

10

Advanced Open-Source LLM Fine-Tuning

Introduction

If I were to admit an ulterior motive for writing this book besides helping you understand and use LLMs, it would be to convince you that with the proper data and fine-tuning, smaller open-source models can be as amazing as huge closed-source models like GPT-4, especially for hyper-specific tasks. By now, I hope you understand the advantages of fine-tuning models over using closed-source models via an API. These closed-source models are truly powerful, but they don't always generalize to what we need—which is why we need to fine-tune them with our own data.

This chapter aims to help you harness the maximum potential of open-source models to deliver results that rival those possible with their larger, closed-source counterparts. By adopting the techniques and strategies outlined in this chapter, you will be able to mold and shape these models to your specific requirements.

As an ML engineer, I'd argue that the beauty of fine-tuning lies in its flexibility and adaptability, which allow us to tailor the models to our unique needs. Whether you're aiming to develop a sophisticated chatbot, a simple classifier, or a tool that can generate creative content, the fine-tuning process ensures that the model aligns with your objectives.

This journey will demand rigor, creativity, problem-solving skills, and a thorough understanding of the underlying principles of machine learning. But rest assured, the reward (pun intended for the final example) is worth the effort. Let's get started, shall we?

Example: Anime Genre Multilabel Classification with BERT

You thought I was done talking about anime? Nope, sorry. Recall that in [Chapter 7](#), we built a recommendation engine using a generated description as the base feature of an anime title; in doing so, one of the features we used was the genre list of the anime. Let's assume that our new goal is to assist people in tagging an anime's genre list given the other features. There are 42 unique genres, as shown in [Figure 10.1](#).

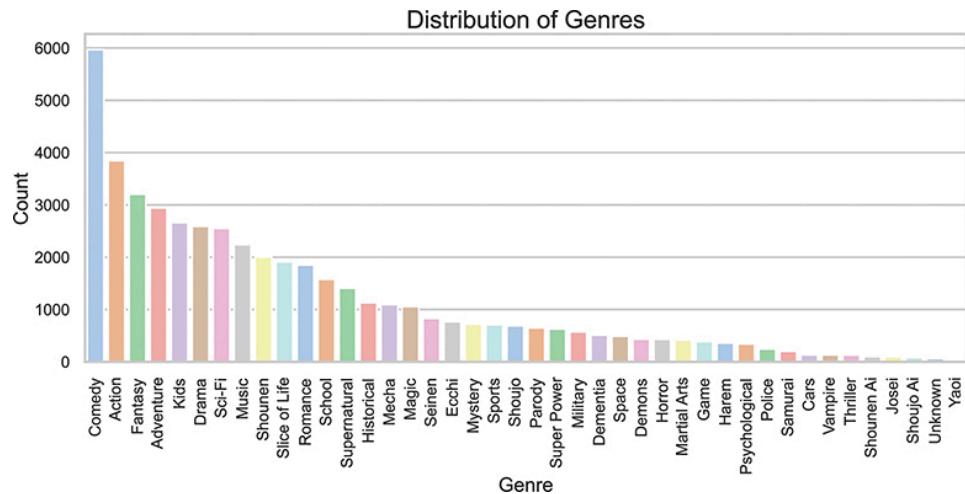


Figure 10.1 We have 42 genres to categorize from in our multilabel anime genre classification task.

Our task is to predict which genre categories an anime falls under given a description of the anime. This is a **multi-label classification**, in that each instance may have one or more labels attached to it. Because our classification task is a bit more nuanced than a simple single-label classification, we will need some different metrics to better understand how our model is performing.

Using the Jaccard Score to Measure Performance for Multilabel Genre Prediction of Anime Titles

To evaluate the performance of our genre prediction model, we will use the Jaccard score, a metric that measures the similarity between sets of items. This score is appropriate for our multilabel (we can predict multiple labels per item) genre prediction task, as it will enable us to assess the accuracy of our model in predicting the correct genres for each anime title.

[Listing 10.1](#) shows how we can define custom metrics in our `Trainer`. In this case, we will define four metrics:

- **Jaccard score:** Similar to how we used the Jaccard score in [Chapter 7](#), it will help us gauge the similarity and diversity of sample sets in this example. In the context of evaluating model performance, a higher Jaccard score indicates that the model's predictions are more similar to the actual labels.

- **F1 score:** The F1 score is a measure of a model's accuracy on a dataset. It is used to evaluate binary classification systems, which classify examples as either “positive” or “negative.” The F1 score is the harmonic mean of the precision and recall; it reaches its best value at 1 (perfect precision and recall) and its worst at 0.
- **ROC/AUC:** The receiver operating characteristic (ROC) is a probability curve; the area under the curve (AUC) represents the degree or measure of separability. The AUC indicates how well a model distinguishes between classes: The higher the AUC, the better the model is at predicting 0s as 0s and 1s as 1s.
- **Accuracy:** As you might expect, accuracy quantifies how often the predicted label matches the true label exactly. While it's easy to interpret, this metric can be misleading for imbalanced datasets, where the model can achieve a high accuracy by merely predicting the majority class.

