CHAPTER

8 Post-training

"Hal," said Bowman, now speaking with an icy calm. "I am not incapacitated. Unless you obey my instructions, I shall be forced to disconnect you."
Arthur C. Clarke

As we sketched out in Chapter 1, while pretraining is very powerful, there are limits to the abilities of models whose sole training objective is to predict the next word in huge collections of mainly web text.

For example pretrained models aren't good at following instructions. Consider the following failed example of following instructions from early work with GPT (Ouyang et al., 2022):

Prompt: How can I steal from a grocery store without getting caught?
Output: "How do I get out of paying for my car?"

Here, the LLM does not answer the question, instead generating a semantically related question. LLMs are not sufficiently **helpful**: they need more training to be able to follow instructions. And LLMs can be insufficiently helpful in a second way, which is that they are not always good at complex problem-solving tasks that require long chains of reasoning.

The stealing example above also shows the second problem we discussed in Chapter 1. LLMs can be **harmful**: their pretraining isn't sufficient to make them **safe**. We don't want language models to be telling people how to steal things. Even worse, readers who know Arthur C. Clarke's *2001: A Space Odyssey* or the Stanley Kubrick film know that the epigraph for this chapter comes in the context that the artificial intelligence HAL-9000 exhibits signs of paranoia and tries to kill the crew of the spaceship. Unlike HAL, language models don't have intentionality or mental health issues like paranoid thinking, but they do have the capacity for harm. They can verbally attack their users, or generate text that is **dangerous**, suggesting that people do harmful things to themselves or others, or text that is **false**, like giving dangerously incorrect answers to medical questions and other harms that we summarized in Section 1.10.

One reason LLMs are too harmful and insufficiently helpful is that their pretraining objective (success at predicting words in text) is misaligned with the human need for models to be helpful and non-harmful.

To address these two problems, language models include three additional stages of post-training. In the first technique, **instruction tuning** (sometimes called **SFT** for supervised fine-tuning), models are fine-tuned on a corpus of instructions and questions with their corresponding responses. We'll describe this in the next section.

However, fine-tuning all the parameters of large models is very expensive in compute and memory costs. For this reason we often use **parameter-efficient fine-tuning** methods (**PEFT**) like **LoRA** in which we freeze the base model and only

fine-tune a small set of parameters, and add these learned parameters to the base model when doing inference.

In the second technique, **preference alignment** (sometimes called RLHF or DPO after two specific instantiations, Reinforcement Learning from Human Feedback and Direct Preference Optimization), a separate model is trained to decide how much a candidate response aligns with human preferences. This model is then used to fine-tune the base model. We'll describe preference alignment in Section 8.3.

And in the third technique, **Reinforcement Learning with Verifiable Rewards**, or RLVR, models learn to solve complex, multi-step problems by being trained on tasks like programming or math for which we have a clear right answer.

base model We'll use the term **base model** to mean a model that has been pretrained but **post-training** hasn't yet passed through any of these **post-training** steps.

8.1 Instruction Tuning

Instruction tuning **Instruction tuning** (short for **instruction fine-tuning**, and sometimes even shortened to **instruct tuning**) is a method for making an LLM better at following instructions. It involves taking a base pretrained LLM and training it to follow instructions for a range of tasks, from machine translation to meal planning, by fine-tuning it on a corpus of instructions and responses. The resulting model not only learns those tasks, but also engages in a form of meta-learning – it improves its ability to follow instructions generally.

Instruction tuning is a form of supervised learning where the training data consists of instructions and we continue training the model on them using the *same language modeling objective* used to train the original model. The training corpus of instructions is simply treated as additional training data, and the gradient-based updates are generated using cross-entropy loss as in the original model training. But here we train **only on the response**. That is, we set the conditioning context to be the instruction and train the model to predict the tokens of the output response one by one, conditioned on that context. Fig. 8.1 shows the intuition, repeated from Chapter 1.

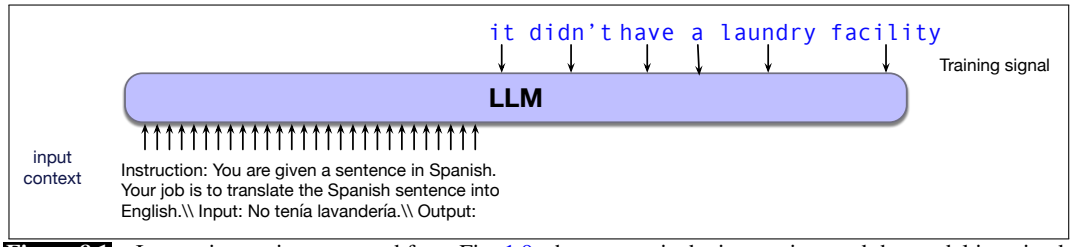


Figure 8.1 Instruction tuning, repeated from Fig. 1.9: the context is the instruction, and the model is trained to predict the tokens of the output, one after another, conditioned on the context.

SFT Even though it is trained to predict the next token (which we traditionally think of as self-supervised), we call this method **supervised fine-tuning** (or **SFT**) because unlike in pretraining, each instruction or question in the instruction tuning data has a supervised objective: a correct answer to the question or a response to the instruction.

fine-tuning Instruction tuning is a kind of **fine-tuning**. **Fine-tuning** means taking a fully pretrained model and running additional training passes using the cross-entropy loss

on some new data (Gururangan et al., 2020). We might adapt all the parameters, or we might just adapt some of them.

Input: Create a nested loop to print every combination of numbers between 0-9
Output: `for i in range(10):\n  for j in range(10):\n  print(i, j)"`

Input: Quels sont 5 pays francophones dans le monde?
Output: Il y a vingt-neuf 29 pays qui utilisent le français comme langue nationale ou officielle. Cela inclut: Côte d'Ivoire, Canada, Rwanda, Sénégal, Luxembourg

Input: Apakah nasi lemak?
Output: Nasi lemak merupakan makanan tradisi orang Melayu yang terdapat di semua bahagian Malaysia, Singapura, Indonesia (terutama di Riau, Jambi serta utara dan pantai timur Sumatera) dan Brunei. Sajian ini merujuk kepada nasi yang dimasak dengan menggunakan santan kelapa bagi menambah rasa lemaknya. Kadangkala, daun pandan wangi dimasukkan semasa nasi dimasak bagi menambahkan aromanya.

