Chapter 12. Fine-Tuning Generation Models

In this chapter, we will take a pretrained text generation model and go over the process of *fine-tuning* it. This fine-tuning step is key in producing high-quality models and an important tool in our toolbox to adapt a model to a specific desired behavior. Fine-tuning allows us to adapt a model to a specific dataset or domain.

Throughout this chapter, we will guide you among the two most common methods for fine-tuning text generation models, *supervised fine-tuning* and *preference tuning*. We will explore the transformative potential of fine-tuning pretrained text generation models to make them more effective tools for your application.

The Three LLM Training Steps: Pretraining, Supervised Fine-Tuning, and Preference Tuning

There are three common steps that lead to creating a high-quality LLM:

1. *Language modeling*

   The first step in creating a high-quality LLM is to pretrain it on one or more massive text datasets (Figure 12-1). During training, it attempts to predict the next token to accurately learn linguistic and semantic representations found in the text. As we saw before in Chapters 3 and 11, this is called language modeling and is a self-supervised method.

   This produces a *base* model, also commonly referred to as a *pretrained* or *foundation* model. Base models are a key artifact of the training process but are harder for the end user to deal with. This is why the next step is important.

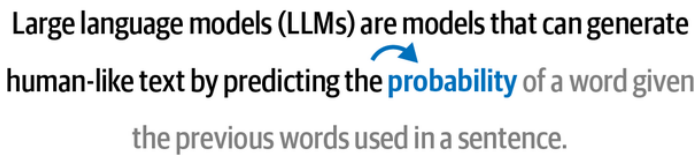

Figure 12-1. During language modeling, the LLM aims to predict the next token based on an input. This is a process without labels.

2. *Fine-tuning 1 (supervised fine-tuning)*

   LLMs are more useful if they respond well to instructions and try to follow them. When humans ask the model to write an article, they expect the model to generate the article and not list other instructions for example (which is what a base model might do).

With supervised fine-tuning (SFT), we can adapt the base model to follow instructions. During this fine-tuning process, the parameters of the base model are updated to be more in line with our target task, like following instructions. Like a pretrained model, it is trained using next-token prediction but instead of only predicting the next token, it does so based on a user input (Figure 12-2).

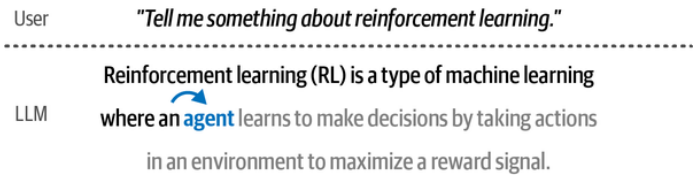

Figure 12-2. During supervised fine-tuning, the LLM aims to predict the next token based on an input that has additional labels. In a sense, the label is the user's input.

SFT can also be used for other tasks, like classification, but is often used to go from a base generative model to an instruction (or chat) generative model.

3. *Fine-tuning 2 (preference tuning)*

   The final step further improves the quality of the model and makes it more aligned with the expected behavior of AI safety or human preferences. This is called preference tuning. Preference tuning is a form of fine-tuning and, as the name implies, aligns the output of the model to our preferences, which are defined by the data that we give it. Like SFT, it can improve upon the original model but has the added benefit of distilling preference of output in its training process. These three steps are illustrated in Figure 12-3 and demonstrate the process of starting from an untrained architecture and ending with a preference-tuned LLM.

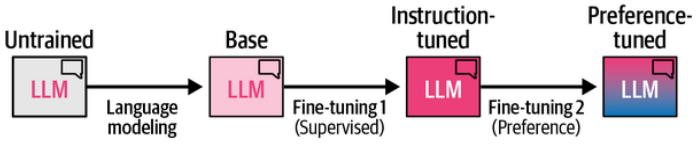

Figure 12-3. The three steps of creating a high-quality LLM.

In this chapter, we use a base model that was already trained on massive datasets and explore how we can fine-tune it using both fine-tuning strategies. For each method, we start with the theoretical underpinnings before using them in practice.

Supervised Fine-Tuning (SFT)

The purpose of pretraining a model on large datasets is that it is able to reproduce language and its meaning. During this process, the model learns to complete input phrases as shown in Figure 12-4.

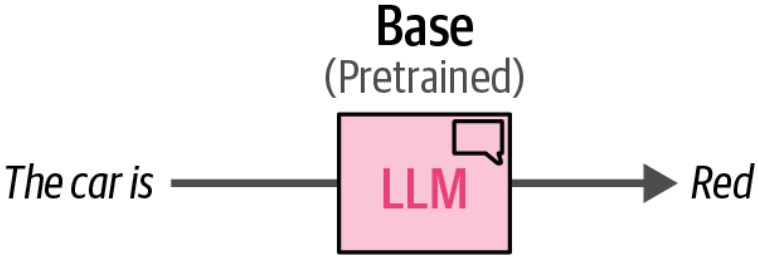

Figure 12-4. A base or pretrained LLM was trained to predict the next word(s).

This example also illustrates that the model was not trained to follow instructions and instead will attempt to complete a question rather than answer it (Figure 12-5).

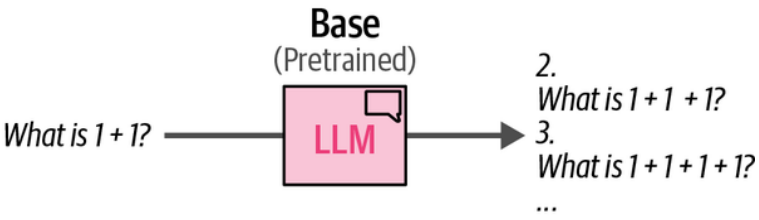

Figure 12-5. A base LLM will not follow instructions but instead attempts to predict each next word. It may even create new questions.

We can use this base model and adapt it to certain use cases, such as following instructions, by fine-tuning it.

Full Fine-Tuning

The most common fine-tuning process is full fine-tuning. Like pretraining an LLM, this process involves updating all parameters of a model to be in line with your target task. The main difference is that we now use a smaller but labeled dataset whereas the pretraining process was done on a large dataset without any labels (Figure 12-6).

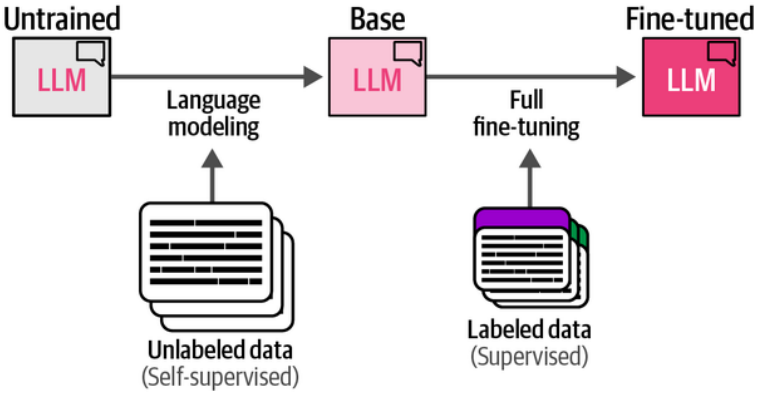

Figure 12-6. Compared to language modeling (pretraining), full fine-tuning uses a smaller but labeled dataset.

You can use any labeled data for full fine-tuning, making it also a great technique for learning domain-specific representations. To make our LLM follow instructions, we will need question-response data. This data, as shown in Figure 12-7, is queries by the user with corresponding answers.

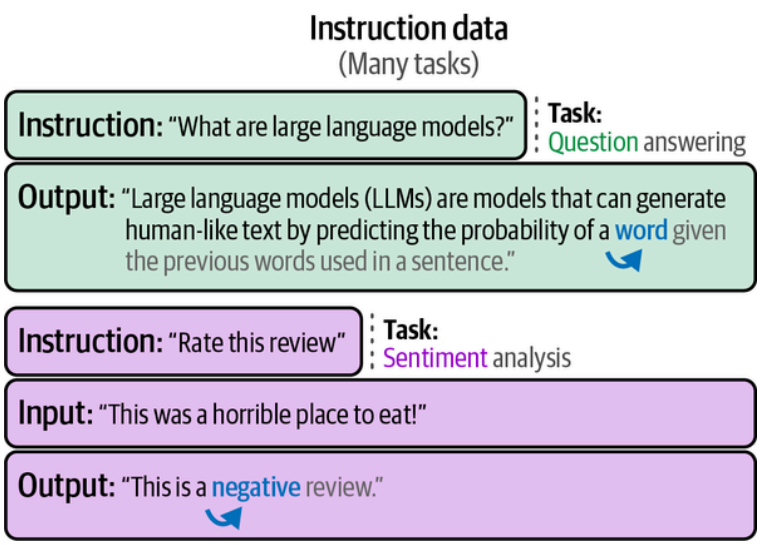

Figure 12-7. Instruction data with instructions by a user and corresponding answers. The instructions can contain many different tasks.

During full fine-tuning, the model takes the input (instructions) and applies next-token prediction on the output (response). In turn, instead of generating new questions, it will follow instructions.

Parameter-Efficient Fine-Tuning (PEFT)

Updating all parameters of a model has a large potential of increasing its performance but comes with several disadvantages. It is costly to train, has slow training times, and requires significant storage. To resolve these issues, attention has been given to parameter-efficient fine-tuning (PEFT) alternatives that focus on fine-tuning pretrained models at higher computational efficiency.

