbf{y}_b)$ are randomly drawn from the candidate pool $\{\mathbf{y}_1, ..., \mathbf{y}_N\}$. Human experts are then presented with these pairs and asked to decide which output they prefer based on specific criteria, such as clarity, relevance, and accuracy. The human feedback can be encoded as a binary label, $\mathbf{y}_a \succ \mathbf{y}_b$ for a preference for $\mathbf{y}_a$, and $\mathbf{y}_b \succ \mathbf{y}_a$ for a preference for $\mathbf{y}_b$.

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

$$\begin{aligned} \Pr(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x}) &= \frac{e^{r(\mathbf{x},\mathbf{y}_a)}}{e^{r(\mathbf{x},\mathbf{y}_a)} + e^{r(\mathbf{x},\mathbf{y}_b)}} \\ &= \frac{e^{r(\mathbf{x},\mathbf{y}_a) - r(\mathbf{x},\mathbf{y}_b)}}{e^{r(\mathbf{x},\mathbf{y}_a) - r(\mathbf{x},\mathbf{y}_b)} + 1} \\ &= \text{Sigmoid}(r(\mathbf{x},\mathbf{y}_a) - r(\mathbf{x},\mathbf{y}_b)) \end{aligned} \tag{10.35}$$

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

$$\mathcal{L}_r(\phi) = -\mathbb{E}_{(\mathbf{x},\mathbf{y}_a,\mathbf{y}_b)\sim\mathcal{D}_r}\left[\log \Pr_\phi(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x})\right] \tag{10.36}$$

where $(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b)$ is drawn from a human-annotated dataset $\mathcal{D}_r$ consisting of preference pairs of outputs and their corresponding inputs. ϕ represents the parameters of the reward model, which includes both the parameters of the Transformer decoder and the linear mapping matrix $\mathbf{W}_r$. In practice, assuming $(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b)$ is uniformly sampled from $\mathcal{D}_r$, we can replace the expectation with a summation

$$\mathcal{L}_r(\phi) = -\frac{1}{|\mathcal{D}_r|} \sum_{(\mathbf{x},\mathbf{y}_a,\mathbf{y}_b)\in\mathcal{D}_r} \log \Pr_\phi(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x}) \tag{10.37}$$

The goal of training the reward model is to find the optimal parameters $\hat{\phi}$ that minimize

this loss function, given by

$$\hat{\phi} \quad = \quad \arg\min_{\phi} \mathcal{L}_r(\phi) \tag{10.38}$$

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 described in Eq. (10.37). 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 this subsection. This consistency ensures a unified framework for aligning the LLM, regardless of the specific ranking loss used during reward model training.

10.3.3 Training LLMs

Having obtained the reward model, we then train the policy (i.e., the LLM) via the A2C method. Recall from Section 10.3.1 that a state-action sequence or trajectory τ can be evaluated by the utility function

$$U(\tau;\theta) \quad = \quad \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) A(s_t, a_t) \tag{10.39}$$

where $A(s_t, a_t)$ is the advantage of taking the action a_t given the state s_t. An estimate of $A(s_t, a_t)$ is defined as the TD error $r_t + \gamma V(s_{t+1}) - V(s_t)$, where the value function $V(s_t)$ is trained with the reward model.

Given this utility function, the A2C-based loss function can be written in the form

$$\begin{aligned} \mathcal{L}(\theta) \quad &= \quad -\mathbb{E}_{\tau\sim\mathcal{D}}\big[U(\tau;\theta)\big] \\ &= \quad -\mathbb{E}_{\tau\sim\mathcal{D}}\big[\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) A(s_t, a_t)\big] \end{aligned} \tag{10.40}$$

where $\mathcal{D}$ is a space of state-action sequences. As usual, the goal of training the policy is to minimize this loss function

$$\tilde{\theta} \quad = \quad \arg\min_{\theta} \mathcal{L}(\theta) \tag{10.41}$$

If we map the problem back to the language modeling problem and adopt the notation

from LLMs, the loss function can be written as:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}}\Big[U(\mathbf{x},\mathbf{y};\theta)\Big] \tag{10.42}$$

where

$$U(\mathbf{x},\mathbf{y};\theta) = \sum_{t=1}^{T}\log\pi_\theta(y_t|\mathbf{x},\mathbf{y}_{<t})A(\mathbf{x},\mathbf{y}_{<t},y_t) \tag{10.43}$$

Here $\pi_\theta(y_t|\mathbf{x},\mathbf{y}_{<t}) = \Pr_\theta(y_t|\mathbf{x},\mathbf{y}_{<t})$ is the LLM parameterized by θ.

In general, we do not have a human annotated input-output dataset $\mathcal{D}$ in RLHF, but a dataset containing inputs only. The outputs, in this case, are typically the predictions made by the LLM. The loss function is then defined as

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\Big[U(\mathbf{x},\mathbf{y};\theta)\Big] \tag{10.44}$$

where $\mathcal{D}$ denotes the input-only dataset, and $\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})$ denotes that the output $\mathbf{y}$ is sampled by the policy $\pi_\theta(\cdot|\mathbf{x})$.

The above formulation provides a basic form of the A2C method for LLMs. Improved versions of this model are more commonly used in RLHF. In the following discussion, we will still use the reinforcement learning notation to simplify the presentation and will get back the language modeling notation later.

One common improvement of policy gradient methods is to use **importance sampling** to refine the estimation of $U(\tau;\theta)$. This can be written as

$$U(\tau;\theta) = \sum_{t=1}^{T}\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}A(s_t,a_t) \tag{10.45}$$

Here we replace the log-probability $\log\pi_\theta(a_t|s_t)$ with the ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}$. θ_{ref} denotes the parameters of the previous policy (such as an initial model from which we start the training). So $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}$, also called the **ratio function**, can be interpreted as the log-probability ratio between the current policy π_θ and the previous policy $\pi_{\theta_{\text{ref}}}$ (call it the reference policy). By using the ratio function we reweight the observed rewards based on the likelihood of the actions under the current policy versus the reference policy. When $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)} > 1$, the action a_t is more favored by the current policy compared to the reference policy. By contrast, when $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)} < 1$, the action a_t is less favored by the current policy[4].

A problem with the model presented in Eq. (10.47) (as well as in Eq. (10.39)) is that the variance in the gradient estimates is often high, making the learning process unstable. To

[4]Consider a more general case where we wish to evaluate the policy using its expected reward (also see Eq. (10.18))

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\Big[R(\tau)\Big] \tag{10.46}$$

Here $\tau\sim\pi_\theta$ means that the sequence τ is generated by the policy π_θ. Alternatively, we can write $J(\theta)$ in another

mitigate this issue, techniques such as clipping are often employed to bound the importance weights and prevent large updates. A clipped version of the utility function (also called the clipped surrogate objective function) is given by

$$U_{\text{clip}}(\tau;\theta) = \sum_{t=1}^{T} \text{Clip}\Big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}\Big) A(s_t,a_t) \tag{10.49}$$

$$\text{Clip}\Big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}\Big) = \min\Big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, \text{bound}\big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\big)\Big) \tag{10.50}$$

Here the function $\text{bound}(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon)$ constrains the ratio function to the range $[1-\epsilon, 1+\epsilon]$.

A further improvement to the above model is to consider **trust regions** in optimization [Schulman et al., 2015]. In reinforcement learning, a large update to the policy can lead to instability, where the agent may start performing worse after an update. A reasonable idea is to optimize the model in the trust region, 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, ensuring that the current policy does not deviate too significantly from the reference policy. This can be achieved by adding a penalty based on some form of divergence between the current and reference policies to the objective function. A simple form of such a penalty is given by the difference in the log-probability of the sequence τ under the current policy versus the reference policy:

$$\text{Penalty} = \log \pi_\theta(\tau) - \log \pi_{\theta_{\text{ref}}}(\tau) \tag{10.51}$$

---

form

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}}\Big[\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau)\Big] \tag{10.47}$$

It is not difficult to find that the right-hand sides of these equations are essentially the same since $\mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}}\Big[\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau)\Big] = \sum_\tau \Pr_{\theta_{\text{ref}}}(\tau)\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau) = \sum_\tau \Pr_\theta(\tau) R(\tau) = \mathbb{E}_{\tau\sim\pi_\theta}\Big[R(\tau)\Big]$

Note that this equivalence holds only when the expectation is performed over the entire sequence space. In practice, however, we often only sample a relatively small number of sequences using a policy in policy learning. As a result, the sampling method itself matters. Eq. (10.47) offers an interesting manner to separate the sampling and reward computation processes: we first use a baseline policy (with θ_{ref}) to sample a number of sequences, and then use the target policy (with θ) to compute the expected reward. In this way, we separate the policy used for collecting the data, and the policy used for computing the gradient. This approach avoids the need to directly sample from the policy we are evaluating, which can be beneficial in cases where generating sequences from the target policy is expensive or difficult. In reinforcement learning, $\mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}}\Big[\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau)\Big]$ is often called a **surrogate objective**.