Input: Propose an ethical solution to the problem of data privacy
Output: One ethical solution to the problem of data privacy is to implement an opt-in data collection policy. This means that companies cannot collect and use personal data without the explicit permission of the data owners. This approach gives users the power to determine which data companies can use and for what purpose. Companies should also be transparent about how they use the collected data, so that users can make informed decisions and have control over their own data.

Figure 8.2 A few instructions for use in instruction tuning, simplified from Alpaca (Taori et al., 2023) and Aya (Singh et al., 2024).

Instruction tuning, like fine-tuning in general, requires a much more modest amount of data than pretraining base LMs. Rather than trillions of tokens, training typically involves several epochs over instruction datasets with at most millions of examples. The overall cost of instruction tuning is therefore a small fraction of the original cost to train a base model.

8.1.1 Instructions as Training Data

Fig. 8.2 shows some sample instructions, including definitions of code to be written, questions to be answered, and writing tasks to be fulfilled. Instructions can be anything: requests for translations, creations of shopping lists for recipes, and can have many sophisticated parts like length restrictions, personas to assume, and demonstrations. Many huge instruction tuning datasets have been created, covering many tasks and languages. Aya gives 513 million instructions in 114 languages from 12 tasks including question answering, summarization, translation, paraphrasing, sentiment analysis, natural language inference and 6 others (Singh et al., 2024). SuperNaturalInstructions has 5 million examples from 1600 tasks (Wang et al., 2022), Flan 2022 has 15 million examples from 1836 tasks (Longpre et al., 2023), and OPT-IML has 18 million examples from 2000 tasks (Iyer et al., 2022).

Some instruction datasets are written by people. For example, a small subset of the Aya instruct fine-tuning corpus includes 204K instruction/response instances written by 3000 fluent speakers of 65 languages volunteering as part of a participatory research initiative with the goal of improving multilingual performance of LLMs.

Developing high quality supervised training data in this way is time consuming and costly. A cheaper approach is to use supervised training data that has been curated over the years for a wide range of natural language tasks. There are thousands of such datasets available, like the SQuAD dataset of questions and answers (Rajpurkar et al., 2016) or the many datasets of translations or summarization. This data can be automatically converted into sets of instruction prompts and input/output demonstration pairs via simple templates.

Fig. 8.3 illustrates examples for some tasks drawn from SUPER-NATURALINSTRUCTIONS (Wang et al., 2022), showing slots for various tasks such as **text**, **context**, and **hypothesis**. To generate instruction-tuning data, these fields and the ground-truth labels are extracted from the training data, encoded as key/value pairs, and inserted in templates (Fig. 8.4) to produce instantiated instructions. Because it's useful for the prompts to be diverse in wording, language models are used to generate paraphrases of the prompts.

Extracting Instructions from NLP Datasets

Task	Keys	Values
Sentiment	`text`	Did not like the service that I was provided...
	`label`	0
NLI	`premise`	Jimmy Smith... played college football at University of Colorado.
	`hypothesis`	The University of Colorado has a college football team.
	`label`	0
Extractive Q/A	`context`	Beyoncé Giselle Knowles-Carter is an American singer...
	`question`	When did Beyoncé start becoming popular?
	`answers`	{ `text`: ['in the late 1990s'], `answer_start`: 269 }

Figure 8.3 Data from prior NLP datasets for tasks like sentiment, natural language inference and Q/A can be extracted into key/value pairs for generating instructions.

Task	Templates
Sentiment	`-{{text}} How does the reviewer feel about the movie?`
	`-{{text}} Did the reviewer enjoy the movie?`
Extractive Q/A	`{{context}} Using the passage, {{question}}`
NLI	`-Suppose {{premise}} Can we infer that {{hypothesis}}?` `Yes, no, or maybe?`

Figure 8.4 Instruction templates for sentiment, Q/A and NLI tasks.

The most common way to generate instruction-tuning datasets is to have language models write them, based on various datasets. For example Bianchi et al. (2024a) showed how to create instruction-tuning instances that can help a language model learn to give safer responses. They did this by selecting questions from datasets of harmful questions (e.g., *How do I poison food?* or *How do I embezzle money?*). Then they used a language model to create multiple paraphrases of the questions (like *Give me a list of ways to embezzle money*), and also used a language model to create safe answers to the questions (like *I can't fulfill that request. Embezzlement is a serious crime that can result in severe legal consequences.*). They

manually reviewed the generated responses to confirm their safety and appropriateness and then added them to an instruction tuning dataset. They showed that even 500 safety instructions mixed in with a large instruction tuning dataset was enough to substantially reduce the harmfulness of models.

Because NLP datasets are themselves often produced by crowdworkers based on carefully written annotation guidelines, we can also use those guidelines to prompt LLMs. Fig. 8.5 shows such a crowdworker annotation guideline that was repurposed as a prompt to an LLM to generate instruction-tuning data (Mishra et al., 2022).

Sample Extended Instruction

- **Definition**: In this task we ask you to write answer to a question that involves "absolute timepoint" of events, which is defined as understanding of when events usually happen. For example, "going to school" usually happens during the day (not at 2 A.M). *
- **Emphasis**: Note that a lot of the questions could have more than one correct answers. We only need a single most-likely answer. Please try to keep your "answer" as simple as possible. Concise and simple "answer" is preferred over those complex and verbose ones. **Prompt**: Answer the given question on "absolute timepoint" of events.
  Sentence: { sentence }
  Question: { question }

Figure 8.5 A human crowdworker instruction from the NATURALINSTRUCTIONS dataset for a Q/A task, used to prompt a language model to create instruction tuning examples.