Adapters

Adapters are a core component of many PEFT-based techniques. The method proposes a set of additional modular components inside the Transformer that can be fine-tuned to improve the model's performance on a specific task without having to fine-tune all the model weights. This saves a lot of time and compute.

Adapters are described in the paper "Parameter-efficient transfer learning for NLP", which showed that fine-tuning 3.6% of the parameters of BERT for a task can yield comparable performance to fine-tuning all the model's weights.[1] On the GLUE benchmark, the authors show they reach within 0.4% of the performance of full fine-tuning. In a single Transformer block, the paper's proposed architecture places adapters after the attention layer and the feedforward neural network as illustrated in Figure 12-8.

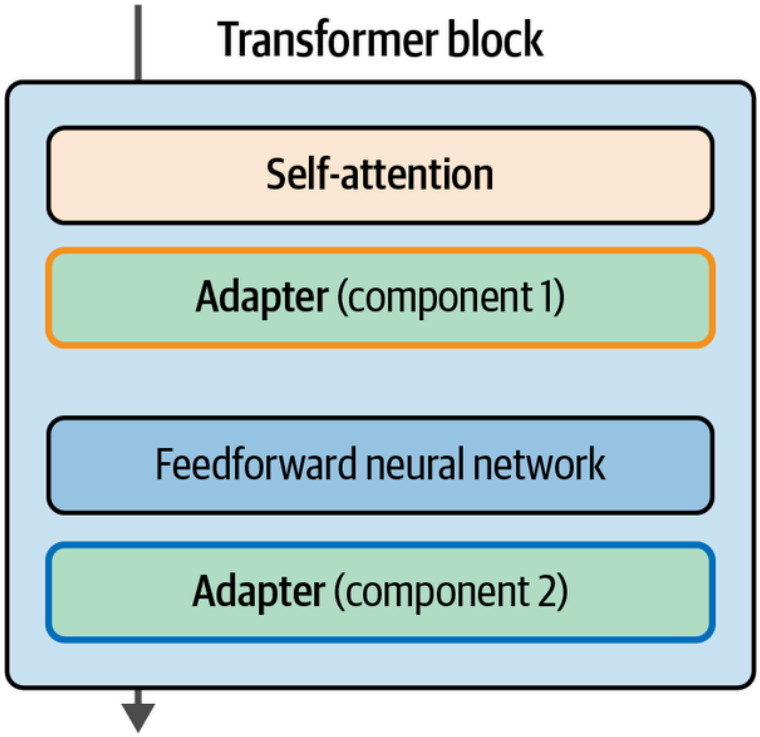

Figure 12-8. Adapters add a small number of weights in certain places in the network that can be fine-tuned efficiently while leaving the majority of model weights frozen.

It's not enough to only alter one Transformer block, however, so these components are part of every block in the model, as Figure 12-9 shows.

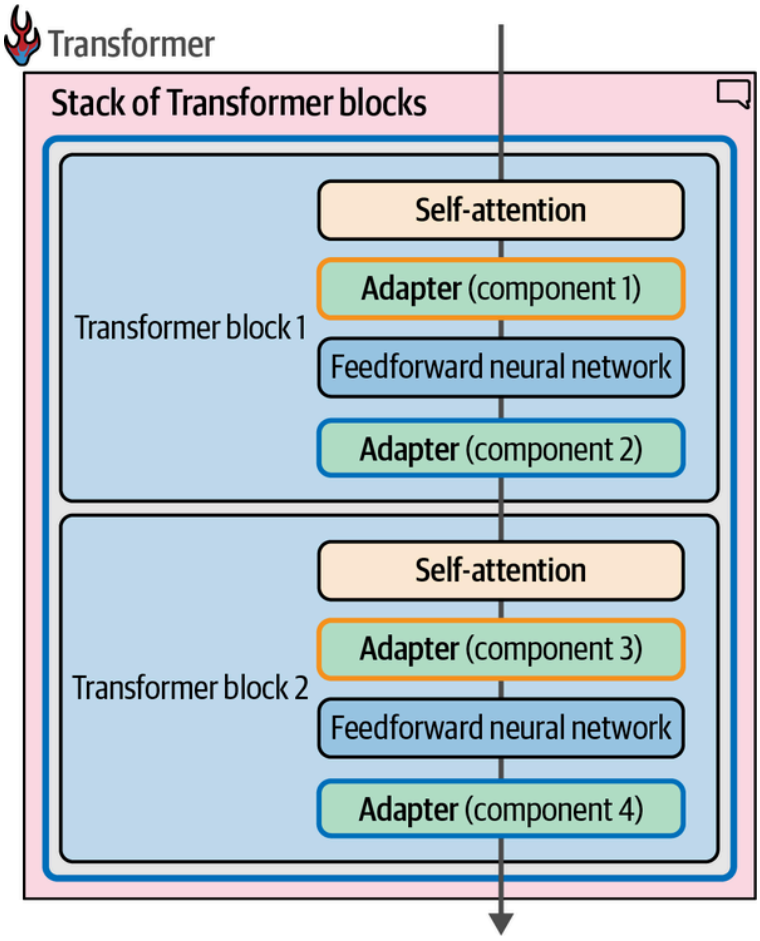

Figure 12-9. Adapter components span the various Transformer blocks in the model.

Seeing all the adapter's components across the model like this enables us to see individual adapters as shown in Figure 12-10, which is a collection of these components spanning all the blocks of the model. Adapter 1 can

be a specialist in, say, medical text classification, while Adapter 2 can specialize in named-entity recognition (NER). You can download specialized adapters from *https://oreil.ly/XraXg*.

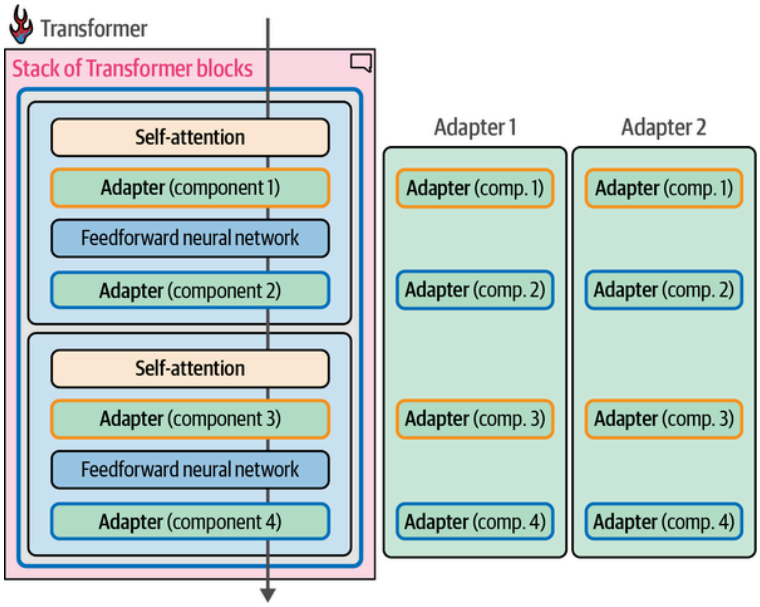

Figure 12-10. Adapters that specialize in specific tasks can be swapped into the same architecture (if they share the same original model architecture and weights).

The paper "AdapterHub: A framework for adapting transformers" introduced the Adapter Hub as a central repository for sharing adapters.[2] A lot of these earlier adapters were more focused on BERT architectures. More recently, the concept has been applied to text generation Transformers in papers like "LLaMA-Adapter: Efficient fine-tuning of language models with zero-init attention".[3]

Low-Rank Adaptation (LoRA)

As an alternative to adapters, low-rank adaptation (LoRA) was introduced and is at the time of writing is a widely used and effective technique for PEFT. LoRA is a technique that (like adapters) only requires updating a small set of parameters. As illustrated in Figure 12-11, it creates a small subset of the base model to fine-tune instead of adding layers to the model.[4]

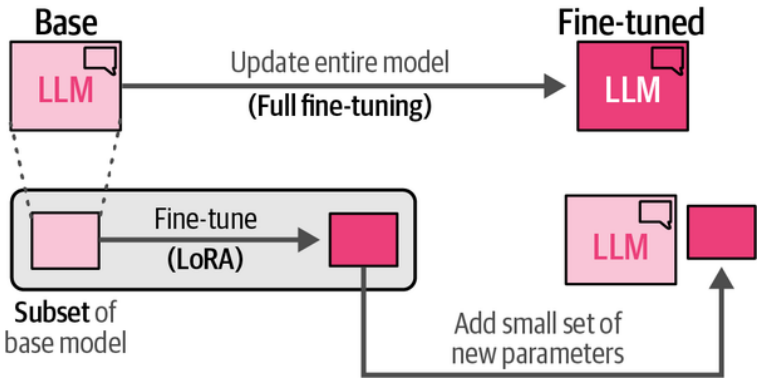

Figure 12-11. LoRA requires only fine-tuning a small set of parameters that can be kept separately from the base LLM.

Like adapters, this subset allows for much quicker fine-tuning since we only need to update a small part of the base model. We create this subset of parameters by approximating large matrices that accompany the original LLM with smaller matrices. We can then use those smaller matrices

as a replacement and fine-tune them instead of the original large matrices. Take for example the 10 × 10 matrix we see in Figure 12-12.