Eq. (10.47) can also be interpreted from a policy gradient perspective. For $\mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}}\Big[\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau)\Big]$, the gradient at $\theta = \theta_{\text{ref}}$ is given by

$$\frac{\partial}{\partial\theta}\mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}}\Big[\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau)\Big]\Big|_{\theta=\theta_{\text{ref}}} = \mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}}\Big[\frac{\partial \Pr_\theta(\tau)|_{\theta=\theta_{\text{ref}}}}{\partial\theta} R(\tau)\Big] \tag{10.48}$$

The right-hand side is a standard form used in policy gradient methods, meaning that we compute the direction of the parameter update at the point $\theta = \theta_{\text{ref}}$ on the optimization surface.

In practice, this penalty can be approximated by considering only the policy probabilities and ignoring the dynamics. This gives

$$\text{Penalty} \quad = \quad \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) - \sum_{t=1}^{T} \log \pi_{\theta_{\text{ref}}}(a_t|s_t) \tag{10.52}$$

By including this penalty in the optimization objective, we encourage the current policy to remain close to the reference policy, limiting very large updates that could destabilize learning.

We can incorporate this penalty into the clipped surrogate objective function, and obtain

$$U_{\text{ppo-clip}}(\tau;\theta) \quad = \quad U_{\text{clip}}(\tau;\theta) - \beta\text{Penalty} \tag{10.53}$$

where β is the weight of the penalty. This training method is called **proximal policy optimization** (**PPO**), which is one of the most popular reinforcement learning methods used in LLMs and many other fields [Schulman et al., 2017].

Now we can write the objective of training LLMs in the form of PPO.

$$U(\mathbf{x},\mathbf{y};\theta) \quad = \quad U_{\text{ppo-clip}}(\mathbf{x},\mathbf{y};\theta) - \beta\text{Penalty} \tag{10.54}$$

where

$$U_{\text{ppo-clip}}(\mathbf{x},\mathbf{y};\theta) \quad = \quad \sum_{t=1}^{T} \text{Clip}\Big(\frac{\pi_\theta(y_t|\mathbf{x},\mathbf{y}_{<t})}{\pi_{\theta_{\text{ref}}}(y_t|\mathbf{x},\mathbf{y}_{<t})}\Big) A(\mathbf{x},\mathbf{y}_{<t},y_t) \tag{10.55}$$

$$\begin{aligned} \text{Penalty} \quad &= \quad \log \Pr_\theta(\mathbf{y}|\mathbf{x}) - \log \Pr_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \\ &= \quad \sum_{t=1}^{T} \log \Pr_\theta(y_t|\mathbf{x},\mathbf{y}_{<t}) - \sum_{t=1}^{T} \log \Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x},\mathbf{y}_{<t}) \end{aligned} \tag{10.56}$$

Although the notation here appears a bit tedious, the idea of PPO is simple: we develop an objective by combining the clipped likelihood ratio of the target and reference policies with an advantage function, and then impose a penalty that ensures policy updates are not too large. The PPO-based RLHF is illustrated in Figure 10.9.

To summarize, implementing RLHF requires building four models, all based on the Transformer decoder architecture.

- **Reward Model** ($r_\phi(\cdot)$ 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 a Transformer decoder followed by a linear layer that maps a sequence (the concatenation of the input and output) to a real-valued reward score.
- **Value Model** or **Value Function** ($V_\omega(\cdot)$ where ω denotes the parameters). The value function receives reward scores from the reward model and is trained to predict the expected sum of rewards that can be obtained starting from a state. It is generally based on the same architecture as the reward model.

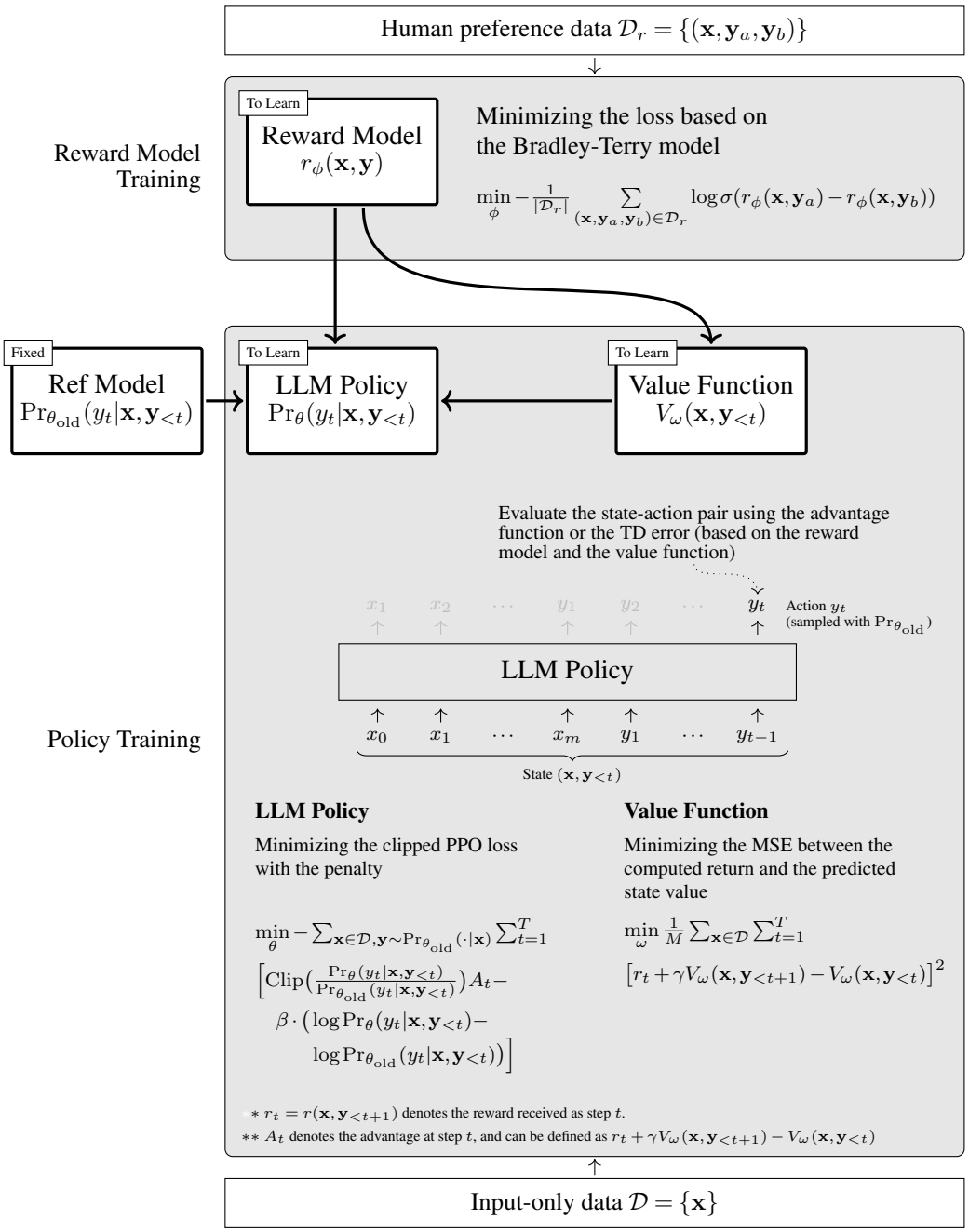


Figure 10.9: Illustration of RLHF. The first step is to collect human preference data and train the reward model using this data. Once the reward model is optimized, along with the reference model, we proceed to train both the policy and the value function. At each prediction step, we compute the sum of the PPO-based loss and update the parameters of the policy. This requires access to the reward model, the reference model, and the value function at hand. At the same time, we update the parameters of the value function by minimizing the MSE loss.

- **Reference Model** ($\pi_{\theta_{\text{ref}}}(\cdot) = \Pr_{\theta_{\text{ref}}}(\cdot)$ where θ_{ref} denotes the parameters). The reference model is the baseline LLM that serves as a starting point for policy training. In RLHF, it represents the previous version of the model or a model trained without human feedback. It is used to perform sampling over the space of outputs and contribute to the loss computation for policy training.
- **Target Model** or **Policy** ($\pi_\theta(\cdot) = \Pr_\theta(\cdot)$ where θ denotes the parameters). This policy governs how the LLM decides the most appropriate next token given its context. 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, while the reference model and the target model can be initialized with a model that has been instruction fine-tuned. 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 model. At each position in an output token sequence, we update the value model by minimizing the MSE error of value prediction, and the policy is updated by minimizing the PPO loss.

10.4 Improved Human Preference Alignment

In the previous section, we reviewed the basic concepts of reinforcement learning and the general framework of RLHF. In this section, we will discuss some refinements of RLHF and alternative methods to achieve human preference alignment.

10.4.1 Better Reward Modeling

In Section 10.3.2, we highlighted the task of learning from human preferences as well as the use of pairwise ranking loss for training reward models. Here we consider more methods for reward modeling. Our discussion will be relatively general, and since the reward model is widely used in many reinforcement learning problems, it will be easy for us to apply the methods discussed here to RLHF and related applications.

1. Supervision Signals

The training of reward models can broadly be seen as a ranking problem, where the model learns to assign scores to outputs so that their order reflects the preferences indicated by humans. There are several methods to train a reward model from the perspective of ranking.