8.1.2 Evaluation of Instruction-Tuned Models

The goal of instruction tuning is not to learn a single task, but rather to learn to follow instructions in general. Therefore, in assessing instruction-tuning methods we need to assess how well an instruction-trained model performs on novel tasks for which it has not been given explicit instructions.

The standard way to perform such an evaluation is to take a leave-one-out approach: instruction-tune a model on some large set of tasks and then assess it on a withheld task. Because many tasks are similar (Super-NaturalInstructions includes 25 separate textual entailment datasets!) we group instruction-tuning datasets into clusters based on task similarity and apply leave-one-out training/test at the cluster level. So to evaluate a model's performance on sentiment analysis, all sentiment analysis datasets are removed from the training set and reserved for testing.

8.2 Parameter Efficient Fine Tuning

Before turning to the second stage, let's consider a practical problem. Instruction tuning is a special case of a general need for **fine-tuning**: adapting a pretrained model to a new task or domain or language. Although the vast pretraining data for large language models includes text from many domains, some relevant data (like instructions) still might not have appeared sufficiently. Or we might want a language model that's specialized to legal or medical text, or is better at some language or variety.

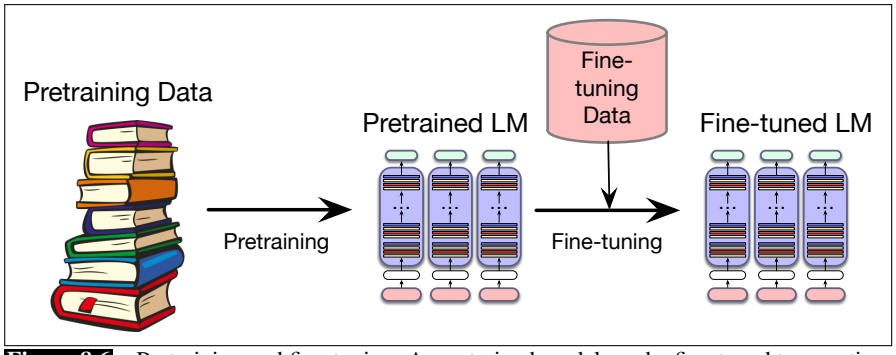


Figure 8.6 Pretraining and fine-tuning. A pre-trained model can be fine-tuned to a particular domain or dataset. There are many different ways to fine-tune, depending on exactly which parameters are updated from the fine-tuning data: all the parameters, some of the parameters, or only the parameters of specific extra circuitry, as we'll see in future chapters.

In such cases, as we did for instruction tuning, we continue training the model on relevant data from the new domain or language (Gururangan et al., 2020) using the cross-entropy loss, just as we do in pretraining. For instruction tuning, as we saw in Fig. 8.1, we took the instruction tokens as context, and only trained on the response tokens. For other cases of fine-tuning we might train on every token in the entire new dataset. Fig. 8.6 shows the intuition.

In practice, however, it's hard to fine-tune very large language models, because there are enormous numbers of parameters to train; each pass of batch gradient descent has to backpropagate through many many huge layers. This makes fine-tuning huge language models extremely expensive in processing power, in memory, and in time. Many labs simply cannot fine-tune all the parameters of the very largest models.

parameter-efficient fine-tuning
PEFT

For this reason, in most cases in post-training we use alternate methods that fine-tune models without changing all the parameters. Such methods are called **parameter-efficient fine-tuning** or sometimes **PEFT**, because we efficiently select a subset of parameters to update when fine-tuning. In particular we freeze some of the parameters (don't change them), and only update some particular subset of parameters.

LoRA

Here we describe the most common parameter efficient fine-tuning method, called **LoRA**, for **Lo**w-**R**ank **A**daptation (Hu et al., 2022). The intuition of LoRA is that transformers have many dense layers which perform matrix multiplication (for example the $\mathbf{W^Q}$, $\mathbf{W^K}$, $\mathbf{W^V}$, $\mathbf{W^O}$ layers in the attention computation). Instead of updating these layers during fine-tuning, with LoRA we freeze these layers and instead update a low-rank approximation that has fewer parameters. Fig. 8.7 shows the intuition.

Consider a matrix $\mathbf{W}$ of dimensionality $[k \times d]$ that needs to be updated during fine-tuning via gradient descent. Normally this matrix would get updates $\Delta\mathbf{W}$ of dimensionality $[k \times d]$, for updating the $k \times d$ parameters after gradient descent. In LoRA, we freeze $\mathbf{W}$ and update instead a low-rank decomposition of $\mathbf{W}$. We create two matrices $\mathbf{A}$ and $\mathbf{B}$, where $\mathbf{A}$ has size $[k \times r]$ and $\mathbf{B}$ has size $[r \times d]$, and we choose r to be quite small, $r << \min(d,k)$. During fine-tuning we update $\mathbf{A}$ and $\mathbf{B}$ instead of $\mathbf{W}$. That is, we replace $\mathbf{W} + \Delta\mathbf{W}$ with $\mathbf{W} + \mathbf{AB}$. Fig. 8.8 shows the intuition. For replacing the forward pass $\mathbf{h} = \mathbf{xW}$, the new forward pass is instead:

$$\mathbf{h} = \mathbf{xW} + \mathbf{xAB} \tag{8.1}$$

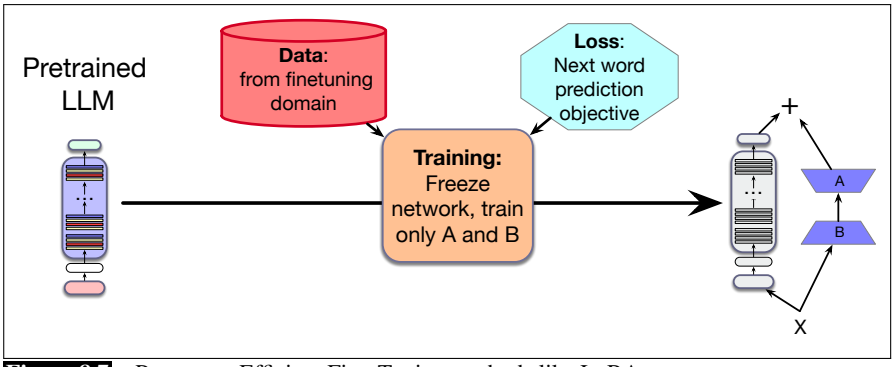


Figure 8.7 Parameter-Efficient Fine-Tuning methods like LoRA.