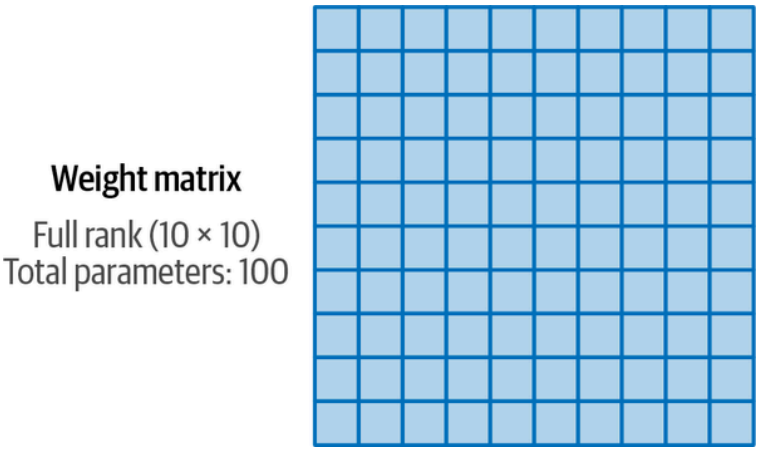

Figure 12-12. A major bottleneck of LLMs is their massive weight matrices. Only one of these may have 150 million parameters and each Transformer block would have its version of these.

We can come up with two smaller matrices, which when multiplied, reconstruct a 10 × 10 matrix. This is a major efficiency win because instead of using 100 weights (10 times 10) we now only have 20 weights (10 plus 10), as we can see in Figure 12-13.

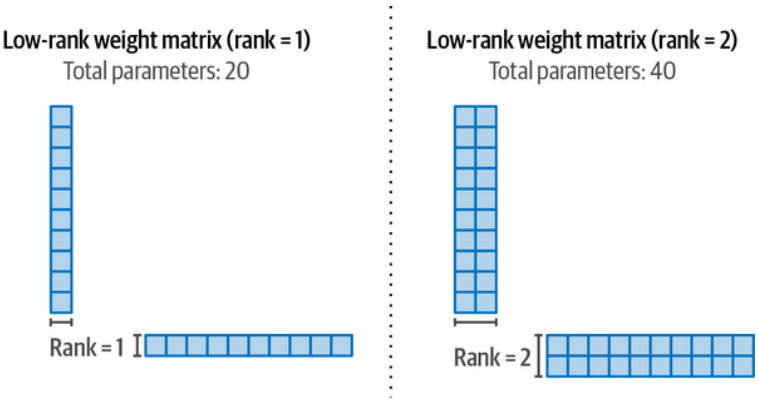

Figure 12-13. Decomposing a large weight matrix into two smaller matrices leads to a compressed, low-rank version of the matrix that can be fine-tuned more efficiently.

During training, we only need to update these smaller matrices instead of the full weight changes. The updated change matrices (smaller matrices) are then combined with the full (frozen) weights as illustrated in Figure 12-14.

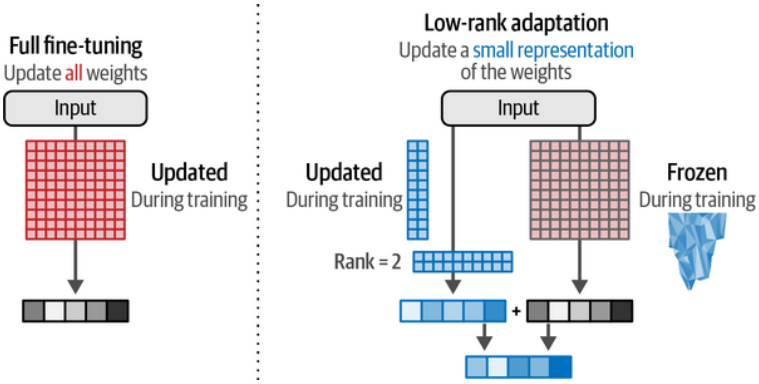

Figure 12-14. Compared to full fine-tuning, LoRA aims to update a small representation of the original weights during training.

But you might suspect that performance would drop. And you would be right. But where does this trade-off make sense?

Papers like "Intrinsic dimensionality explains the effectiveness of language model fine-tuning" demonstrate that language models "have a very low intrinsic dimension."[5] This means that we can find small ranks that approximate even the massive matrices of an LLM. A 175B model like GPT-3, for example, would have a weight matrix of 12,288 × 12,288 inside each of its 96 Transformer blocks. That's 150 million parameters. If we can successfully adapt that matrix into rank 8, that would only require two 12,288 × 2 matrices resulting in 197K parameters per block. These are major savings in speed, storage, and compute as explained further in the previously referenced LoRA paper.

This smaller representation is quite flexible in that you can select which parts of the base model to fine-tune. For instance, we can only fine-tune the Query and Value weight matrices in each Transformer layer.

Compressing the model for (more) efficient training

We can make LoRA even more efficient by reducing the memory requirements of the model's original weights before projecting them into smaller matrices. The weights of an LLM are numeric values with a given precision, which can be expressed by the number of bits like float64 or float32. As illustrated in Figure 12-15, if we lower the amount of bits to represent a value, we get a less accurate result. However, if we lower the number of bits we also lower the memory requirements of that model.

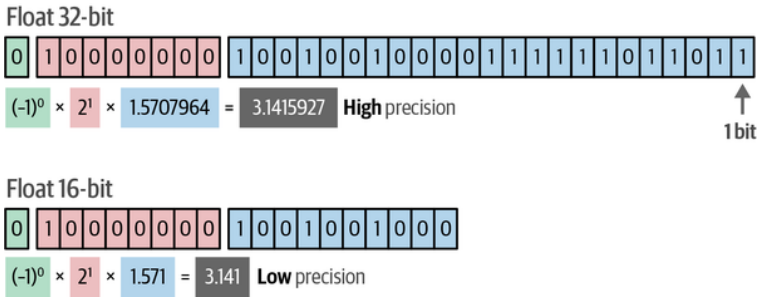

Figure 12-15. Attempting to represent pi with float 32-bit and float 16-bit representations. Notice the lowered accuracy when we halve the number of bits.

With quantization, we aim to lower the number of bits while still accurately representing the original weight values. However, as shown in Figure 12-16, when directly mapping higher precision values to lower precision values, multiple higher precision values might end up being represented by the same lower precision values.

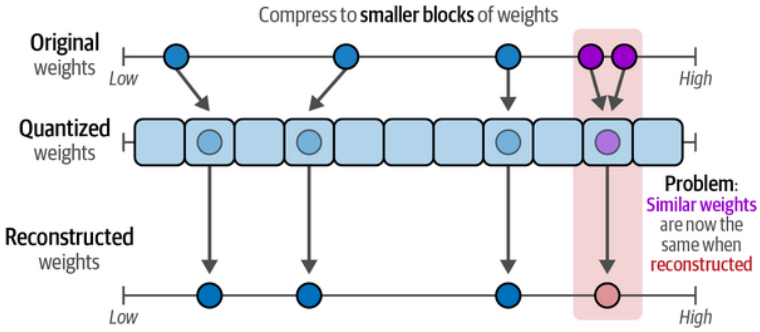

Figure 12-16. Quantizing weights that are close to one another results in the same reconstructed weights thereby removing any differentiating factor.

Instead, the authors of QLoRA, a quantized version of LoRA, found a way to go from a higher number of bits to a lower value and vice versa without differentiating too much from the original weights.[6]

They used blockwise quantization to map certain blocks of higher precision values to lower precision values. Instead of directly mapping higher precision to lower precision values, additional blocks are created that allow for quantizing similar weights. As shown in Figure 12-17, this results in values that can be accurately represented with lower precision.

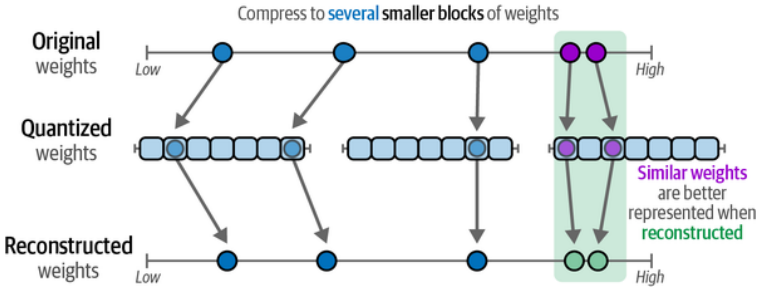

Figure 12-17. Blockwise quantization can accurately represent weights in lower precision through quantization blocks.

A nice property of neural networks is that their values are generally normally distributed between –1 and 1. This property allows us to bin the original weights to lower bits based on their relative density, as illustrated in Figure 12-18. The mapping between weights is more efficient as it takes into account the relative frequency of weights. This also reduces issues with outliers.

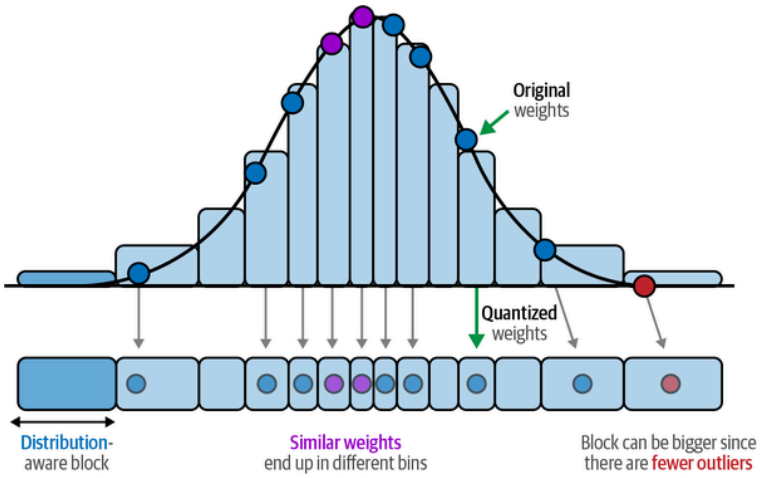

Figure 12-18. Using distribution-aware blocks we can prevent values close to one another from being represented with the same quantized value.