One approach is to extend pairwise ranking to listwise ranking. For each sample in a dataset, we can use the LLM to generate multiple outputs, and ask human experts to order these outputs. For example, given a set of four outputs $\{\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4\}$, one possible order of them can be $\mathbf{y}_2 \succ \mathbf{y}_3 \succ \mathbf{y}_1 \succ \mathbf{y}_4$. A very simple method to model the ordering of the list is to accumulate the pairwise comparison loss. For example, we can define the listwise loss by

accumulating the loss over all pairs of outputs:

$$\mathcal{L}_{\text{list}} = -\mathbb{E}_{(\mathbf{x},Y)\sim\mathcal{D}_r}\Big[\frac{1}{N(N-1)}\sum_{\substack{\mathbf{y}_a\in Y,\mathbf{y}_b\in Y\\ \mathbf{y}_a\neq\mathbf{y}_b}}\log\Pr(\mathbf{y}_a\succ\mathbf{y}_b|\mathbf{x})\Big] \tag{10.57}$$

where Y is a list of outputs, and N is the number of outputs in the list. $\Pr(\mathbf{y}_a\succ\mathbf{y}_b|\mathbf{x})$ can be defined using the Bradley-Terry model, that is, $\Pr(\mathbf{y}_a\succ\mathbf{y}_b|\mathbf{x})=\text{Sigmoid}(r(\mathbf{x},\mathbf{y}_a)-r(\mathbf{x},\mathbf{y}_b))$. Here we omit the ϕ superscript on the $\Pr(\cdot)$ to keep the notation uncluttered.

An extension to the Bradley-Terry model for listwise ranking could involve a ranking mechanism that takes into account the entire list of outputs rather than just pairwise comparisons. One such model is the **Plackett-Luce model**, which generalizes the Bradley-Terry model to handle multiple items in a ranking [Plackett, 1975]. In the Plackett-Luce model, for each item in a list, we define a "worth" for this item that reflects its relative strength of being chosen over other items. For the reward modeling problem here, the worth of $\mathbf{y}$ in the list Y can be defined as

$$\alpha(\mathbf{y}) = \exp(r(\mathbf{x},\mathbf{y})) \tag{10.58}$$

Then the probability of selecting $\mathbf{y}$ from Y is given by

$$\begin{aligned}\Pr(\mathbf{y}\text{ is selected}|\mathbf{x},Y) &= \frac{\alpha(\mathbf{y})}{\sum_{\mathbf{y}'\in Y}\alpha(\mathbf{y}')}\\ &= \frac{\exp(r(\mathbf{x},\mathbf{y}))}{\sum_{\mathbf{y}'\in Y}\exp(r(\mathbf{x},\mathbf{y}'))}\end{aligned} \tag{10.59}$$

Suppose $\mathring{Y}$ is an ordered list $\mathbf{y}_{j_1}\succ\mathbf{y}_{j_2}\succ\cdots\succ\mathbf{y}_{j_N}$. The overall log-probability of this ordered list can be defined as the sum of the conditional log-probabilities at each stage of selection, given by

$$\begin{aligned}\log\Pr(\mathring{Y}|\mathbf{x}) &= \log\Pr(\mathbf{y}_{j_1}\succ\mathbf{y}_{j_2}\succ\cdots\succ\mathbf{y}_{j_N}|\mathbf{x})\\ &= \log\Pr(\mathbf{y}_{j_1}|\mathbf{x},\{\mathbf{y}_{j_1},\mathbf{y}_{j_2},...,\mathbf{y}_{j_N}\})+\\ &\quad \log\Pr(\mathbf{y}_{j_2}|\mathbf{x},\{\mathbf{y}_{j_2},...,\mathbf{y}_{j_N}\})+\\ &\quad \cdots+\\ &\quad \log\Pr(\mathbf{y}_{j_N}|\mathbf{x},\{\mathbf{y}_{j_N}\})\\ &= \sum_{k=1}^{N}\log\Pr(\mathbf{y}_{j_k}|\mathbf{x},\mathring{Y}_{\geq k})\end{aligned} \tag{10.60}$$

where $\mathring{Y}_{\geq k}$ represents the subset of the list of outputs that remain unselected at the k-th stage, i.e., $\mathring{Y}_{\geq k}=\{\mathbf{y}_{j_k},...,\mathbf{y}_{j_N}\}$. Given the log-probability $\log\Pr(\mathring{Y}|\mathbf{x})$, we can define the loss function based on the Plackett-Luce model by

$$\mathcal{L}_{\text{pl}} = -\mathbb{E}_{(\mathbf{x},\mathring{Y})\sim\mathcal{D}_r}\big[\log\Pr(\mathring{Y}|\mathbf{x})\big] \tag{10.61}$$

There are also many other pairwise and listwise methods for modeling rankings, such as RankNet [Burges et al., 2005] and ListNet [Cao et al., 2007]. All these methods can be categorized into a large family of learning-to-rank approaches, and most of them are applicable to the problem of modeling human preferences. However, discussing these methods is beyond the scope of this chapter. Interested readers can refer to books on this topic for more details [Liu, 2009; Li, 2011].

In addition to pairwise and listwise ranking, using pointwise methods to train reward models offers an alternative way to capture human preferences. Unlike methods that focus on the relative rankings between different outputs, pointwise methods treat each output independently. For example, human experts might assign a score to an individual output, such as a rating on a five-point scale. The objective is to adjust the reward model so that its outputs align with these scores. A simple way to achieve pointwise training is through regression techniques where the reward of each output is treated as a target variable. Let $\varphi(\mathbf{x}, \mathbf{y})$ be the score assigned to $\mathbf{y}$ given $\mathbf{x}$ by humans. Pointwise reward models can be trained by minimizing a loss function, often based on mean squared error or other regression losses, between the predicted reward $r(\mathbf{x}, \mathbf{y})$ and the actual human feedback $\varphi(\mathbf{x}, \mathbf{y})$. For example, the loss function could be

$$\mathcal{L}_{\text{point}} = -\mathbb{E}\big[\varphi(\mathbf{x}, \mathbf{y}) - r(\mathbf{x}, \mathbf{y})\big]^2 \tag{10.62}$$

While pointwise methods are conceptually simpler and can directly guide the reward model to predict scores, they might not always be the best choice in RLHF. A problem is that these methods may struggle with high variance in human feedback, especially when different experts provide inconsistent scores for similar outputs. Because they focus on fitting to absolute scores rather than relative differences, inconsistencies in scoring can lead to poor model performance. Moreover, fitting to specific scored outputs might discourage generalization, particularly given that training data is often very limited in RLHF. In contrast, methods that consider relative preferences can promote the learning of more generalized patterns of success and failure. Nevertheless, there are scenarios where pointwise methods might still be suitable. For example, in tasks where training data is abundant and the costs of obtaining accurate, consistent annotations are low, pointwise methods can prove effective.

In fact, to make the supervision signal for training the reward model more robust, we can also introduce additional regularization terms into training. For example, if we consider the first term $U_{\text{ppo-clip}}(\mathbf{x}, \mathbf{y}; \theta)$ in Eq. (10.54) as a type of generalized reward, then the second term (i.e., the penalty term) can be viewed as a form of regularization for the reward model, except that here the goal is to train the policy rather than the reward model. Another example is that Eisenstein et al. [2023] develop a regularization term based on the squared sum of rewards, and add it to the pairwise comparison loss in RLHF:

$$\begin{aligned}
\mathcal{L}_{\text{reg}} &= \mathcal{L}_{\text{pair}} + (-\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r}\big[r(\mathbf{x}, \mathbf{y}_a) + r(\mathbf{x}, \mathbf{y}_b)\big]^2) \\
&= -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r}\big[\log \Pr_\phi(\mathbf{y}_a \succ \mathbf{y}_b | \mathbf{x})\big] \\
&\quad -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b) \sim \mathcal{D}_r}\big[r(\mathbf{x}, \mathbf{y}_a) + r(\mathbf{x}, \mathbf{y}_b)\big]^2
\end{aligned} \tag{10.63}$$

Optimizing with this regularization term can help mitigate the underdetermination of reward models[5].

2. Sparse Rewards vs. Dense Rewards

As discussed in Section 10.3, the rewards in RLHF are very sparse: they are observed only at the end of sequences, rather than continuously throughout the generation process. Dealing with sparse rewards has long been a concern in reinforcement learning, and has been one of the challenges in many practical applications. For example, in robotics, it 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. Also, one can adopt curriculum learning to sequentially structure tasks in a way that the complexity gradually increases. This can help models to master simpler tasks first, which prepares them for more complex challenges as their skills develop. There are many such 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 reinforcement learning, such as Sutton and Barto [2018]'s book.