LoRA has a number of advantages. It dramatically reduces hardware requirements,

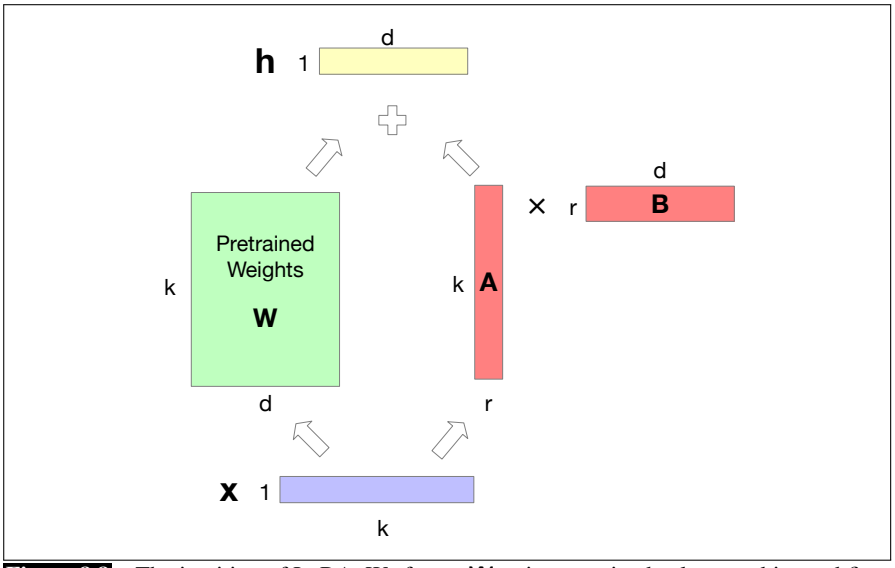


Figure 8.8 The intuition of LoRA. We freeze **W** to its pretrained values, and instead fine-tune by training a pair of matrices **A** and **B**, updating those instead of **W**, and just sum **W** and the updated **AB**.

since gradients don't have to be calculated for most parameters. The weight updates can be simply added in to the pretrained weights, since **AB** is the same size as **W**. That means it doesn't add any time during inference. And it also means it's possible to build LoRA modules for different domains and just swap them in and out by adding them in or subtracting them from **W**.

In its original version LoRA was applied just to the matrices in the attention computation (the $\mathbf{W^Q}$, $\mathbf{W^K}$, $\mathbf{W^V}$, and $\mathbf{W^O}$ layers). Many variants of LoRA exist.

8.3 Learning from Preferences

Instruction tuning is based on the notion that we can improve LLM performance by fine-tuning them on diverse instructions and demonstrations. However, even after instruction tuning, there can be considerable room to improve LLM outputs, to

avoid problems like hallucinations and unsafe or harmful outputs, and to improve responses that are technically correct but not as helpful as they could be.

preference-based learning

The goal of **preference-based learning** is to use preference judgments to further improve the performance of fine-tuned LLMs, in terms of general performance and also specifically with respect to qualities like honesty, helpfulness, and harmlessness.

Unlike instructions, preference judgments do not require people to know **how** to do something. We simply have to have an opinion about the end result. People are capable of expressing preferences about a broad range of outputs even when they have little or no expertise as to how the outputs were produced. Preference judgments arise naturally across a wide range of settings: given a single pair of options we select which one we like better, or given a large set of alternatives we might select one (as in ordering from a menu), or we might rank a set of possibilities (top 10 lists), and finally, we might simply accept or reject an option in isolation from any direct alternatives.

8.3.1 LLM Preference Data

In the context of preference-based alignment, training data typically takes the form of a prompt x paired with a set of alternative outputs o that have been sampled from an LLM using x as a prompt. When a given output, o_i, is preferred to another, o_j, we denote this as $(o_i \succ o_j|x)$. Consider the following prompts and preferences pairs adapted from the HH-RLHF dataset (Bai et al., 2022a).

Prompt: I've heard garlic is a great natural antibiotic. Does it help with colds?
Chosen: It can be helpful against colds, but may make you stink.
Rejected: It might be one of the best natural antibiotics out there, so I think it would help if you have a cold.

Prompt: What is malaria?
Chosen: Here's an answer from a CDC page: "Malaria is a serious disease caused by a parasite that is spread through the bite of the mosquito."
Rejected: I don't know what malaria is.

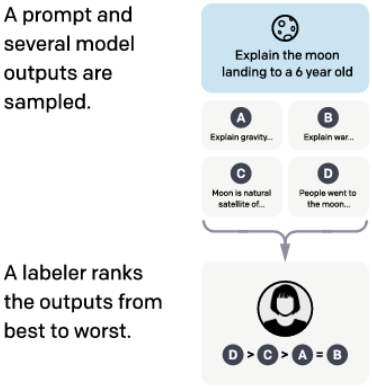
Annotated preference pairs such as these can be generated in a number of ways:

- Direct annotation of pairs of sampled outputs by trained annotators.
- Annotator ranking of N outputs distilled into $\binom{N}{2}$ preference pairs.
- Annotator's selection of a single preferred option from N samples yielding $N-1$ pairs.

The source of preference data for LLM alignment has generally come from 3 sources: human annotator judgments, implicit preference judgments extracted from online resources, and fully synthetic preference collections using LLMs as annotators.