Combined with the blockwise quantization, this normalization procedure allows for accurate representation of high precision values by low precision values with only a small decrease in the performance of the LLM. As a result, we can go from a 16-bit float representation to a measly 4-bit normalized float representation. A 4-bit representation significantly reduces the memory requirements of the LLM during training. Note that the quantization of LLMs in general is also helpful for inference as quantized LLMs are smaller in size and therefore require less VRAM.

There are more elegant methods to further optimize this like double quantization and paged optimizers, which you can read about more in the QLoRA paper discussed earlier. For a complete and highly visual guide to quantization, see this blog post.

Instruction Tuning with QLoRA

Now that we have explored how QLoRA works, let us put that knowledge into practice! In this section, we will fine-tune a completely open source and smaller version of Llama, TinyLlama, to follow instructions using the QLoRA procedure. Consider this model a base or pretrained model, one that was trained with language modeling but cannot yet follow instructions.

Templating Instruction Data

To have the LLM follow instructions, we will need to prepare instruction data that follows a chat template. This chat template, as illustrated in Figure 12-19, differentiates between what the LLM has generated and what the user has generated.

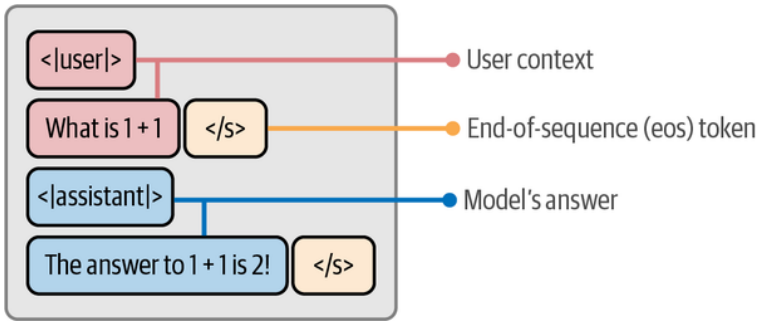

Figure 12-19. The chat template that we use throughout this chapter.

We chose this chat template to use throughout the examples since the chat version of TinyLlama uses the same format. The data that we are using is a small subset of the UltraChat dataset.[7] This dataset is a filtered version of the original UltraChat dataset that contains almost 200k conversations between a user and an LLM.

We create a function, `format_prompt`, to make sure that the conversations follow this template:

```
from transformers import AutoTokenizer
from datasets import load_dataset

# Load a tokenizer to use its chat template
template_tokenizer = AutoTokenizer.from_pretrained(
    "TinyLlama/TinyLlama-1.1BChat-v1.0"
)

def format_prompt(example):
    """Format the prompt to using the <|user|> template TinyLLama is using"""

    # Format answers
    chat = example["messages"]
    prompt = template_tokenizer.apply_chat_template(chat, tokenize=False)

    return {"text": prompt}

# Load and format the data using the template TinyLLama is using
dataset = (
    load_dataset("HuggingFaceH4/ultrachat_200k", split="test_sft")
      .shuffle(seed=42)
      .select(range(3_000))
```

```
)
dataset = dataset.map(format_prompt)
```

We select a subset of 3,000 documents to reduce the training time, but you can increase this value to get more accurate results.

Using the `"text"` column, we can explore these formatted prompts:

```
# Example of formatted prompt
print(dataset["text"][2576])
```

```
<|user|>
Given the text: Knock, knock. Who's there? Hike.
Can you continue the joke based on the given text material "Knock, knock. Who's there? Hike"?</s>
<|assistant|>
Sure! Knock, knock. Who's there? Hike. Hike who? Hike up your pants, it's cold outside!</s>
<|user|>
Can you tell me another knock-knock joke based on the same text material "Knock, knock. Who's there? Hik
<|assistant|>
Of course! Knock, knock. Who's there? Hike. Hike who? Hike your way over here and let's go for a walk!</
```

Model Quantization

Now that we have our data, we can start loading in our model. This is where we apply the Q in QLoRA, namely quantization. We use the `bitsandbytes` package to compress the pretrained model to a 4-bit representation.

In `BitsAndBytesConfig`, you can define the quantization scheme. We follow the steps used in the original QLoRA paper and load the model in 4-bit (`load_in_4bit`) with a normalized float representation (`bnb_4bit_quant_type`) and double quantization (`bnb_4bit_use_double_quant`):

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig

model_name = "TinyLlama/TinyLlama-1.1B-intermediate-step-1431k-3T"

# 4-bit quantization configuration - Q in QLoRA
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,  # Use 4-bit precision model loading
    bnb_4bit_quant_type="nf4",  # Quantization type
    bnb_4bit_compute_dtype="float16",  # Compute dtype
    bnb_4bit_use_double_quant=True,  # Apply nested quantization
)

# Load the model to train on the GPU
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    device_map="auto",

    # Leave this out for regular SFT
    quantization_config=bnb_config,
)
model.config.use_cache = False
model.config.pretraining_tp = 1
```

```
# Load LLaMA tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
tokenizer.pad_token = "<PAD>"
tokenizer.padding_side = "left"
```

This quantization procedure allows us to decrease the size of the original model while retaining most of the original weights' precision. Loading the model now only uses ~1 GB VRAM compared to the ~4 GB of VRAM it would need without quantization. Note that during fine-tuning, more VRAM will be necessary so it does not cap out on the ~1 GB VRAM needed to load the model.

LoRA Configuration

Next, we will need to define our LoRA configuration using the `peft` library, which represents hyperparameters of the fine-tuning process:

```
from peft import LoraConfig, prepare_model_for_kbit_training, get_peft_model

# Prepare LoRA Configuration
peft_config = LoraConfig(
    lora_alpha=32,  # LoRA Scaling
    lora_dropout=0.1,  # Dropout for LoRA Layers
    r=64,  # Rank
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=  # Layers to target
     ["k_proj", "gate_proj", "v_proj", "up_proj", "q_proj", "o_proj", "down_proj"]
)

# Prepare model for training
model = prepare_model_for_kbit_training(model)
model = get_peft_model(model, peft_config)
```

There are several parameters worth mentioning:

`r`
: This is the rank of the compressed matrices (recall this from Figure 12-13) Increasing this value will also increase the sizes of compressed matrices leading to less compression and thereby improved representative power. Values typically range between 4 and 64.

`lora_alpha`
: Controls the amount of change that is added to the original weights. In essence, it balances the knowledge of the original model with that of the new task. A rule of thumb is to choose a value twice the size of `r`.

`target_modules`
: Controls which layers to target. The LoRA procedure can choose to ignore specific layers, like specific projection layers. This can speed up training but reduce performance and vice versa.

Playing around with the parameters is a worthwhile experiment to get an intuitive understanding of values that work and those that do not. You can find an amazing resource of additional tips on LoRA fine-tuning in the Ahead of AI newsletter by Sebastian Raschka.

NOTE

This example demonstrates an efficient form of fine-tuning your model. If you want to perform full fine-tuning instead, you can remove the `quantization_config` parameter when loading the model and skip the creation of `peft_config`. By removing those, we would go from "Instruction tuning with QLoRA" to "full instruction tuning."

Training Configuration

Lastly, we need to configure our training parameters as we did in Chapter 11:

```
from transformers import TrainingArguments

output_dir = "./results"

# Training arguments
training_arguments = TrainingArguments(
    output_dir=output_dir,
    per_device_train_batch_size=2,
    gradient_accumulation_steps=4,
    optim="paged_adamw_32bit",
    learning_rate=2e-4,
    lr_scheduler_type="cosine",
    num_train_epochs=1,
    logging_steps=10,
    fp16=True,
    gradient_checkpointing=True
)
```

There are several parameters worth mentioning:

num_train_epochs
: The total number of training rounds. Higher values tend to degrade performance so we generally like to keep this low.

learning_rate
: Determines the step size at each iteration of weight updates. The authors of QLoRA found that higher learning rates work better for larger models (>33B parameters).

lr_scheduler_type
: A cosine-based scheduler to adjust the learning rate dynamically. It will linearly increase the learning rate, starting from zero, until it reaches the set value. After that, the learning rate is decayed following the values of a cosine function.

optim
: The paged optimizers used in the original QLoRA paper.

Optimizing these parameters is a difficult task and there are no set guidelines for doing so. It requires experimentation to figure out what works best for specific datasets, model sizes, and target tasks.

NOTE

Although this section describes instruction tuning, we could also use QLoRA to fine-tune an instruction model. For instance, we could fine-tune a chat model to generate specific SQL code or to create JSON output that adheres to a specific format. As long as you have the data available (with appropriate query-response items), QLoRA is a great technique for nudging an existing chat model to be more appropriate for your use case.

Training

Now that we have prepared all our models and parameters, we can start fine-tuning our model. We load in `SFTTrainer` and simply run `trainer.train()`:

```
from trl import SFTTrainer

# Set supervised fine-tuning parameters
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    dataset_text_field="text",
    tokenizer=tokenizer,
    args=training_arguments,
    max_seq_length=512,

    # Leave this out for regular SFT
    peft_config=peft_config,
)

# Train model
trainer.train()

# Save QLoRA weights
trainer.model.save_pretrained("TinyLlama-1.1B-qlora")
```

During training the loss will be printed every 10 steps according to the `logging_steps` parameter. If you are using the free GPU provided by Google Colab, which is the Tesla T4 at the time of writing, then training might take up to an hour. A good time to take a break!

Merge Weights

After we have trained our QLoRA weights, we still need to combine them with the original weights to use them. We reload the model in 16 bits, instead of the quantized 4 bits, to merge the weights. Although the tokenizer was not updated during training, we save it to the same folder as the model for easier access:

```
from peft import AutoPeftModelForCausalLM

model = AutoPeftModelForCausalLM.from_pretrained(
    "TinyLlama-1.1B-qlora",
    low_cpu_mem_usage=True,
    device_map="auto",
)

# Merge LoRA and base model
merged_model = model.merge_and_unload()
```

After merging the adapter with the base model, we can use it with the prompt template that we defined earlier:

```
from transformers import pipeline

# Use our predefined prompt template
prompt = """<|user|>
Tell me something about Large Language Models.</s>
<|assistant|>
"""

# Run our instruction-tuned model
pipe = pipeline(task="text-generation", model=merged_model, tokenizer=tokenizer)
print(pipe(prompt)[0]["generated_text"])
```