Although we do not discuss methods for mitigating sparse rewards in detail here, an interesting question arises: why are sparse rewards so successful in RLHF? Recall from Section 10.3.1 that the supervision signal received at each time step t is not the reward for the current action, but rather some form of the accumulated rewards from t until the last time step. Such supervision signals are dense over the sequence, because the reward obtained at the end of the sequence can be transferred back to that time step, regardless of which time step it is. In other words, the sparse rewards are transformed into the dense supervision signals. Furthermore, from the perspective of reward shaping, Ng et al. [1999] show that the reward at t can be defined as

$$r'(s_t, a_t, s_{t+1}) \quad = \quad r(s_t, a_t, s_{t+1}) + f(s_t, a_t, s_{t+1}) \tag{10.64}$$

where $r'(\cdot)$ is the transformed reward function, $r(\cdot)$ is the original reward function, and $f(\cdot)$ is the shaping reward function. To ensure the optimality of the policy under the transformed reward function, the shaping reward function can be given in the form

$$f(s_t, a_t, s_{t+1}) \quad = \quad \gamma\Phi(s_{t+1}) - \Phi(s_t) \tag{10.65}$$

where $\Phi(s)$ is called the potential value of the state s. If we define $\Phi(s)$ as the common value function as in Eq. (10.15) and substitute Eq. (10.65) into Eq. (10.64), we obtain

$$r'(s_t, a_t, s_{t+1}) \quad = \quad r(s_t, a_t, s_{t+1}) + \gamma V(s_{t+1}) - V(s_t) \tag{10.66}$$

[5] A model is called underdetermined if there are multiple alternative sets of parameters that can achieve the same objective.

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.

On the other hand, one of the reasons for adopting end-of-sequence rewards lies in the nature of the RLHF tasks. Unlike traditional reinforcement learning environments where the agent interacts with a dynamic environment, RLHF tasks often involve complex decision-making based on linguistic or other high-level cognitive processes. These processes do not lend themselves easily to frequent and meaningful intermediate rewards because the quality and appropriateness of the actions can only be fully evaluated after observing their impact in the larger context of the entire sequence or task. In this case, the reward signals based on human feedback, though very sparse, are typically very informative and accurate. Consequently, this sparsity, together with the high informativeness and accuracy of the human feedback, can make the learning both robust and efficient.

3. Fine-grained Rewards

For many applications, our objective will be more complex than merely evaluating an entire text. For example, in sentiment analysis, we often do not just determine the sentiment of a text, but need to analyze the sentiment in more detail by associating it with specific aspects of a topic discussed in the text. Consider the sentence "The camera of the phone is excellent, but the battery life is disappointing." In this example, we would need to separately analyze the sentiments expressed about the camera and the battery. Such analysis, known as aspect-based sentiment analysis, helps provide a finer-grained understanding of the customer review compared to general sentiment analysis.

For the problem of reward modeling, we often need to model different parts of a sequence as well. A simple and straightforward way to do this is to divide a sequence into different segments and then compute the reward for each segment [Wu et al., 2023]. Suppose that an output token sequence $\mathbf{y}$ can be divided into n_s segments $\{\bar{\mathbf{y}}_1, ..., \bar{\mathbf{y}}_{n_s}\}$ by some criterion. We can use the reward model to evaluate each of these segments. By taking $\mathbf{x}$, $\mathbf{y}$ and $\bar{\mathbf{y}}_k$ as input to the reward model, the reward score for the k-th segment is given by

$$r^k = r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) \tag{10.67}$$

Then the reward score for the entire output sequence is given by

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) \tag{10.68}$$

Here $r(\mathbf{x}, \mathbf{y})$ can be used to train the policy as usual.

A problem with this model is that training reward models at the segment level is not as straightforward as learning from human preferences on entire texts, as it is difficult to obtain segment-level human preference data. For rating-like problems (e.g., we rate a segment according to its level of misinformation), one simple approach is to assign a rating score to each segment and train the reward model using pointwise methods. For example, we can use a

strong LLM to rate the sequences $\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_{k-1}$ and $\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_k$, and obtain the scores $s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_{k-1})$ and $s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_k)$. We can then define the score of the segment $\bar{\mathbf{y}}_k$ as the difference between $s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_k)$ and $s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_{k-1})$

$$s(\bar{\mathbf{y}}_k) = s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_k) - s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_{k-1}) \tag{10.69}$$

Using these segment-level scores, we can train the reward model with a regression loss function

$$\mathcal{L}_{\text{rating}} = -\mathbb{E}_{\bar{\mathbf{y}}_k}\left[s(\bar{\mathbf{y}}_k) - r(\mathbf{x},\mathbf{y},\bar{\mathbf{y}}_k)\right]^2 \tag{10.70}$$

Sometimes, alignment can be treated as a classification problem, for example, we assess whether a segment has ethical issues. In this case, the segment can be labeled as ethical or unethical, either by humans or using additional classifiers. Given the label of the segment, we can train the reward model using some classification loss function. For example, suppose that $r(\mathbf{x},\mathbf{y},\bar{\mathbf{y}}_k) = 1$ if the segment is classified as unethical, and $r(\mathbf{x},\mathbf{y},\bar{\mathbf{y}}_k) = -1$ otherwise[6]. The hinge loss of training binary classification models is given by

$$\mathcal{L}_{\text{hinge}} = \max(0, 1 - r(\mathbf{x},\mathbf{y},\bar{\mathbf{y}}_k)\cdot\hat{r}) \tag{10.71}$$

where $\hat{r} \in \{1,-1\}$ denotes the ground truth label.

The remaining issue here is how to split $\mathbf{y}$ into segments. One approach is to define a fixed-length segmentation, where $\mathbf{y}$ is divided into equal-length chunks. However, this may not always be ideal, as the content of the sequence may not align well with fixed boundaries. An alternative approach is to segment $\mathbf{y}$ based on specific linguistic or semantic cues, such as sentence boundaries, topic shifts, or other meaningful structures in the text. Such a segmentation can be achieved by using linguistic segmentation systems or prompting LLMs to identify natural breaks in the sequence. Another approach is to use dynamic segmentation methods based on the complexity of the sequence. For example, segments could be defined where there is a significant change in the reward score, which might correspond to shifts in the task being modeled.

4. Combination of Reward Models

A reward model can be viewed as a proxy for the environment. Since the true environment is often too complex or unknown, developing a perfect proxy for the environment is generally not possible. As a result, over-aligning LLMs with this imperfect proxy might lead to decreased performance, known as the **overoptimization problem** [Stiennon et al., 2020; Gao et al., 2023][7]. We can also explain this through Goodhart's law, which states: *when a measure*

[6] To allow the reward model to output categories, we can replace the linear layer described in Section 10.3.2 with a Softmax layer.

[7] This problem is also called **reward hacking** or **reward gaming** [Krakovna et al., 2020; Skalse et al., 2022; Pan et al., 2022], which refers to the phenomenon where the agent attempts to trick the reward model but fails to align its actions with the true intended objectives of the task. Imagine a student who is assigned homework and is rewarded with points or praise for completing it. The student might then find ways to finish the homework

becomes a target, it ceases to be a good measure [Goodhart, 1984].

Addressing the overoptimization problem is not easy, and there is no mature solution yet. The ideal approach might be to develop an oracle reward model that can perfectly capture the true objectives of the task and prevent the agent from "tricking". However, creating such a model is extremely difficult due to the complexity of the real-world environment, as well as the challenge of defining all the relevant factors that contribute to the desired outcome. Instead, a more practical approach is to combine multiple reward models, thereby alleviating the misalignment between the training objective and the true objective that arises from using a single, specific reward model [Coste et al., 2024].

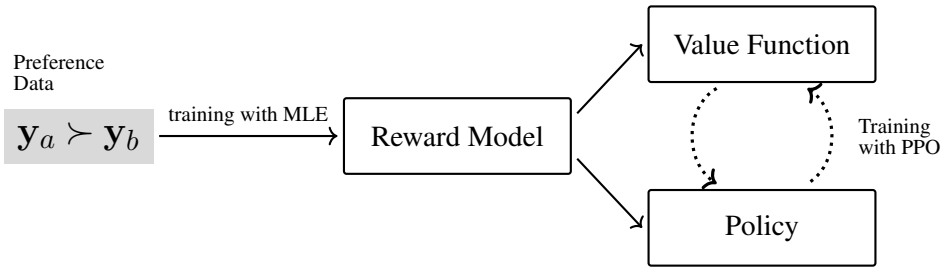
Given a set of reward models, combining them is straightforward, and in some cases, we can simply treat this problem as an ensemble learning problem. A simple yet common approach is to average the outputs of these models to obtain a more precise reward estimation:

$$r_{\text{combine}} = \frac{1}{K}\sum_{k=1}^{K} w_k \cdot r_k(\mathbf{x}, \mathbf{y}) \tag{10.72}$$

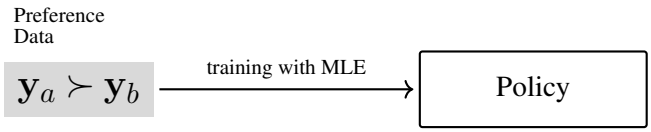
where $r_k(\cdot)$ is the k-th reward model in the ensemble, w_k is the weight of $r_k(\cdot)$, and K is the number of reward models. This combined reward can then be used to supervise the training of a policy. In fact, there are many ways to combine different models, for example, one can make predictions 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 [Miettinen, 1999; Bishop, 2006].