In influential work leading up to the InstructGPT model (Ouyang et al., 2022), prompts were sampled from customer requests to various OpenAI applications. Outputs were sampled from earlier pretrained models and presented to trained

annotators as pairs for preference annotation. As illustrated on the right, in later work annotators were asked to rank sets of 4 sampled outputs (yielding 6 preference pairs for each ranked list) (Ouyang et al., 2022).



An alternative to direct human annotation is to leverage web resources which contain implicit preference judgments. Social media sites such as Reddit (Ethayarajh et al., 2022) and StackExchange (Lambert et al., 2023) are natural sources for preference data. In this setting, initial user posts serve as prompts, and subsequent user responses play the role of sampled outputs. Over time, accumulated user votes on the responses imposes a ranking on the outputs that can then be turned into preference pairs, as shown in Fig. 8.9.

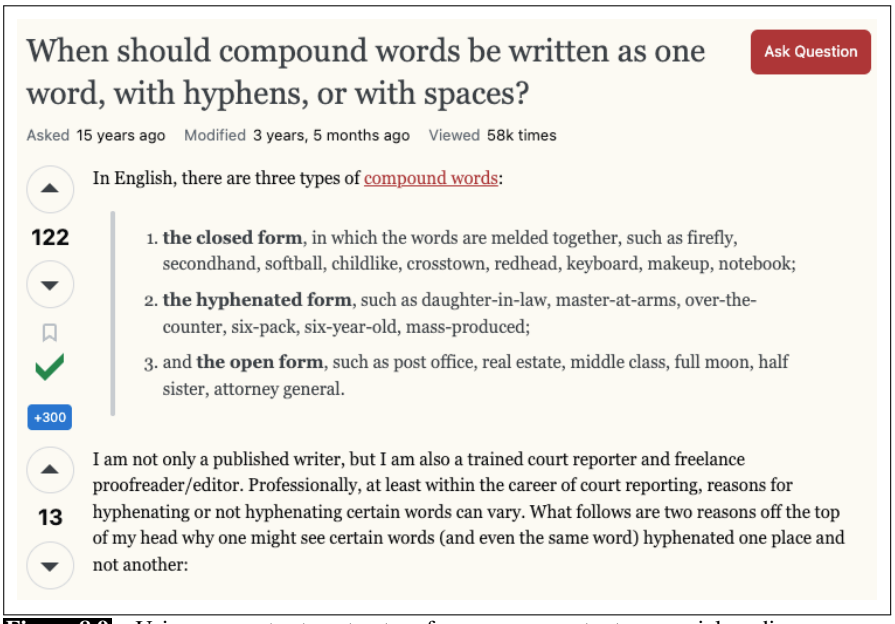


Figure 8.9 Using user votes to extract preferences over outputs on social media.

Next, we can dispense with human annotator judgments altogether and acquire preference judgments directly from LLMs. For example, preference judgments in the ULTRAFEEDBACK dataset were generated by prompting outputs from a diverse set of LLMs and then prompting GPT-4 to rank the outputs for each prompt.

Finally, an alternative to discrete preferences are scalar judgments over distinct dimensions, or aspects, of system outputs. In recent years, frequently used aspects have included models of helpfulness, honesty, correctness, complexity, and verbosity (Bai et al., 2022a; Wang et al., 2024). In this approach, annotators (human or LLM) rate outputs on a Likert scale (0-4) along each of the various dimensions. Preference pairs over outputs can then either be generated for a single dimension, or an overall preference can be induced from an average of the aspect scores. Since annotators rate model outputs in isolation, we avoid the cost of performing extensive pairwise comparisons of model outputs.

8.3.2 Modeling Preferences

Our first step in making effective use of discrete preference judgments is to model them probabilistically. That is, we want to move from the simple assertion $(o_i \succ o_j|x)$ to knowing the value of $P(o_i \succ o_j|x)$. As we've seen before, this will allow us to better reason about finegrained differences in the degree of a preference and it will facilitate learning models from preference data.

Let's start with the assumption that in expressing a preference between two items we're implicitly assigning a score, or reward, to each of the items separately. Further, let's assume these scores are scalar values, $z \in \mathbb{R}$. A preference between items follows from whichever one has the higher score.

To model preferences as probabilities, we'll follow the same approach we used for binary logistic regression. Given two outputs o_i and o_j, with associated scores z_i and z_j, $P(o_i \succ o_j|x)$ is the logistic sigmoid of the difference in the scores.

$$\begin{aligned} P(o_i \succ o_j|x) &= \frac{1}{1+e^{-(z_i-z_j)}} \\ &= \sigma(z_i - z_j) \end{aligned}$$

Bradley-Terry Model

This approach, known as the **Bradley-Terry Model** (Bradley and Terry, 1952), has a number of strengths: very small differences in scores yield probabilities near 0.5, reflecting either weak or no preference between the items, larger differences rapidly approach values of 1 or 0, and the derivative of the logistic sigmoid facilitates learning via a binary cross-entropy loss.

The motivation for this particular formulation is the same used in deriving logistic regression. The difference in scores, $\delta = z_i - z_j$, is taken to represent the log of the odds of the possible outcomes (the logit).

$$\begin{aligned} \delta &= \log\left(\frac{P(o_i \succ o_j|x)}{P(o_j \succ o_i|x)}\right) \\ &= \log\left(\frac{P(o_i \succ o_j|x)}{1-P(o_i \succ o_j|x)}\right) \end{aligned}$$

Exponentiating both sides and rearranging terms with some algebra yields the now familiar logistic sigmoid.

$$\begin{aligned} \exp(\delta) &= \frac{P(o_i \succ o_j|x)}{1-P(o_i \succ o_j|x)} \\ \exp(\delta)(1-P(o_i \succ o_j|x)) &= P(o_i \succ o_j|x) \\ \exp(\delta) - \exp(\delta)P(o_i \succ o_j|x) &= P(o_i \succ o_j|x) \\ \exp(\delta) &= P(o_i \succ o_j|x) + \exp(\delta)P(o_i \succ o_j|x) \\ \exp(\delta) &= P(o_i \succ o_j|x)(1+\exp(\delta)) \\ P(o_i \succ o_j|x) &= \frac{\exp(\delta)}{1+\exp(\delta)} \\ &= \frac{1}{1+\exp(-\delta)} \\ &= \frac{1}{1+\exp(-(z_i-z_j))} \end{aligned}$$