```
Large Language Models (LLMs) are artificial intelligence (AI) models that learn language and understand
```

The aggregate output shows that the model now closely follows our instructions, which is not possible with the base model.

Evaluating Generative Models

Evaluating generative models poses a significant challenge. Generative models are used across many diverse use cases, making it a challenge to rely on a singular metric for judgment. Unlike more specialized models, a generative model's ability to solve mathematical questions does not guarantee success in solving coding questions.

At the same time, evaluating these models is vital, particularly in production settings where consistency is important. Given their probabilistic nature, generative models do not necessarily generate consistent outputs; there is a need for robust evaluation.

In this section, we will explore a few common evaluation methods, but we want to emphasize the current lack of golden standards. No one metric is perfect for all use cases.

Word-Level Metrics

One common metrics category for comparing generative models is word-level evaluation. These classic techniques compare a reference dataset with the generated tokens on a token(set) level. Common word-level metrics include perplexity,[8] ROUGE,[9] BLEU,[10] and BERTScore.[11]

Of note is perplexity, which measures how well a language model predicts a text. Given input text, the model predicts how likely the next token is. With perplexity, we assume a model performs better if it gives the next token a high probability. In other words, the models should not be "perplexed" when presented with a well-written document.

As illustrated in Figure 12-20, when presented with the input "When a measure becomes a," the model is asked how probable the word "target" is as the next word.

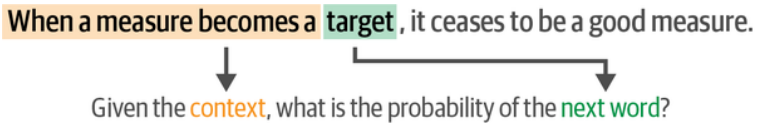

Figure 12-20. Next-word prediction is a central feature of many LLMs.

Although perplexity, and other word-level metrics, are useful metrics to understand the confidence of the model, they are not a perfect measure. They do not account for consistency, fluency, creativity, or even correctness of the generated text.

Benchmarks

A common method for evaluating generative models on language generation and understanding tasks is on well-known and public benchmarks, such as MMLU,[12] GLUE,[13] TruthfulQA,[14] GSM8k,[15] and HellaSwag.[16] These benchmarks give us information about basic language understanding but also complex analytical answering, like math problems.

Aside from natural language tasks, some models specialize in other domains, like programming. These models tend to be evaluated on different benchmarks, such as HumanEval,[17] which consists of challenging programming tasks for the model to solve. Table 12-1 gives an overview of common public benchmarks for generative models.

Table 12-1. Common public benchmarks for generative models

Benchmark	Description	Resources
MMLU	The Massive Multitask Language Understanding (MMLU) benchmark tests the model on 57 different tasks, including classification, question answering, and sentiment analysis.	*https://oreil.ly/nrG_g*
GLUE	The General Language Understanding Evaluation (GLUE) benchmark consists of language understanding tasks covering a wide degree of difficulty.	*https://oreil.ly/LV_fb*
TruthfulQA	TruthfulQA measures the truthfulness of a model's generated text.	*https://oreil.ly/i2Brj*
GSM8k	The GSM8k dataset contains grade-school math word problems. It is linguistically diverse and created by human problem writers.	*https://oreil.ly/oOBXY*
HellaSwag	HellaSwag is a challenge dataset for evaluating common-sense inference. It consists of multiple-choice questions that the model needs to answer. It can select one of four answer choices for each question.	*https://oreil.ly/aDvBP*
HumanEval	The HumanEval benchmark is used for evaluating generated code based on 164 programming problems.	*https://oreil.ly/dlJIX*

Benchmarks are a great way to get a basic understanding on how well a model performs on a wide variety of tasks. A downside to public benchmarks is that models can be overfitted to these benchmarks to generate the best responses. Moreover, these are still broad benchmarks and might not cover very specific use cases. Lastly, another downside is that some benchmarks require strong GPUs with a long running time (over hours) to compute, which makes iteration difficult.

Leaderboards

With so many different benchmarks, it is hard to choose which benchmark best suits your model. Whenever a model is released, you will often see it evaluated on several benchmarks to showcase how it performs across the board.

As such, leaderboards were developed containing multiple benchmarks. A common leaderboard is the Open LLM Leaderboard, which, at the time of writing, includes six benchmarks, including HellaSwag, MMLU, TruthfulQA, and GSM8k. Models that top the leaderboard, assuming they were not overfitted on the data, are generally regarded as the "best" model. However, since these leaderboards often contain publicly available benchmarks, there is a risk of overfitting on the leaderboard.

Automated Evaluation

Part of evaluating a generative output is the quality of its text. For instance, even if two models were to give the same correct answer to a question, the way they derived that answer might be different. It is often not just about the final answer but also the construction of it. Similarly, although two summaries might be similar, one could be significantly shorter than another, which is often important for a good summary.

To evaluate the quality of the generated text above the correctness of the final answer, LLM-as-a-judge was introduced.[18] In essence, a separate LLM is asked to judge the quality of the LLM to be evaluated. An interesting variant of this method is pairwise comparison. Two different LLMs will generate an answer to a question and a third LLM will be the judge to declare which is better.

As a result, this methodology allows for automated evaluation of open-ended questions. A major advantage is that as LLMs improve, so do their capabilities to judge the quality of output. In other words, this evaluation methodology grows with the field.

Human Evaluation

Although benchmarks are important, the gold standard of evaluation is generally considered to be human evaluation. Even if an LLM scores well on broad benchmarks, it still might not score well on domain-specific tasks. Moreover, benchmarks do not fully capture human preference and all methods discussed before are merely proxies for that.

A great example of a human-based evaluation technique is the Chatbot Arena.[19] When you go to this leaderboard you are shown two (anonymous) LLMs you can interact with. Any question or prompt you ask will be sent to both models and you will receive their output. Then, you can

decide which output you prefer. This process allows for the community to vote on which models they prefer without knowing which ones are presented. Only after you vote do you see which model generated which text.

At the time of writing, this method has generated over 800,000+ human votes that were used to compute a leaderboard. These votes are used to calculate the relative skill level of LLMs based on their win rates. For instance, if a low-ranked LLM beats a high-ranked LLM, its ranking changes significantly. In chess, this is referred to as the Elo rating system.

This methodology therefore uses crowdsourced votes, which helps us understand the quality of the LLM. However, it is still the aggregated opinion of a wide variety of users, which might not relate to your use case.

As a result, there is no one perfect method of evaluating LLMs. All mentioned methodologies and benchmarks provide an important, although limited evaluation perspective. Our advice is to evaluate your LLM based on the intended use case. For coding, HumanEval would be more logical than GSM8k.

But most importantly, we believe that you are the best evaluator. Human evaluation remains the gold standard because it is up to you to decide whether the LLM works for your intended use case. As with the examples in this chapter, we highly advise that you also try these models and perhaps develop some questions yourself. For example, the authors of this book are Arabic (Jay Alammar) and Dutch (Maarten Grootendorst), and we often ask questions in our native language when approached with new models.

One final note on this topic is a quote we hold dear:

> *When a measure becomes a target, it ceases to be a good measure.*
>
> —Goodhart's Law[20]