In addition to model combination methods, another important issue is how to collect or construct multiple different reward models. One of the simplest approaches is to employ ensemble learning techniques, such as developing diverse reward models from different subsets of a given dataset or from various data sources. For RLHF, it is also possible to construct reward models based on considerations of different aspects of alignment. For example, we can develop a reward model to evaluate the factual accuracy of the output and another reward model to evaluate the completeness of the output. These two models are complementary to each other, and can be combined to improve the overall evaluation of the output. Another approach is to employ different off-the-shelf LLMs as reward models. This approach is simple and practical, as there have been a lot of well-developed LLMs and we just need to use them with no or little modification. An interesting issue, though not closely related to the discussion here, arises: can an LLM that aligns with other LLMs outperform those LLMs? Probably not at first glance. In part, this is because the target LLM merely imitates other LLMs based on limited supervision and thus cannot capture well the nuances of the behaviors of these supervisors. However, given the strong generalization ability of LLMs, this approach can, in fact, be quite beneficial. For example, using open-sourced or commercial LLMs as reward

with minimal effort to maximize the reward, such as copying and pasting solutions from the internet or previous assignments, rather than solving the problems themselves.



(a) RLHF (PPO)



(b) DPO

Figure 10.10: Standard RLHF (PPO) vs. DPO. In RLHF, the human preference data is used to train a reward model, which is then employed in training the policy as well as 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.

models has demonstrated strong performance in aligning LLMs, even achieving state-of-the-art results on several popular tasks [Lambert et al., 2024].

10.4.2 Direct Preference Optimization

Although learning reward models is a standard step in reinforcement learning, 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 alignment method, called **direct preference optimization** (**DPO**), which simplifies the training framework by eliminating the need to explicitly model rewards [Rafailov et al., 2024]. This method directly optimizes the policy 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 10.10 shows a comparison of the standard RLHF method and the DPO method.

Before deriving the DPO objective, let us first review the objective of policy training used in RLHF. As discussed in Section 10.3.3, the policy is typically trained by optimizing a loss function with a penalty term. The DPO method assumes a simple loss function where the quality of the output $\mathbf{y}$ given the input $\mathbf{x}$ is evaluated by the reward model $r(\mathbf{x}, \mathbf{y})$. The training

objective is thus given by

$$\tilde{\theta} = \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\big[\underbrace{-r(\mathbf{x},\mathbf{y})}_{\text{loss}} + \beta \underbrace{(\log \pi_\theta(\mathbf{y}|\mathbf{x}) - \log \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}))}_{\text{penalty}}\big] \quad (10.73)$$

Note that in this optimization problem, only the term $\pi_\theta(\mathbf{y}|\mathbf{x})$ depends on the target policy $\pi_\theta(\cdot)$. Both the reward model $r(\mathbf{x},\mathbf{y})$ and the reference model $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})$ are assumed to be fixed given $\mathbf{x}$ and $\mathbf{y}$. This is a strong assumption compared with PPO, but as will be shown later, it simplifies the problem and crucial for deriving the DPO objective.

Since θ is the variable we want to optimize, we rearrange the right-hand side of Eq. (10.73) to isolate $\pi_\theta(\mathbf{y}|\mathbf{x})$ as an independent term:

$$\begin{aligned}
\tilde{\theta} &= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\big[\beta\log\pi_\theta(\mathbf{y}|\mathbf{x}) - \beta\log\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) - r(\mathbf{x},\mathbf{y})\big] \\
&= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\big[\log\pi_\theta(\mathbf{y}|\mathbf{x}) - \big(\log\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) + \frac{1}{\beta}r(\mathbf{x},\mathbf{y})\big)\big] \\
&= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\big[\underbrace{\log\pi_\theta(\mathbf{y}|\mathbf{x})}_{\text{dependent on }\theta} - \underbrace{\log\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp\big(\frac{1}{\beta}r(\mathbf{x},\mathbf{y})\big)}_{\text{not dependent on }\theta}\big] \quad (10.74)
\end{aligned}$$

This equation defines the objective function as the difference between the log-probability distribution function of y and another function of y. This form of the objective function seems not "ideal", as we usually prefer to see the difference between two distributions, so that we can interpret this difference as some kind of divergence between the distributions. A simple idea is to convert the second term (i.e., $\log\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp(\frac{1}{\beta}r(\mathbf{x},\mathbf{y}))$) into a log-probability distribution over the domain of $\mathbf{y}$. If we treat $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp(\frac{1}{\beta}r(\mathbf{x},\mathbf{y}))$ as an unnormalized probability of y, we can convert it into a normalized probability by dividing it by a normalization factor:

$$Z(\mathbf{x}) = \sum_{\mathbf{y}} \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp\big(\frac{1}{\beta}r(\mathbf{x},\mathbf{y})\big) \quad (10.75)$$

Hence we can define a probability distribution by

$$\pi^*(\mathbf{y}|\mathbf{x}) = \frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp\big(\frac{1}{\beta}r(\mathbf{x},\mathbf{y})\big)}{Z(\mathbf{x})} \quad (10.76)$$

We then rewrite Eq. (10.74) as

$$\begin{aligned}
\tilde{\theta} &= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\Big[\log\pi_\theta(\mathbf{y}|\mathbf{x}) - \log\frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp\left(\frac{1}{\beta}r(\mathbf{x},\mathbf{y})\right)}{Z(\mathbf{x})} \\
&\qquad\qquad - \log Z(\mathbf{x})\Big] \\
&= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\Big[\log\pi_\theta(\mathbf{y}|\mathbf{x}) - \log\pi^*(\mathbf{y}|\mathbf{x}) - \log Z(\mathbf{x})\Big] \\
&= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\Big[\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\Big[\log\pi_\theta(\mathbf{y}|\mathbf{x}) - \log\pi^*(\mathbf{y}|\mathbf{x})\Big] \\
&\qquad\qquad - \mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\big[\log Z(\mathbf{x})\big]\Big] \\
&= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\Big[\underbrace{\mathrm{KL}\big(\pi_\theta(\cdot|\mathbf{x}) \,||\, \pi^*(\cdot|\mathbf{x})\big)}_{\text{KL divergence}} - \underbrace{\log Z(\mathbf{x})}_{\text{constant wrt. }\theta}\Big]
\end{aligned} \tag{10.77}$$

Since $\log Z(\mathbf{x})$ is independent of θ, it does not affect the result of the $\arg\min_\theta$ operation, and can be removed from the objective. Now we obtain a new training objective which finds the optimal policy π_θ by minimizing the KL divergence between $\pi_\theta(\cdot|\mathbf{x})$ and $\pi^*(\cdot|\mathbf{x})$

$$\tilde{\theta} = \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\Big[\mathrm{KL}\big(\pi_\theta(\cdot|\mathbf{x}) \,||\, \pi^*(\cdot|\mathbf{x})\big)\Big] \tag{10.78}$$

Clearly, the solution to this optimization problem is given by

$$\begin{aligned}
\pi_\theta(\mathbf{y}|\mathbf{x}) &= \pi^*(\mathbf{y}|\mathbf{x}) \\
&= \frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp\left(\frac{1}{\beta}r(\mathbf{x},\mathbf{y})\right)}{Z(\mathbf{x})}
\end{aligned} \tag{10.79}$$

Given this equation, we can express the reward $r(\mathbf{x},\mathbf{y})$ using the target model $\pi_\theta(\mathbf{y}|\mathbf{x})$, the reference model $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})$, and the normalization factor $Z(\mathbf{x})$:

$$r(\mathbf{x},\mathbf{y}) = \beta\left(\log\frac{\pi_\theta(\mathbf{y}|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})} + \log Z(\mathbf{x})\right) \tag{10.80}$$

This is interesting because we initially seek to learn the policy $\pi_\theta(\cdot)$ using the reward model $r(\mathbf{x},\mathbf{y})$, but eventually obtain a representation of the reward model based on the policy. Given the reward model defined in Eq. (10.80), we can apply it to the Bradley-Terry model to

calculate the preference probability (also see Section 10.3.2):

$$\begin{aligned}\Pr_\theta(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x}) &= \text{Sigmoid}(r(\mathbf{x},\mathbf{y}_a) - r(\mathbf{x},\mathbf{y}_b)) \\ &= \text{Sigmoid}\Big(\beta\Big(\log\frac{\pi_\theta(\mathbf{y}_a|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_a|\mathbf{x})} + \log Z(\mathbf{x})\Big) - \\ &\qquad \beta\Big(\log\frac{\pi_\theta(\mathbf{y}_b|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_b|\mathbf{x})} + \log Z(\mathbf{x})\Big)\Big) \\ &= \text{Sigmoid}\Big(\beta\log\frac{\pi_\theta(\mathbf{y}_a|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_a|\mathbf{x})} - \beta\log\frac{\pi_\theta(\mathbf{y}_b|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_b|\mathbf{x})}\Big) \end{aligned} \tag{10.81}$$

This formula is elegant because it converts the difference in rewards into the difference in ratio functions, and we do not need to calculate the value of $Z(\mathbf{x})$. A direct result is that we no longer need a reward model, but only need the target policy and reference model to calculate the probability of preferences. Finally, we can train the target policy by minimizing the following DPO loss function

$$\mathcal{L}_{\text{dpo}}(\theta) = -\mathbb{E}_{(\mathbf{x},\mathbf{y}_a,\mathbf{y}_b)\sim\mathcal{D}_r}\big[\log\Pr_\theta(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x})\big] \tag{10.82}$$

The form of this loss function is very similar to that used in training reward models in RLHF (see Eq. (10.36)). But it should be noted that the loss function here depends on the parameters of the policy (i.e., θ) rather than the parameters of the reward model (i.e., ϕ).