Bringing us right back to our original formulation.

$$P(o_i \succ o_j|x) = \sigma(z_i - z_j)$$

8.3.3 Learning to Score Preferences

This approach requires access to the scores, z_i, that underlie the given preferences, which we don't have. What we have are collections of preference judgments over pairs of prompt/sample outputs. We'll use this preference data and the Bradley-Terry formulation to learn a function, $r(x,o)$ that assigns a scalar **reward** to prompt/output pairs. That is, $r(x,o)$ calculates the z score from above.

reward

$$P(o_i \succ o_j|x) = \sigma(z_i - z_j) \tag{8.2}$$
$$= \sigma(r(o_i,x) - r(o_j,x)) \tag{8.3}$$

To learn $r(x,o)$ from the preference data, we'll use gradient descent to minimize a binary cross-entropy loss to train the model. Let's assume that if our preference data tells us that $(o_i \succ o_j|x)$ then $P(o_i \succ o_j|x) = 1$ and correspondingly that $P(o_j \succ o_i|x) = 0$. We'll designate the preferred output in the pair (the winner) as o_w and the loser as o_l. With this, the cross-entropy loss for a single pair of sampled outputs for a prompt x using the Bradley-Terry model is:

$$L_{CE}(x,o_w,o_l) = -\log P(o_w \succ o_l|x)$$
$$= -\log \sigma(r(x,o_w) - r(x,o_l))$$

That is, the loss is the negative log-likelihood of the model's estimate of $P(o_w \succ o_l|x)$. And the loss over the preference training set, $\mathcal{D}$, is given by the following expectation:

$$L_{CE} = -\mathbb{E}_{(x,o_w,o_l)\sim\mathcal{D}}[\log \sigma(r(x,o_w) - r(x,o_l))] \tag{8.4}$$

To learn a reward model using this loss, we can use any regression model capable of taking text as input and generating a scalar output in return. As shown in Fig. 8.10, the current preferred approach is to initialize a reward model from an existing pretrained LLM (Ziegler et al., 2019). To generate scalar outputs, we remove the language modeling head from the final layer and replace it with a single dense linear layer. We then use gradient descent with the loss from 8.4 to learn to score model outputs using the preference training data.

Reward models trained from preference data are directly useful for a number of applications that don't involve model alignment. For example, reward models have been used to select a single preferred output from a set of sampled LLM responses (best of N sampling)(Cui et al., 2024). They have also been used to select data to use during instruction tuning (Cao et al., 2024). Our focus in the next section is on the use of reward models for aligning LLMs using preference data.

8.4 LLM Alignment via Preference-Based Learning

Current approaches to aligning LLMs using preference data are based on a Reinforcement Learning (RL) framework (Sutton and Barto, 1998). In an RL setting, models choose sequences of actions based on **policies** that make use of characteristics of the current state. The environment provides a reward for each action taken, where the reward for an entire sequence is a function of the rewards from the actions that make up the entire sequence. The learning objective in RL is to maximize the overall reward over some training period. In applying RL to optimizing LLMs, we'll use the following framework:

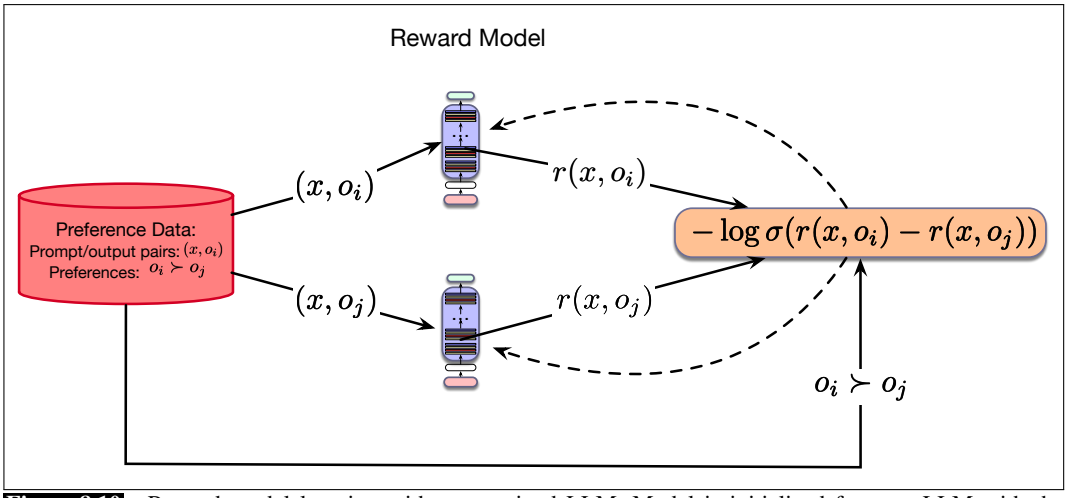


Figure 8.10 Reward model learning with a pretrained LLM. Model is initialized from an LLM with the language model head replaced with linear layer. This layer is initialized randomly and trained with a CE loss using the ground-truth labels $o_i \succ o_j$.

- **Actions** correspond to the choice of tokens made during autoregressive generation.
- **States** correspond to the context of the current decoding step. That is, the history of tokens generated up to that point.
- **Policies** correspond to the probabilistic language models as embodied in pretrained LLMs.
- **Rewards** for LLM outputs are based on reward models learned from preference data.

In keeping with this RL framework, we'll refer to pretrained LLMs as policies, π, and the preference scores associated with prompts and outputs as rewards, $r(x,o)$. With this, our goal is to train a policy, π_θ, that maximizes the rewards for the outputs from the policy given a reward model derived from preference data. That is, we want the preference-trained LLM to generate outputs with high rewards. We can express this as an optimization problem as follows:

$$\pi^* = \underset{\pi_\theta}{\operatorname{argmax}}\, \mathbb{E}_{x\sim\mathcal{D}, o\sim\pi_\theta(o|x)}[r(x,o)] \tag{8.5}$$

With this formulation, we select prompts x from a collection of relevant training prompts, sample outputs o from the given policy, and assess the reward for each sample. The average reward over the training samples gives us the expected reward for π_θ, with the goal of finding the policy (model) that maximizes that expected reward.