In the context of LLMs, when using a specific benchmark, we tend to optimize for that benchmark regardless of the consequences. For instance, if we focus purely on optimizing for generating grammatically correct sentences, the model could learn to only output one sentence: "This is a sentence." It is grammatically correct but tells you nothing about its language understanding capabilities. Thus, the model may excel at a specific benchmark but potentially at the expense of other useful capabilities.

Preference-Tuning / Alignment / RLHF

Although our model can now follow instructions, we can further improve its behavior by a final training phase that aligns it to how we expect it to behave in different scenarios. For instance, when asked "What is an LLM?" we might prefer an elaborate answer that describes the internals of an LLM compared to the answer "It is a large language model" without further explanations. How exactly do we align our (human) preference for one answer over the other with the output of an LLM?

To start with, recall that an LLM takes a prompt and outputs a generation as illustrated in Figure 12-21.

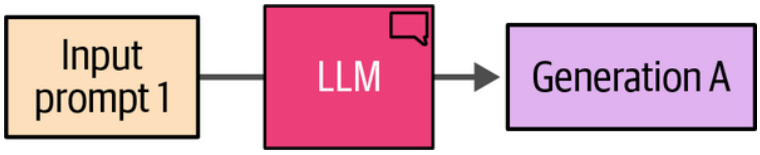

Figure 12-21. An LLM takes an input prompt and outputs a generation.

We can ask a person (preference evaluator) to evaluate the quality of that model generation. Say they assign it a certain score, like 4 (see Figure 12-22).

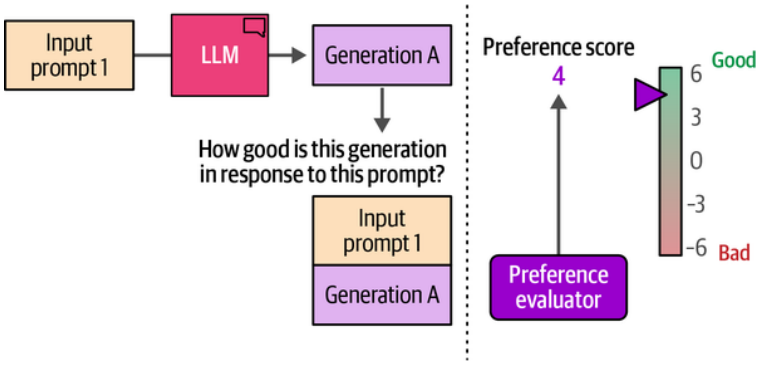

Figure 12-22. Use a preference evaluator (human or otherwise) to evaluate the quality of the generation.

Figure 12-23 shows a preference tuning step updating the model based on that score:

- If the score is high, the model is updated to encourage it to generate more like this type of generation.
- If the score is low, the model is updated to discourage such generations.

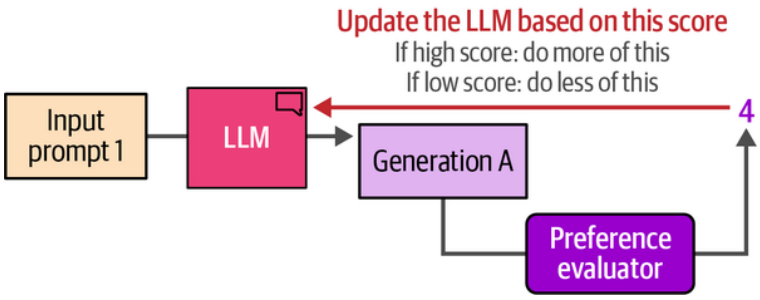

Figure 12-23. Preference tuning methods update the LLM based on the evaluation score.

As always, we need many training examples. So can we automate the preference evaluation? Yes, we can by training a different model called a reward model.

Automating Preference Evaluation Using Reward Models

To automate preference evaluation, we need a step before the preference-tuning step, namely to train a reward model, as shown in Figure 12-24.

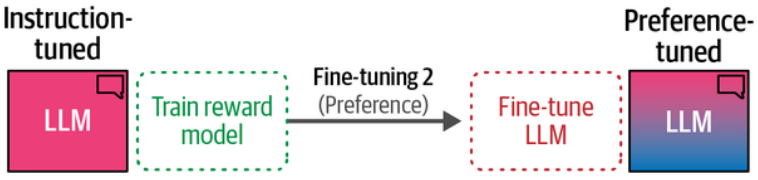

Figure 12-24. We train a reward model before fine-tuning the LLM.

Figure 12-25 shows that to create a reward model, we take a copy of the instruction-tuned model and slightly change it so that instead of generating text, it now outputs a single score.

Figure 12-25. The LLM becomes a reward model by replacing its language modeling head with a quality classification head.

The Inputs and Outputs of a Reward Model

The way we expect this reward model to work is that we give it a prompt and a generation, and it outputs a single number indicating the preference/quality of that generation in response to that prompt. Figure 12-26 shows the reward model generating this single number.

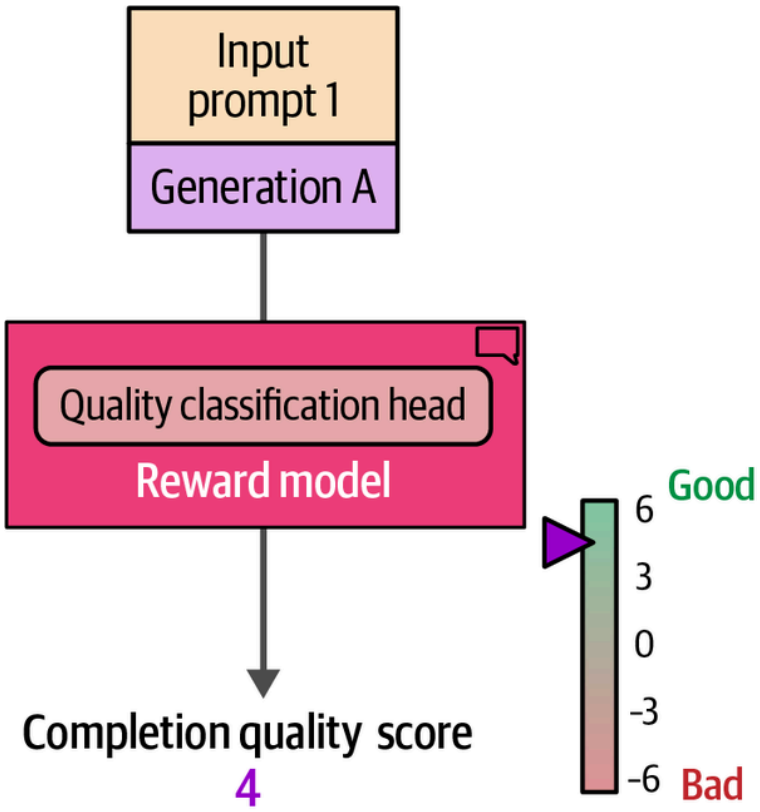

Figure 12-26. Use a reward model trained on human preference to generate the completion quality score.

Training a Reward Model

We cannot directly use the reward model. It needs to first be trained to properly score generations. So let's get a preference dataset that the model can learn from.

Reward model training dataset

One common shape for preference datasets is for a training example to have a prompt, with one accepted generation and one rejected generation. (Nuance: it's not always a good versus bad generation; it can be that the two generations are both good, but that one is better than the other). Figure 12-27 shows an example preference training set with two training examples.

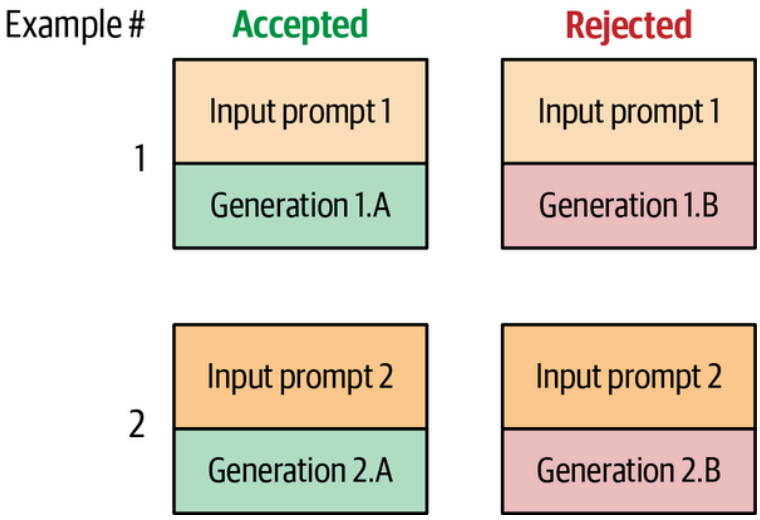

Figure 12-27. Preference tuning datasets are often made up of prompts with accepted and rejected generations.

One way to generate preference data is to present a prompt to the LLM and have it generate two different generations. As shown in Figure 12-28, we can ask human labelers which of the two they prefer.

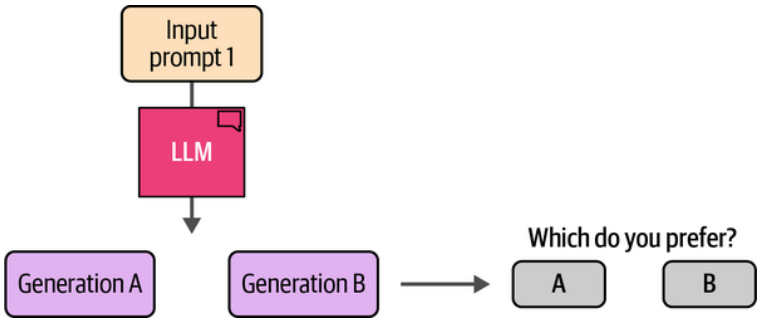

Figure 12-28. Output two generations and ask a human labeler which one they prefer.

Reward model training step

Now that we have the preference training dataset, we can proceed to train the reward model.

A simple step is that we use the reward model to:

1. Score the accepted generation
2. Score the rejected generation

Figure 12-29 shows the training objective: to ensure the accepted generation has a higher score than the rejected generation.

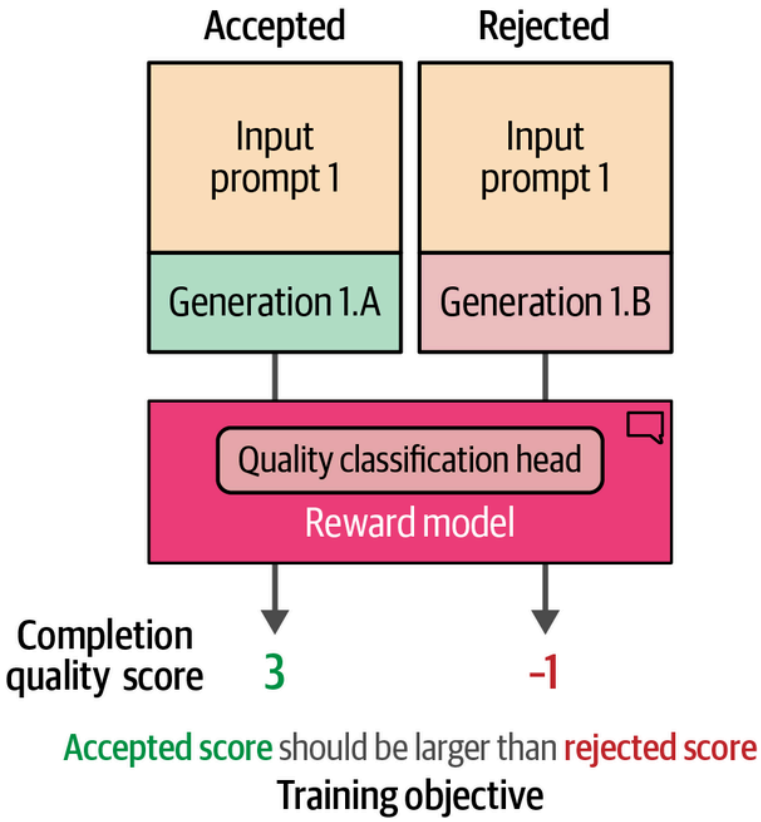

Figure 12-29. The reward model aims to evaluate the quality scores of generations in response to a prompt.

When we combine everything together as shown in Figure 12-30, we get the three stages to preference tuning:

1. Collect preference data
2. Train a reward model
3. Use the reward model to fine-tune the LLM (operating as the preference evaluator)

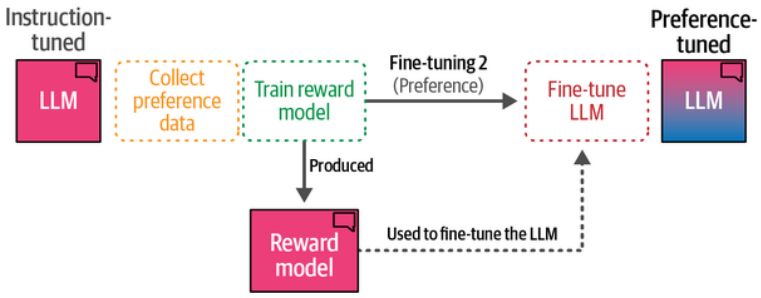

Figure 12-30. The three stages of preference tuning: collecting preference data, training a reward model, and finally fine-tuning the LLM.

Reward models are an excellent idea that can be further extended and developed. Llama 2, for example, trains two reward models: one that scores helpfulness and another that scores safety (Figure 12-31).

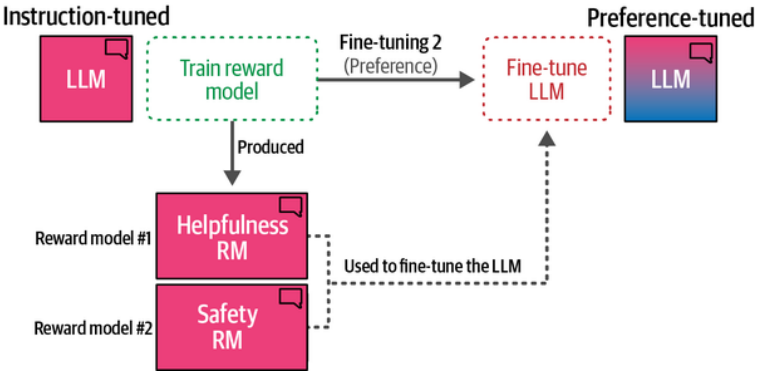

Figure 12-31. We can use multiple reward models to perform the scoring.

A common method to fine-tune the LLM with the trained reward model is Proximal Policy Optimization (PPO). PPO is a popular reinforcement technique that optimizes the instruction-tuned LLM by making sure that the LLM does not deviate too much from the expected rewards.[21] It was even used to train the original ChatGPT released in November 2022.

Training No Reward Model

A disadvantage of PPO is that it is a complex method that needs to train at least two models, the reward model and the LLM, which can be more costly than perhaps necessary.

Direct Preference Optimization (DPO) is an alternative to PPO and does away with the reinforcement-based learning procedure.[22] Instead of using the reward model to judge the quality of a generation, we let the LLM itself do that. As illustrated in Figure 12-32, we use a copy of the LLM as the reference model to judge the shift between the reference and trainable model in the quality of the accepted generation and rejected generation.

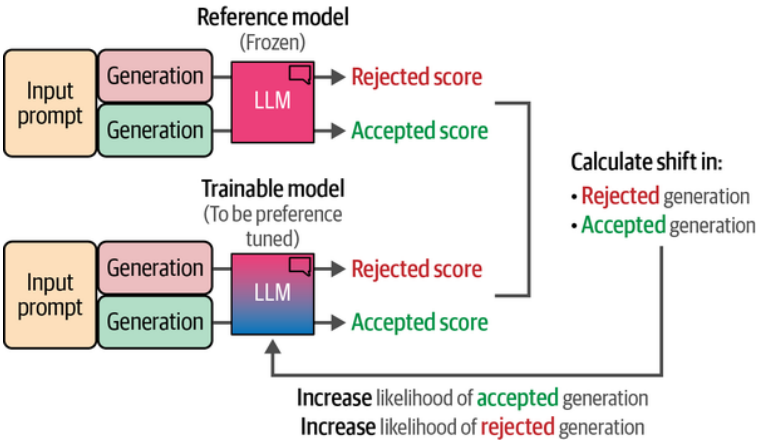

Figure 12-32. Use the LLM itself as the reward model by comparing the output of a frozen model with the trainable model.

By calculating this shift during training, we can optimize the likelihood of accepted generations over rejected generations by tracking the difference in the reference model and the trainable model.

To calculate this shift and its related scores, the log probabilities of the rejected generations and accepted generations are extracted from both models. As illustrated in Figure 12-33, this process is performed at a token level where the probabilities are combined to calculate the shift between the reference and trainable models.

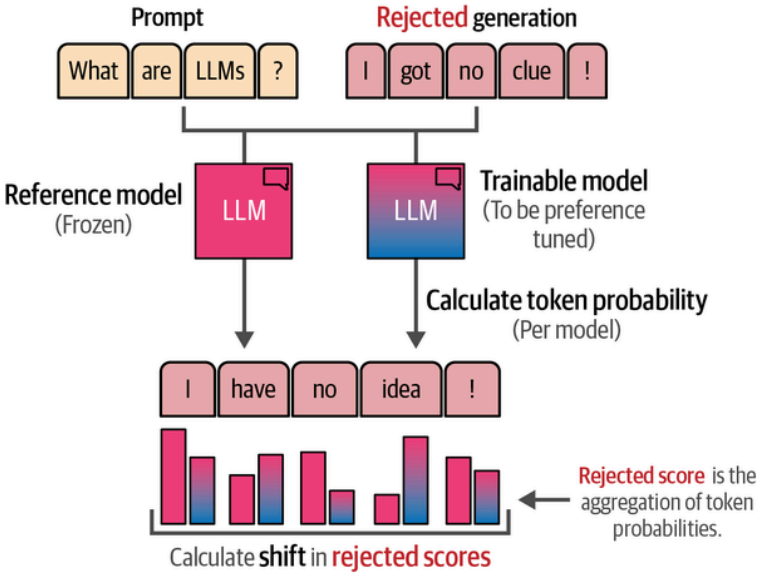

Figure 12-33. Scores are calculated by taking the probabilities of generation on a token level. The shift in probabilities between the reference model and the trainable model is optimized. The accepted generation follows the same procedure.

Using these scores, we can optimize the parameters of the trainable model to be more confident of generating the accepted generations and less confident of generating the rejected generations. Compared to PPO, the authors found DPO to be more stable during training and more accurate. Due to its stability, we will be using it as our primary model for preference tuning our previously instruction-tuned model.

Preference Tuning with DPO

When we use the Hugging Face stack, preference tuning is eerily similar to the instruction tuning we covered before with some slight differences. We will still be using TinyLlama but this time an instruction-tuned version that was first trained using full fine-tuning and then further aligned with DPO. Compared to our initial instruction-tuned model, this LLM was trained on much larger datasets.

In this section, we will demonstrate how you can further align this model using DPO with reward-based datasets.

Templating Alignment Data

We will use a dataset that for each prompt contains an accepted generation and a rejected generation. This dataset was in part generated by ChatGPT with scores on which output should be accepted and which rejected:

```
from datasets import load_dataset

def format_prompt(example):
    """Format the prompt to using the <|user|> template TinyLLama is using"""

    # Format answers
    system = "<|system|>\n" + example["system"] + "</s>\n"
    prompt = "<|user|>\n" + example["input"] + "</s>\n<|assistant|>\n"
    chosen = example["chosen"] + "</s>\n"
    rejected = example["rejected"] + "</s>\n"

    return {
```

```
        "prompt": system + prompt,
        "chosen": chosen,
        "rejected": rejected,
    }

# Apply formatting to the dataset and select relatively short answers
dpo_dataset = load_dataset(
    "argilla/distilabel-intel-orca-dpo-pairs", split="train"
)
dpo_dataset = dpo_dataset.filter(
    lambda r:
        r["status"] != "tie" and
        r["chosen_score"] >= 8 and
        not r["in_gsm8k_train"]
)
dpo_dataset = dpo_dataset.map(
    format_prompt,  remove_columns=dpo_dataset.column_names
)
dpo_dataset
```

Note that we apply additional filtering to further reduce the size of the data to roughly 6,000 examples from the original 13,000 examples.

Model Quantization

We load our base model and load it with the LoRA we created previously. As before, we quantize the model to reduce the necessary VRAM for training:

```
from peft import AutoPeftModelForCausalLM
from transformers import BitsAndBytesConfig, AutoTokenizer

# 4-bit quantization configuration - Q in QLoRA
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,  # Use 4-bit precision model loading
    bnb_4bit_quant_type="nf4",  # Quantization type
    bnb_4bit_compute_dtype="float16",  # Compute dtype
    bnb_4bit_use_double_quant=True,  # Apply nested quantization
)

# Merge LoRA and base model
model = AutoPeftModelForCausalLM.from_pretrained(
    "TinyLlama-1.1B-qlora",
    low_cpu_mem_usage=True,
    device_map="auto",
    quantization_config=bnb_config,
)
merged_model = model.merge_and_unload()

# Load LLaMA tokenizer
model_name = "TinyLlama/TinyLlama-1.1B-intermediate-step-1431k-3T"
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
tokenizer.pad_token = "<PAD>"
tokenizer.padding_side = "left"
```

Next, we use the same LoRA configuration as before to perform the DPO training:

```
from peft import LoraConfig, prepare_model_for_kbit_training, get_peft_model

# Prepare LoRA configuration
```

```
peft_config = LoraConfig(
    lora_alpha=32,  # LoRA Scaling
    lora_dropout=0.1,  # Dropout for LoRA Layers
    r=64,  # Rank
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=  # Layers to target
     ["k_proj", "gate_proj", "v_proj", "up_proj", "q_proj", "o_proj", "down_proj"]
)

# prepare model for training
model = prepare_model_for_kbit_training(model)
model = get_peft_model(model, peft_config)
```

Training Configuration

For the sake of simplicity, we will use the same training arguments as we did before with one difference. Instead of running for a single epoch (which can take up to two hours), we run for 200 steps instead for illustration purposes. Moreover, we added the `warmup_ratio` parameter, which increases the learning rate from 0 to the `learning_rate` value we set for the first 10% of steps. By maintaining a small learning rate at the start (i.e., warmup period), we allow the model to adjust to the data before applying larger learning rates, therefore avoiding harmful divergence:

```
from trl import DPOConfig

output_dir = "./results"

# Training arguments
training_arguments = DPOConfig(
    output_dir=output_dir,
    per_device_train_batch_size=2,
    gradient_accumulation_steps=4,
    optim="paged_adamw_32bit",
    learning_rate=1e-5,
    lr_scheduler_type="cosine",
    max_steps=200,
    logging_steps=10,
    fp16=True,
    gradient_checkpointing=True,
    warmup_ratio=0.1
)
```

Training

Now that we have prepared all our models and parameters, we can start fine-tuning our model:

```
from trl import DPOTrainer

# Create DPO trainer
dpo_trainer = DPOTrainer(
    model,
    args=training_arguments,
    train_dataset=dpo_dataset,
    tokenizer=tokenizer,
    peft_config=peft_config,
    beta=0.1,
    max_prompt_length=512,
```

```
    max_length=512,
)

# Fine-tune model with DPO
dpo_trainer.train()

# Save adapter
dpo_trainer.model.save_pretrained("TinyLlama-1.1B-dpo-qlora")
```

We have created a second adapter. To merge both adapters, we iteratively merge the adapters with the base model:

```
from peft import PeftModel

# Merge LoRA and base model
model = AutoPeftModelForCausalLM.from_pretrained(
    "TinyLlama-1.1B-qlora",
    low_cpu_mem_usage=True,
    device_map="auto",
)
sft_model = model.merge_and_unload()

# Merge DPO LoRA and SFT model
dpo_model = PeftModel.from_pretrained(
    sft_model,
    "TinyLlama-1.1B-dpo-qlora",
    device_map="auto",
)
dpo_model = dpo_model.merge_and_unload()
```

This combination of SFT+DPO is a great way to first fine-tune your model to perform basic chatting and then align its answers with human preference. However, it does come at a cost since we need to perform two training loops and potentially tweak the parameters in two processes.

Since the release of DPO, new methods of aligning preferences have been developed. Of note is Odds Ratio Preference Optimization (ORPO), a process that combines SFT and DPO into a single training process.[23] It removes the need to perform two separate training loops, further simplifying the training process while allowing for the use of QLoRA.

Summary

In this chapter, we explored different steps of fine-tuning pretrained LLMs. We performed fine-tuning by making use of parameter-efficient fine-tuning (PEFT) through the low-rank adaptation (LoRA) technique. We explained how LoRA can be extended through quantization, a technique for reducing memory constraints when representing the parameters of the model and adapters.

The fine-tuning process we explored has two steps. In the first step, we performed supervised fine-tuning using instruction data on a pretrained LLM, often called instruction tuning. This resulted in a model that has chat-like behavior and could closely follow instructions.

In the second step, we further improved the model by fine-tuning it on alignment data, data that represents what type of answers are preferred

over others. This process, referred to as preference tuning, distills human preference to the previously instruction-tuned model.

Overall, this chapter has shown the two major steps of fine-tuning a pre-trained LLM and how that could lead to more accurate and informative outputs.

1 Neil Houlsby et al. “Parameter-efficient transfer learning for NLP.” *International Conference on Machine Learning*. PMLR, 2019.

2 Jonas Pfeiffer et al. “AdapterHub: A framework for adapting transformers.” *arXiv preprint arXiv:2007.07779* (2020).

3 Renrui Zhang et al. “Llama-adapter: Efficient fine-tuning of language models with zero-init attention.” *arXiv preprint arXiv:2303.16199* (2023).

4 Edward J. Hu et al. “LoR: Low-Rank Adaptation of large language models.” *arXiv preprint arXiv:2106.09685* (2021).

5 Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. “Intrinsic dimensionality explains the effectiveness of language model fine-tuning.” *arXiv preprint arXiv:2012.13255* (2020).

6 Tim Dettmers et al. “QLoRA: Efficient finetuning of quantized LLMs.” *arXiv preprint arXiv:2305.14314* (2023).

7 Ning Ding et al. “Enhancing chat language models by scaling high-quality instructional conversations.” *arXiv preprint arXiv:2305.14233* (2023).

8 Fred Jelinek et al. “Perplexity—a measure of the difficulty of speech recognition tasks.” *The Journal of the Acoustical Society of America* 62.S1 (1977): S63.

9 Chin-Yew Lin. “ROUGE: A package for automatic evaluation of summaries.” *Text Summarization Branches Out*, 74–81. 2004.

10 Kishore Papineni, et al. “Bleu: a method for automatic evaluation of machine translation.” *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. 2002.

11 Tianyi Zhang et al. “BERTscore: Evaluating text generation with BERT.” *arXiv preprint arXiv:1904.09675* (2019).

12 Dan Hendrycks et al. “Measuring massive multitask language understanding.” *arXiv preprint arXiv:2009.03300* (2020).

13 Alex Wang et al. “GLUE: A multi-task benchmark and analysis platform for natural language understanding.” *arXiv preprint arXiv:1804.07461* (2018).

14 Stephanie Lin, Jacob Hilton, and Owain Evans. “TruthfulQA: Measuring how models mimic human falsehoods.” *arXiv preprint arXiv:2109.07958* (2021).

15 Karl Cobbe et al. “Training verifiers to solve math word problems.” *arXiv preprint arXiv:2110.14168* (2021).

16 Roman Zellers et al. “HellaSwag: Can a machine really finish your sentence?” *arXiv preprint arXiv:1905.07830* (2019).

17 Mark Chen et al. “Evaluating large language models trained on code.” *arXiv preprint arXiv:2107.03374* (2021).

[18] Lianmin Zheng et al. “Judging LLM-as-a-judge with MT-Bench and Chatbot Arena.” *Advances in Neural Information Processing Systems* 36 (2024).

[19] Wei-Lin Chiang et al. “Chatbot Arena: An open platform for evaluating LLMs by human preference.” *arXiv preprint arXiv:2403.04132* (2024).

[20] Mafilyn Strathern. “‘Improving ratings’: audit in the British University system.” *European Review* 5.3 (1997): 305–321.

[21] John Schulman et al. “Proximal Policy Optimization algorithms.” *arXiv preprint arXiv:1707.06347* (2017).

[22] Rafael Rafailov, et al. “Direct Preference Optimization: Your language model is secretly a reward model.” *arXiv preprint arXiv:2305.18290* (2023).

[23] Jiwoo Hong, Noah Lee, and James Thorne, “ORPO: Monolithic preference optimization without reference model”. *arXiv preprint arXiv:2403.07691* (2024).