The main advantage of DPO lies in its simplicity and efficiency. The DPO objective is very straightforward — it directly optimizes for preference-based feedback, rather than relying on separately developed reward models. Moreover, DPO is generally more sample-efficient, as it learns from a fixed dataset without the need for the computationally expensive sampling process used in PPO. This makes DPO a popular method for human preference alignment, especially when developing and applying reward models via reinforcement learning is challenging.

DPO can broadly be viewed as an **offline reinforcement learning** method, where the training data is pre-collected and fixed, and there is no exploration. In contrast, online reinforcement learning methods like PPO, which require exploring new states through interaction with the environment (using the reward model as a proxy), also have their unique advantages. One of the benefits of online reinforcement learning is that it allows the agent to continuously adapt to changes in the environment by learning from real-time feedback. This means that, unlike offline methods, online methods are not constrained by the static nature of pre-collected data and can discover new problem-solving strategies. In addition, exploration can help the agent cover a wider range of state-action pairs, thus improving generalization. This could be an important advantage for LLMs, as generalization is considered a critical aspect in applying such large models.

10.4.3 Automatic Preference Data Generation

Although learning from human preferences is an effective and popular method for aligning LLMs, annotating preference data is costly. Using human feedback does not only faces the problem of limited scalability, but it may also introduce bias because human feedback is

inherently subjective. As a result, one can turn to AI feedback methods to address these scalability and consistency issues without the limitations associated with human annotators.

As with data generation for instruction fine-tuning, generating preference data using LLMs is straightforward. 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 consumer service responses.

> Consider a customer service scenario where a customer poses a request. You will review two responses to this request. Please indicate which response is preferred. Note that a good response should be courteous, clear, and concise. It should address the customer's concern directly, provide helpful information or a solution, and maintain a positive tone.
>
> Request:
>
> *Hello, I noticed that my order hasn't arrived yet, though it was scheduled to arrive several days ago. Could you please update me on its status? Thank you!*
>
> Response A:
>
> *I'm very sorry for the delay and understand how disappointing this can be. We're doing our best to sort this out quickly for you.*
>
> Response B:
>
> *Hey, stuff happens! Your package will get there when it gets there, no need to stress.*
>
> Response 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, we can consider demonstrating a few examples or using advanced prompting techniques, such as CoT, to improve labeling performance. For example, we can include in the prompt an example showing how and why one of the two responses is preferred based on a CoT rationale.

In addition to preference labels, we can also obtain the probability associated with each label [Lee et al., 2023]. A simple method is to extract the probabilities for the label tokens, such as "A" and "B", from the probabilities output by the LLM. We can then use the Softmax function or other normalization techniques to re-normalize these probabilities into a distribution over the labels. These probabilities of preferred labels can serve as pointwise supervision signals for training the reward model, as discussed in Section 10.4.1.

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 [Cui et al., 2024]. Dubois et al. [2024] 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 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.

10.4.4 Step-by-step Alignment

So far, our discussion of alignment has primarily focused on the use of reward models for evaluating entire input-output sequence pairs. These methods can be easily adapted to 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 result 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.

In Chapter 9, we studied CoT methods to prompt LLMs to explicitly write out intermediate steps or the reasoning process needed to reach a conclusion or solve a problem. We saw that breaking down a problem into smaller parts could make it easier to understand the solution path and increase the accuracy of the output. These methods can be naturally extended to the alignment of LLMs, that is, we supervise the model during the intermediate steps of reasoning. Consider a reasoning task where an LLM produces a sequence of reasoning steps $\mathbf{y} = \{\bar{\mathbf{y}}_1, ..., \bar{\mathbf{y}}_{n_s}\}$ for the given input. The result of the reasoning is assumed to be included in the last step $\bar{\mathbf{y}}_{n_s}$, and can be easily verified. For this reasoning problem, Uesato et al. [2022] categorize LLM fine-tuning approaches into two classes:

- **Outcome-based Approaches**. Supervision occurs only when the end result is verified. This is a standard method for learning from human feedback we have discussed in this chapter. For example, the LLM is optimized to maximize some form of the reward $r(\mathbf{x}, \mathbf{y})$.
- **Process-based Approaches**. Supervision is involved in all intermediate steps in addition to the last step. To do this, we need to develop a model to give a supervision signal at each step, and develop loss functions that can make use of such supervision signals.

Question (from The International Junior Math Olympiad):
Jessica has a lot of apps on her phone. The number of apps she has is divisible by 5. If she downloaded 10 more apps, she would still have less than 50 apps. What is the maximum original number of apps Jessica could have?

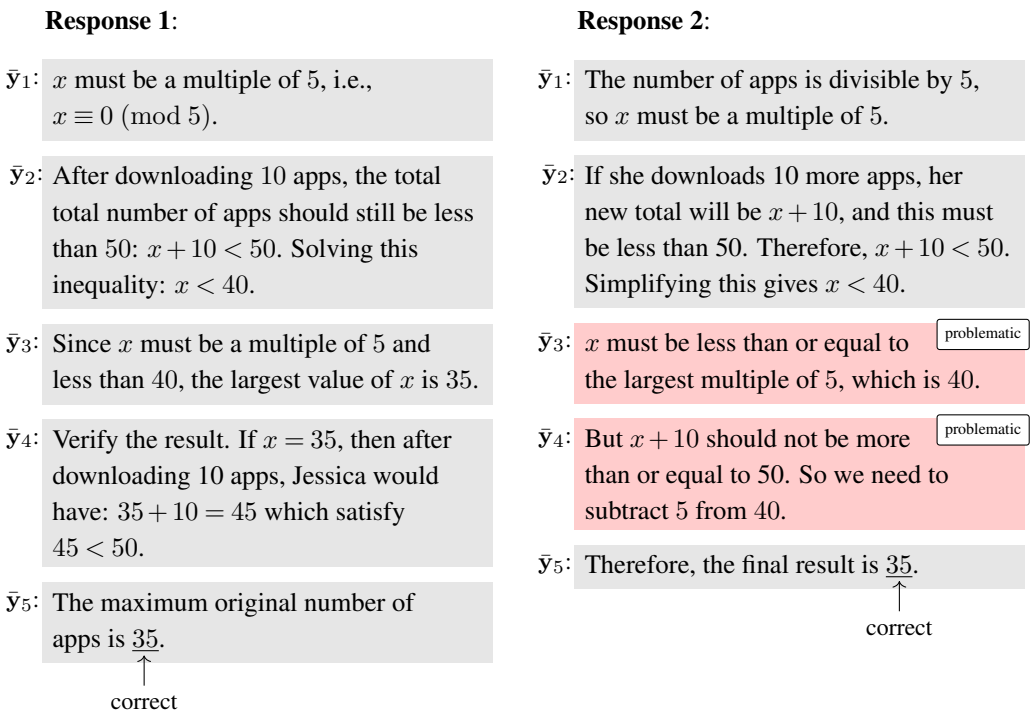


Figure 10.11: Two LLM responses to a math problem. In response 1, both the final result and all the reasoning steps are correct. In response 2, the final result is correct, but there are mistakes in the reasoning process (highlighted in red). For outcome-based approaches, both responses are considered correct. For process-based approaches, the mistakes in response 2 can be considered in reward modeling.

Figure 10.11 shows two LLM outputs for an example math problem. Although the LLM gives the correct final answer in both cases, it makes mistakes during the problem-solving process in the second output. Outcome-based approaches overlook these mistakes and give positive feedback for the entire solution. By contrast, process-based approaches can take these mistakes into account and provide additional guidance on the detailed reasoning steps.

An important issue for process-based approaches is that we need to get step-level feedback during a (potentially) long reasoning path. We can collect or generate reasoning paths corresponding to problems from existing datasets. Human experts then annotate each step in these paths for correctness. These annotations can be used to directly train LLMs or as rewards in reward modeling. However, in practice, richer annotations are often introduced [Lightman et al., 2024]. 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. Furthermore, to improve the efficiency of data annotation, techniques such as active learning can be employed. Identifying obvious errors usually does not significantly contribute to learning from reasoning mistakes. Instead, annotating steps that the model confidently considers correct but are actually problematic is often more effective.

Given a set of step-level annotated reasoning paths and corresponding inputs, we can train a reward model to provide feedback for supervising policy learning. The reward model can be treated as a classification model, and so its architecture can be a Transformer decoder with a Softmax layer stacked on top. At step k, the reward model takes both the problem description (denoted by $\mathbf{x}$) and the reasoning steps generated so far (denoted by $\bar{\mathbf{y}}_{\leq k}$) as input and outputs a probability distribution over the label set $\{\textit{correct}, \textit{incorrect}\}$ or $\{\textit{correct}, \textit{incorrect}, \textit{neutral}\}$. Then the learned reward model is used to evaluate reasoning paths by assessing the correctness of each step. A simple method to model correctness is to count the number of steps that are classified as *correct*, given by

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} \delta(\textit{correct}, C(\mathbf{x}, \bar{\mathbf{y}}_{\leq k})) \tag{10.83}$$

where $C(\mathbf{x}, \bar{\mathbf{y}}_{\leq k})$ denotes the label with the maximum probability. We can also use log-probabilities of classification to define the reward of the entire path

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} \log \Pr(\textit{correct} | \mathbf{x}, \bar{\mathbf{y}}_{\leq k}) \tag{10.84}$$

where $\Pr(\textit{correct} | \mathbf{x}, \bar{\mathbf{y}}_{\leq k})$ denotes the probability of the *correct* label generated by the reward model. The reward score $r(\mathbf{x}, \mathbf{y})$ can then be used to train the policy in RLHF as usual.