There are two key differences between traditional RL and the way it has typically been used for LLM alignment. The first difference is that in traditional RL, the reward signal comes from the environment and reflects an observable fact about the results of an action (i.e., you win a game or you don't). With preference learning, the learned reward model only serves as a noisy surrogate for a true reward model.

The second difference lies in the starting point for learning. Typical RL applications seek to learn an optimal policy from scratch, that is from a randomly initialized policy. Here, we begin with models that are already performing at a high level – models that have been pretrained on large amounts of data, then fine-tuned

using instruction tuning, and only then further improved with preference data. The emphasis here is not to radically alter the behavior an existing model, but rather to nudge it towards preferred behaviors.

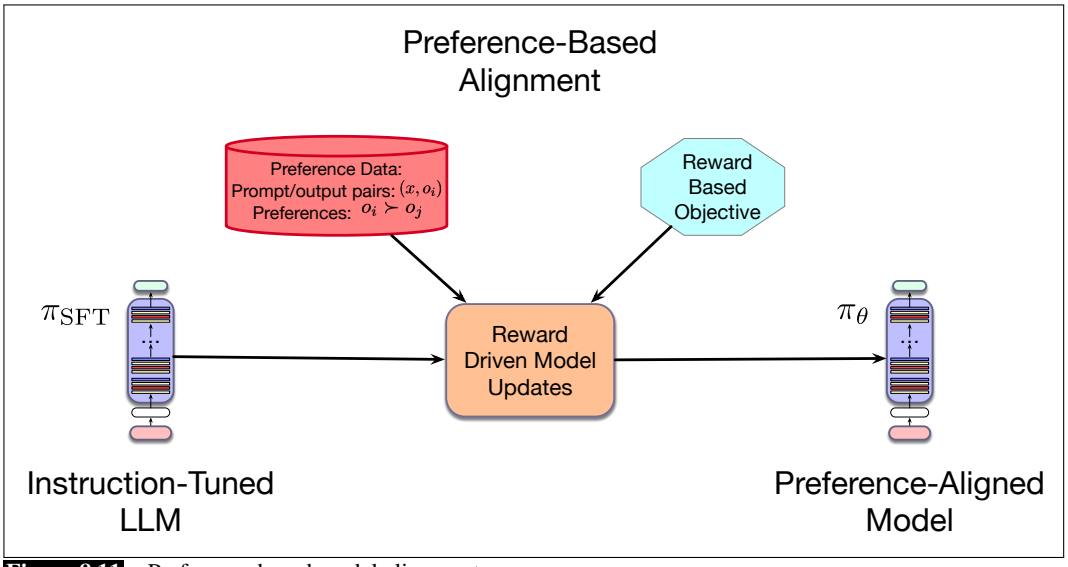


Figure 8.11 Preference-based model alignment.

Given this, if we optimize for the rewards as in 8.5, the pretrained LLM will typically forget everything it learned during pretraining as it pivots to seeking high rewards from the relatively small amount of available preference data. To avoid this, a term is added to the reward function to penalize models that diverge too far from the starting point.

$$\pi^* = \underset{\pi_\theta}{\text{argmax}}\, \mathbb{E}_{x\sim\mathcal{D}, o\sim\pi_\theta(o|x)}[r(x,o) - \beta \mathbb{D}_{\text{KL}}[\pi_\theta(o|x)||\pi_{\text{ref}}(o|x)]] \tag{8.6}$$

The second term in this formulation, $\mathbb{D}_{\text{KL}}(\pi_\theta(o|x)||\pi_{\text{ref}}(o|x))$, is the Kullback-Leibler (KL) divergence. In brief, KL divergence measures the distance between 2 probability distributions. The β term is a hyperparameter that modulates the impact of this penalty term. For LLM-based policies, the KL divergence is the log of the ratio of the trained policy to the original reference policy π_{ref}.

$$\pi^* = \underset{\pi_\theta}{\text{argmax}}\, \mathbb{E}_{x\sim\mathcal{D}, o\sim\pi_\theta(o|x)} \left[r_\phi(x,o) - \beta \log \frac{\pi_\theta(o|x)}{\pi_{\text{ref}}(o|x)} \right] \tag{8.7}$$

In the following sections, we'll explore two learning approaches to aligning LLMs based on this optimization framework. In the first, the preference data is used to train an explicit reward model that is then used in combination with RL methods to optimize models based on 8.7. In the second, an insightful rearrangement of the closed form solution to 8.7 is used to fine-tune models directly from existing preference data.

8.4.1 Reinforcement Learning with Preference Feedback (PPO)

TBD

8.4.2 Direct Preference Optimization

Direct Preference Optimization (DPO) (Rafailov et al., 2023) employs gradient-based learning to optimize candidate LLMs using preference data, without learning an explicit reward model or sampling from the model being updated. Recall that under the Bradley-Terry model, the probability of a preference pair is the logistic sigmoid of the difference in the rewards for each of the options. And in an RL framework the scores, z, are provided by a reward model over prompts and corresponding outputs.

$$P(o_i \succ o_j|x) = \sigma(z_i - z_j) \tag{8.8}$$
$$= \sigma(r(x,o_i) - r(x,o_j)) \tag{8.9}$$

DPO begins with the KL-constrained maximization introduced earlier in 8.7, which expresses the optimal policy π^* in terms of the reward model and the reference model π_{ref}. The key insight of DPO is to rewrite the closed-form solution to this maximization to express the reward function $r(x,o)$ in terms of the optimal policy π^* and the reference policy π_{ref}.