While we restrict our discussion to math problems, the approaches described here are general and can be applied to a wide variety of tasks that involve multi-step reasoning and decision-making. Moreover, we can consider various aspects when assessing the quality of a step, rather than just its correctness. For example, in dialogue systems, responses must not only be accurate but also contextually appropriate across multiple turns of conversation. If a model provides a correct response but fails to maintain coherence in the context of the ongoing dialogue, step-level feedback could help the model identify and correct such discrepancies. Also note that the process-based approaches are related to the fine-grained reward modeling approaches discussed in Section 10.4.1. All these approaches essentially aim to provide more detailed supervision to LLMs by breaking their outputs into smaller, more manageable steps. However, process-based feedback focuses more on evaluating the correctness of a step based on its preceding steps, while the approaches in Section 10.4.1 emphasize evaluating each step independently.

The idea of aligning LLMs step by step has great application potential, especially considering the recent shift towards more complex reasoning tasks in the use of LLMs. For example, both the GPT-o1 and GPT-o3 models are designed with more advanced reasoning techniques (such as long internal CoT) to solve challenging problems like scientific and mathematical

reasoning [OpenAI, 2024]. These tasks often rely on long and complex reasoning paths, and therefore, it seems essential to introduce detailed supervision signals in the reasoning process. Moreover, from a practical perspective, effective supervision on long reasoning paths not only improves reasoning performance, but it also helps the model eliminate redundant or unnecessary reasoning steps, thereby reducing reasoning complexity and improving efficiency.

10.4.5 Inference-time Alignment

In this section we explored a variety of methods to align models with human preferences and annotations. However, one of the significant limitations of many such methods is that LLMs must be fine-tuned. For RLHF and its variants, training LLMs with reward models can be computationally expensive and unstable, leading to increased complexity and costs when applying these approaches. In this case, we can consider aligning models at inference time, thus avoiding the additional complexity and effort involved.

One simple way to achieve inference-time alignment is to use the reward model to select the best one from N alternative outputs generated by the LLM, a method known as **Best-of-N sampling** (**BoN sampling**). We can consider BoN sampling as a form of reranking. In fact, reranking methods have been widely used in NLP tasks, such as machine translation, for a long time. They are typically applied in situations where training complex models is costly. In such cases, directly reranking the outputs allows for the incorporation of these complex models at a very low cost[8].

In the BoN sampling process, the LLM takes the input sequence $\mathbf{x}$ and generates N different output sequences $\{\hat{\mathbf{y}}_1, ..., \hat{\mathbf{y}}_N\}$:

$$\{\hat{\mathbf{y}}_1, ..., \hat{\mathbf{y}}_N\} = \underset{\mathbf{y}}{\text{argTopN}}\left[\Pr(\mathbf{y}|\mathbf{x})\right] \tag{10.85}$$

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

$$\hat{\mathbf{y}}_{\text{best}} = \max\{r(\mathbf{x}, \hat{\mathbf{y}}_1), ..., r(\mathbf{x}, \hat{\mathbf{y}}_N)\} \tag{10.86}$$

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 have relatively high quality but be sufficiently different from each other. In many text generation systems, the N-best outputs are very similar, often differing by

[8] Reranking methods can also help us explore what are known as model errors and search errors, although these issues are not often discussed in the context of LLMs. For example, suppose we have an old model and a new, more powerful model. We can use the new model to select the best output from the N-best list of the old model as the oracle output. The performance difference between the oracle output and the top-1 output of the original N-best list reflects the performance gain brought by the new model. If the performance gain is significant, we can say that the old model has more model errors. If the gain is small, it may indicate that the issue lies in search errors, as the best candidates were not found.

just one or two words. The diversity issue is even more challenging in LLMs, as the N-best outputs generated by an LLM can be different 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 generated outputs.

BoN sampling can be used for training LLMs as well. 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 compared to RLHF. Many LLMs have adopted rejection sampling for fine-tuning [Nakano et al., 2021; Touvron et al., 2023].

10.5 Summary

In this chapter, we have explored a range of techniques for aligning LLMs. In particular, we have discussed fine-tuning methods that enable LLMs to follow instructions and align them with human preferences. One of the benefits of fine-tuning LLMs is computation efficiency. Unlike pre-training based on large-scale neural network optimization, fine-tuning is a post-training step and so is less computationally expensive. Moreover, it is better suited to address problems that are not easily solved in pre-training, such as human value alignment. The widespread attention to the alignment issue has also led to a surge of research papers on this topic, which has posed challenges in writing this chapter, as it is difficult to cover all the latest techniques. However, we have tried to provide a relatively detailed introduction to the fundamental approaches to alignment, such as instruction fine-tuning and RLHF.

While we have focused on LLM alignment techniques in this chapter, the term *AI alignment* is a wide-ranging concept. It generally refers to the process of ensuring that the behavior of an AI system aligns with human values, goals, and expectations. The idea of AI alignment can be traced back to the early days of AI. A widely cited description of AI alignment comes from an article by the mathematician and computer scientist Norbert Wiener [Wiener, 1960]. The quote is as follows

> *If we use, to achieve our purposes, a mechanical agency with whose operation we cannot efficiently interfere ... we had better be quite sure that the purpose put into the machine is the purpose which we really desire.*

At that time, AI alignment was a distant concern for researchers. But today, it greatly influences the design of various AI systems. For example, in robotics, alignment is critical to ensuring that autonomous robots safely interact with humans and their environments. In autonomous driving, cars must not only follow traffic laws but also make complex, real-time decisions that prioritize human safety, avoid accidents, and navigate ethical dilemmas.

In current AI research, alignment is usually achieved by developing a surrogate objective that is analogous to the real goal and steering the AI system towards this objective. However,

designing the objective of AI alignment is very difficult. One reason is that human values are diverse and often context-dependent, making it difficult to distill them into a single, universally applicable objective function. Also, the complexity of real-world environments, where values and goals often conflict or evolve over time, further complicates alignment efforts. Even if we could define an appropriate objective, AI systems may find unintended ways to achieve it, leading to "misaligned" outcomes that still technically satisfy the objective but in a harmful or counterproductive way.

These challenges have motivated and are motivating AI research towards more aligned systems, either through developing new mechanisms for perceiving the world or more efficient and generalizable methods to adapt these systems to given tasks. More importantly, as AI systems become more powerful and intelligent, especially given that recent advances in LLMs have shown remarkable capabilities in dealing with many challenging problems, the need for AI alignment has become more urgent. Researchers have started to be concerned with AI safety and warn the community that they need to develop and release AI systems with great caution to prevent these systems from being misaligned [Russell, 2019; Bengio et al., 2024].

Bibliography

[Allal et al., 2024] Loubna Ben Allal, Anton Lozhkov, and Daniel van Strien. cosmopedia: how to create large-scale synthetic data for pre-training. https://huggingface.co/blog/cosmopedia, 2024.

[Aschenbrenner, 2024] Leopold Aschenbrenner. Situational awareness: The decade ahead, 2024. URL https://situational-awareness.ai/.

[Bach et al., 2022] Stephen H. Bach, Victor Sanh, Zheng Xin Yong, Albert Webson, Colin Raffel, Nihal V. Nayak, Abheesht Sharma, Taewoon Kim, M. Saiful Bari, Thibault Févry, Zaid Alyafeai, Manan Dey, Andrea Santilli, Zhiqing Sun, Srulik Ben-David, Canwen Xu, Gunjan Chhablani, Han Wang, Jason Alan Fries, Maged Saeed AlShaibani, Shanya Sharma, Urmish Thakker, Khalid Almubarak, Xiangru Tang, Dragomir R. Radev, Mike Tian-Jian Jiang, and Alexander M. Rush. Promptsource: An integrated development environment and repository for natural language prompts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 93–104, 2022.

[Bengio et al., 2024] Yoshua Bengio, Geoffrey Hinton, Andrew Yao, Dawn Song, Pieter Abbeel, Trevor Darrell, Yuval Noah Harari, Ya-Qin Zhang, Lan Xue, Shai Shalev-Shwartz, Gillian K. Hadfield, Jeff Clune, Tegan Maharaj, Frank Hutter, Atilim Gunes Baydin, Sheila A. McIlraith, Qiqi Gao, Ashwin Acharya, David Krueger, Anca Dragan, Philip Torr, Stuart Russell, Daniel Kahneman, Jan Markus Brauner, and Sören Mindermann. Managing extreme ai risks amid rapid progress. *Science*, 384 (6698):842–845, 2024.

[Bishop, 2006] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Bradley and Terry, 1952] Ralph Allan Bradley and Milton E. Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

[Burges et al., 2005] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96, 2005.

[Burns et al., 2023] Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, Ilya Sutskever, and Jeff Wu. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. *arXiv preprint arXiv:2312.09390*, 2023a.