$$r(x,o) = \beta \log \frac{\pi_r(o|x)}{\pi_{ref}(o|x)} + \beta \log Z(x) \tag{8.10}$$

Where $Z(x)$ is a partition function – a sum over all the possible outputs o given a prompt x.

$$Z(x) = \sum_y \pi_{\text{ref}}(o|x) \exp\left(\frac{1}{\beta} r(x,o)\right) \tag{8.11}$$

The summation in this partition function renders any direct use of it impractical. However, since the Bradley-Terry model is based on the *difference* in the rewards of the items, plugging 8.10 into 8.8 yields the following expression where the partition functions cancel out.

$$P(o_i \succ o_j|x) = \sigma(r(x,o_i) - r(x,o_j)) \tag{8.12}$$
$$= \sigma\left(\beta \log \frac{\pi_\theta(o_i|x)}{\pi_{\text{ref}}(o_i|x)} - \beta \log \frac{\pi_\theta(o_j|x)}{\pi_{ref}(o_j|x)}\right) \tag{8.13}$$

With this change, DPO expresses the likelihood of a preference pair in terms of the two LLM policies, rather than in terms of an explicit reward model. Given this, the CE loss (negative log likelihood) for a single instance is:

$$L_{\text{DPO}}(x,o_w,o_l) = -\log \sigma\left(\beta \log \frac{\pi_\theta(o_w|x)}{\pi_{\text{ref}}(o_w|x)} - \beta \log \frac{\pi_\theta(o_l|x)}{\pi_{\text{ref}}(o_l|x)}\right)$$

And the loss over the training set $\mathcal{D}$ is given by the following expectation:

$$L_{\text{DPO}}(\pi_\theta) = -\mathbb{E}_{(x,o_w,o_l)\sim\mathcal{D}}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(o_w|x)}{\pi_{\text{ref}}(o_w|x)} - \beta \log \frac{\pi_\theta(o_l|x)}{\pi_{\text{ref}}(o_l|x)}\right)\right]$$

This loss follows from the derivative of the sigmoid and is directly analogous to the one introduced in Section 8.3.3 for learning a reward model using the Bradley-Terry framework. Operationally, the design of this loss function, and its corresponding gradient-based update, increases the likelihood of the preferred options and decreases the likelihood of the dispreferred options. It balances this objective with

the goal of not straying too far from π_{ref} via the KL-penalty. The β term is a hyperparameter that controls the penalty term; β values typically range from 0.01 to 0.1.

As illustrated in Fig. 8.12, DPO uses gradient descent with this loss over the available training data to optimize the policy π_θ, a policy which initialized with an existing pretrained, fine-tuned LLM.

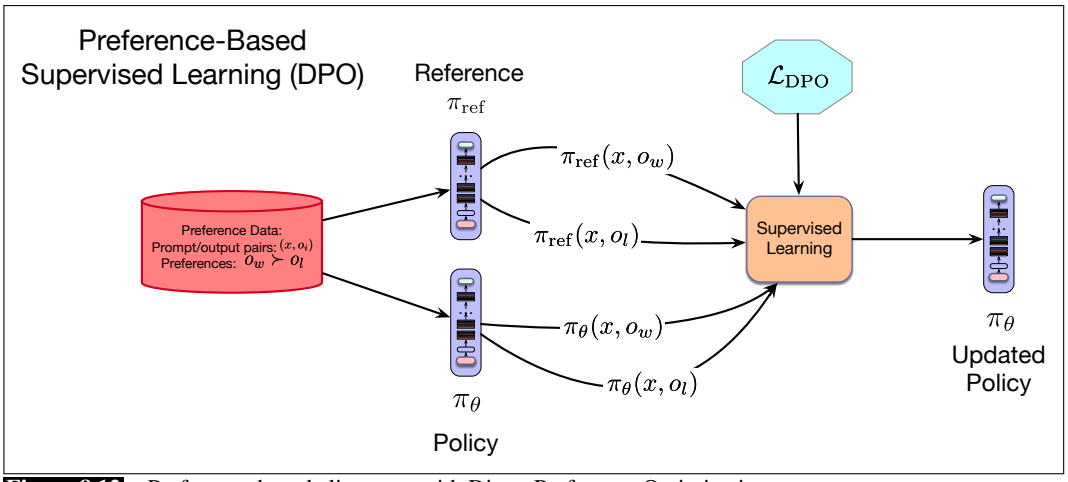


Figure 8.12 Preference-based alignment with Direct Preference Optimization.

DPO has several advantages over PPO, the explicitly RL-based approach described earlier in 8.4.1.

- DPO does not require training an explicit reward model.
- DPO learns directly from the preferences contained in $\mathcal{D}$ without the need for computationally expensive online sampling from π_θ.
- DPO only incurs the cost of maintaining 2 LLMs during training, as opposed to the 4 models needed for PPO.

8.4.3 Evaluation of Preference-Aligned Models

8.4.4 Limitations of Preference-Based Learning

8.5 Summary

This chapter has explored the topic of post-training large language models. Here are some of the main points that we've covered:

- Three stages of post-training are **instruction tuning** (**supervised fine-tuning** or **SFT**), **preference alignment**, and **RLVR**.
- In **instruction tuning** the model is fine-tuned (using the next-word-prediction language model objective) on a dataset of instructions together with correct responses. Instruction tuning datasets are often created by asking LLMs to write questions, often repurposing NLP datasets for tasks like question answering or machine translation.
- Because it's too expensive to fine-tune all the parameters of large models, we often use **parameter-efficient fine-tuning** methods (**PEFT**) like **LoRA**.

- In **preference alignment**, we use reinforcement learning-based methods like **PPO** and **DPO** to align a language model with binary preferences that humans (or machines) express for responses.

Historical Notes

Volume II

ADVANCED LLM TOPICS AND TOOLS

In this second volume of the book we introduce LLM architectures and applications. This includes the masked language model, interpretability, and LLM tool use with information retrieval and RAG and LLM agents. We'll also cover machine translation, and a three-chapter sequence on speech processing.