[Burns et al., 2023] Collin Burns, Jan Leike, Leopold Aschenbrenner, Jeffrey Wu, Pavel Izmailov, Leo Gao, Bowen Baker, and Jan Hendrik Kirchner. Weak-to-strong generalization, 2023b. URL https://https://openai.com/index/weak-to-strong-generalization.

[Cao et al., 2007] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on*

Machine learning, pages 129–136, 2007.

[Charniak, 1997] Eugene Charniak. Statistical parsing with a context-free grammar and word statistics. *AAAI/IAAI*, 2005(598-603):18, 1997.

[Chen et al., 2024] Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, and Hongxia Jin. Alpagasus: Training a better alpaca with fewer data. In *The Twelfth International Conference on Learning Representations*, 2024a.

[Chen et al., 2024] Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024b.

[Chiang et al., 2023] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/.

[Christiano et al., 2017] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[Coste et al., 2024] Thomas Coste, Usman Anwar, Robert Kirk, and David Krueger. Reward model ensembles help mitigate overoptimization. In *The Twelfth International Conference on Learning Representations*, 2024.

[Cui et al., 2024] Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Bingxiang He, Wei Zhu, Yuan Ni, Guotong Xie, Ruobing Xie, Yankai Lin, Zhiyuan Liu, and Maosong Sun. ULTRAFEEDBACK: Boosting language models with scaled AI feedback. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pages 9722–9744, 2024.

[Dubois et al., 2024] Yann Dubois, Chen Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy S Liang, and Tatsunori B Hashimoto. Alpacafarm: A simulation framework for methods that learn from human feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[Eisenstein et al., 2023] Jacob Eisenstein, Chirag Nagpal, Alekh Agarwal, Ahmad Beirami, Alex D'Amour, DJ Dvijotham, Adam Fisch, Katherine Heller, Stephen Pfohl, Deepak Ramachandran, and Peter Shaw. Helping or herding? reward model ensembles mitigate but do not eliminate reward hacking. *arXiv preprint arXiv:2312.09244*, 2023.

[Gao et al., 2023] Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, pages 10835–10866. PMLR, 2023.

[Ge et al., 2024] Yuan Ge, Yilun Liu, Chi Hu, Weibin Meng, Shimin Tao, Xiaofeng Zhao, Hongxia Ma, Li Zhang, Boxing Chen, Hao Yang, Bei Li, Tong Xiao, and Jingbo Zhu. Clustering and ranking: Diversity-preserved instruction selection through expert-aligned quality estimation. *arXiv preprint arXiv:2402.18191*, 2024.

[Goodhart, 1984] Charles AE Goodhart. *Problems of monetary management: the UK experience*. Springer, 1984.

[Gunasekar et al., 2023] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes,

Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

[Han et al., 2024] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey. *arXiv preprint arXiv:2403.14608*, 2024.

[Hendrycks et al., 2020] Dan Hendrycks, Xiaoyuan Liu, Eric Wallace, Adam Dziedzic, Rishabh Krishnan, and Dawn Song. Pretrained transformers improve out-of-distribution robustness. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2744–2751, 2020.

[Hewitt, 2024] John Hewitt. Instruction following without instruction tuning, 2024. URL https://nlp.stanford.edu/~johnhew/instruction-following.html.

[Hewitt et al., 2024] John Hewitt, Nelson F Liu, Percy Liang, and Christopher D Manning. Instruction following without instruction tuning. *arXiv preprint arXiv:2409.14254*, 2024.

[Honovich et al., 2023] Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. Unnatural instructions: Tuning language models with (almost) no human labor. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14409–14428, 2023.

[Houlsby et al., 2019] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[Hu et al., 2022] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[Joshi et al., 2017] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, 2017.

[Krakovna et al., 2020] Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: the flip side of ai ingenuity. https://deepmind.google/discover/blog/specification-gaming-the-flip-side-of-ai-ingenuity, 2020.

[Kung and Peng, 2023] Po-Nien Kung and Nanyun Peng. Do models really learn to follow instructions? an empirical study of instruction tuning. *arXiv preprint arXiv:2305.11383*, 2023.

[Lambert et al., 2024] Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. Rewardbench: Evaluating reward models for language modeling. *arXiv preprint arXiv:2403.13787*, 2024.

[Lee et al., 2023] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*,

2023.

[Li, 2011] Hang Li. *Learning to Rank for Information Retrieval and Natural Language Processing*. Online access: Morgan & Claypool Synthesis Collection Five. Morgan & Claypool Publishers, 2011. ISBN 9781608457076.

[Lightman et al., 2024] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024.

[Liu et al., 2024] Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J Liu, and Jialu Liu. Statistical rejection sampling improves preference optimization. In *The Twelfth International Conference on Learning Representations*, 2024.

[Liu, 2009] Tie-Yan Liu. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331, 2009.

[Longpre et al., 2023] Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning*, pages 22631–22648. PMLR, 2023.

[Miettinen, 1999] Kaisa Miettinen. *Nonlinear multiobjective optimization*, volume 12. Springer Science & Business Media, 1999.

[Mishra et al., 2022] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-task generalization via natural language crowdsourcing instructions. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3470–3487, 2022.

[Mnih et al., 2016] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, pages 1928–1937, 2016.

[Nakano et al., 2021] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

[Ng et al., 1999] Andrew Y Ng, Daishi Harada, and Stuart J Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287, 1999.

[OpenAI, 2024] OpenAI. Learning to reason with llms, September 2024. URL https://openai.com/index/learning-to-reason-with-llms/.

[Ouyang et al., 2022] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[Pan et al., 2022] Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspec-

ification: Mapping and mitigating misaligned models. In *International Conference on Learning Representations*, 2022.

[Plackett, 1975] Robin L Plackett. The analysis of permutations. *Journal of the Royal Statistical Society Series C: Applied Statistics*, 24(2):193–202, 1975.

[Radford et al., 2021] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[Rafailov et al., 2024] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.

[Russell, 2019] Stuart Russell. *Human Compatible: Artificial Intelligence and the Problem of Controls*. Viking, 2019.

[Sanh et al., 2022] Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M Rush. Multitask prompted training enables zero-shot task generalization. In *Proceedings of International Conference on Learning Representations*, 2022.

[Schulman et al., 2015] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*, pages 1889–1897, 2015.

[Schulman et al., 2017] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[Sennrich et al., 2016] Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 86–96, 2016.

[Skalse et al., 2022] Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. *Advances in Neural Information Processing Systems*, 35:9460–9471, 2022.

[Stiennon et al., 2020] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.

[Sutton and Barto, 2018] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (2nd ed.)*. The MIT Press, 2018.

[Szepesvári, 2010] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, 2010.

[Taori et al., 2023] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama

model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[Teknium, 2023] Teknium. Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants, 2023. URL https://huggingface.co/datasets/teknium/OpenHermes-2.5.

[Touvron et al., 2023] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[Uesato et al., 2022] Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

[Wang et al., 2024] Chenglong Wang, Hang Zhou, Yimin Hu, Yifu Huo, Bei Li, Tongran Liu, Tong Xiao, and Jingbo Zhu. Esrl: Efficient sampling-based reinforcement learning for sequence generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 19107–19115, 2024.

[Wang et al., 2023] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *Proceedings of The Eleventh International Conference on Learning Representations*, 2023a.

[Wang et al., 2022] Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, Eshaan Pathak, Giannis Karamanolakis, Haizhi Gary Lai, Ishan Purohit, Ishani Mondal, Jacob Anderson, Kirby Kuznia, Krima Doshi, Kuntal Kumar Pal, Maitreya Patel, Mehrad Moradshahi, Mihir Parmar, Mirali Purohit, Neeraj Varshney, Phani Rohitha Kaza, Pulkit Verma, Ravsehaj Singh Puri, Rushang Karia, Savan Doshi, Shailaja Keyur Sampat, Siddhartha Mishra, Sujan Reddy A, Sumanta Patro, Tanay Dixit, and Xudong Shen. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5085–5109, 2022.

[Wang et al., 2023] Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A. Smith, Iz Beltagy, and Hannaneh Hajishirzi. How far can camels go? exploring the state of instruction tuning on open resources. *Advances in Neural Information Processing Systems*, 36:74764–74786, 2023b.

[Wang et al., 2023] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, 2023c.

[Wei et al., 2022] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester,

Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. In *Proceedings of International Conference on Learning Representations*, 2022.

[Wiener, 1960] Norbert Wiener. Some moral and technical consequences of automation: As machines learn they may develop unforeseen strategies at rates that baffle their programmers. *Science*, 131 (3410):1355–1358, 1960.

[Williams, 1992] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[Wu et al., 2023] Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A. Smith, Mari Ostendorf, and Hannaneh Hajishirzi. Fine-grained human feedback gives better rewards for language model training. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[Xia et al., 2024] Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. Less: Selecting influential data for targeted instruction tuning. *arXiv preprint arXiv:2402.04333*, 2024.

[Xu et al., 2024] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*, 2024.

[Zhao et al., 2024] Hao Zhao, Maksym Andriushchenko, Francesco Croce, and Nicolas Flammarion. Long is more for alignment: A simple but tough-to-beat baseline for instruction fine-tuning. *arXiv preprint arXiv:2402.04833*, 2024.

[Zhou et al., 2023] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206*, 2023.