Listing 10.1 Defining custom metrics for our multilabel genre prediction

[Click here to view code image](#)

```
# Define a function to compute several multilabel metrics
def multi_label_metrics(predictions, labels, threshold=0.5):
    # Initialize the sigmoid function, which we'll use to transform our raw prediction values
    sigmoid = torch.nn.Sigmoid()

    # Apply sigmoid function to our predictions
    probs = sigmoid(torch.Tensor(predictions))

    # Create a binary prediction array based on our threshold
    y_pred = np.zeros(probs.shape)
    y_pred[np.where(probs >= threshold)] = 1

    # Use actual labels as y_true
    y_true = labels

    # Compute F1 score, ROC/AUC score, accuracy, and Jaccard score
    f1_micro_average = f1_score(y_true=y_true, y_pred=y_pred, average='micro')
    roc_auc = roc_auc_score(y_true, y_pred, average='micro')
    accuracy = accuracy_score(y_true, y_pred)
    jaccard = jaccard_score(y_true, y_pred, average='micro')

    # Package the scores into a dictionary and return it
    metrics = {'f1': f1_micro_average,
               'roc_auc': roc_auc,
               'accuracy': accuracy,
               'jaccard': jaccard}
    return metrics
```

```
# Define a function to compute metrics for predictions
def compute_metrics(p: EvalPrediction):
    # Extract the prediction values from the EvalPrediction object
    preds = p.predictions[0] if isinstance(p.predictions, tuple) else p.predictions

    # Compute the multilabel metrics for the predictions and actual labels
    result = multi_label_metrics(predictions=preds, labels=p.label_ids)

    # Return the results
    return result
```

A Simple Fine-Tuning Loop

To fine-tune our model, we will set up the following components, each of which plays a crucial role in the customization process:

- **Dataset:** We will use our previously prepared training and testing sets from the MyAnimeList dataset. The dataset serves as the foundation for the entire fine-tuning process, as it contains the input data (synopses) and target labels (genres) that the model will learn to predict. Properly splitting the dataset into training, validation (taken from the training set), and testing sets is vital for evaluating the performance of our customized model on unseen data.
- **Data collator:** The data collator is responsible for processing and preparing the input data for our model. It takes raw input data, such as text, and transforms it into a format that the model can understand, typically involving tokenization, padding, and batching. By using a data collator, we ensure that our input data is correctly formatted and efficiently fed into the model during training.
- **TrainingArguments :** `TrainingArguments` is a configuration object provided by the Hugging Face library that allows us to specify various hyperparameters and options for the training process. These can include learning rate, batch size, number of training epochs, and more. By setting up `TrainingArguments`, we can fine-tune the training process to achieve optimal performance for our specific task.
- **Weights & Biases and Trainer :** Weights & Biases (WandB) is a library that facilitates tracking and visualizing the progress of the training process. By integrating WandB, we can monitor key metrics, such as loss and accuracy, and gain insights into how well our model is performing over time. `Trainer` is a utility provided by the Hugging Face library that manages the fine-tuning process. It handles tasks such as loading data, updating model weights, and evaluating the model's performance. By setting up a `Trainer`, we can streamline the fine-tuning process and ensure that our model is effectively trained on the task at hand.

Figure 10.2 visualizes the basic deep learning training loop using Hugging Face's built-in fine-tuning components.

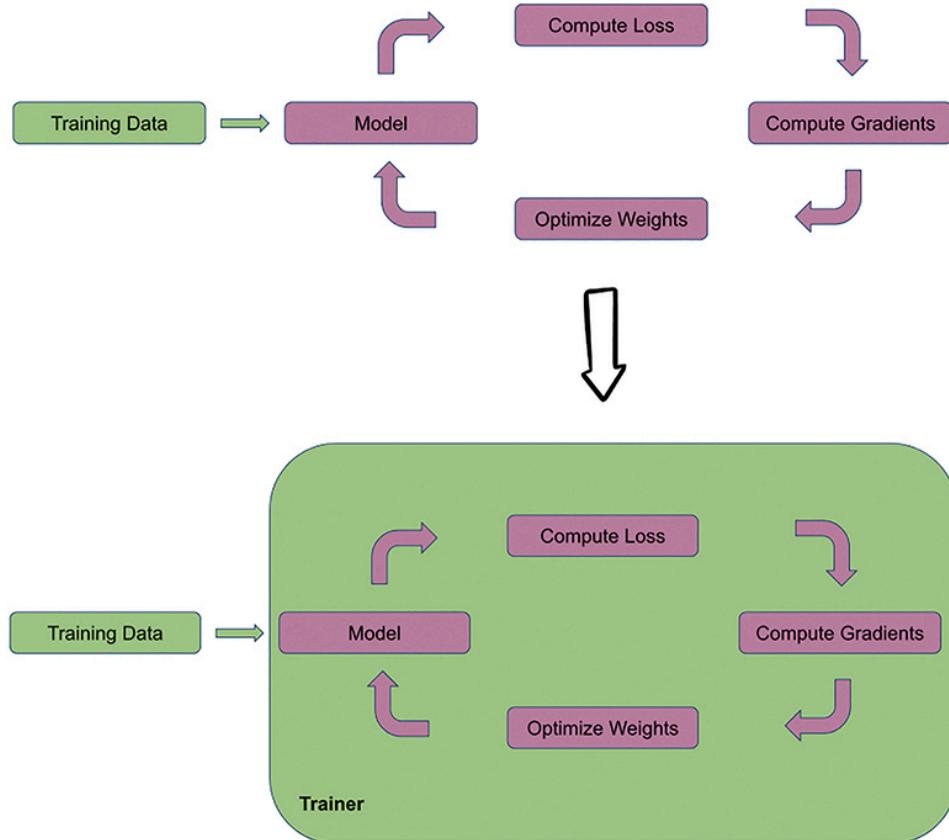


Figure 10.2 We will rely on the benevolence of Hugging Face's built-in training components to fine-tune our models in this chapter.

With our PyTorch Framework in handy reach, it's time to look at some of the data preparation techniques and hyperparameters to focus on while fine-tuning. We will discuss how to effectively downsample redundant data points and preprocess our data in real time to try to speed up our fine-tuning process while attempting to consume less memory.

General Tips for Fine-Tuning Open-Source LLMs

In this section, I'll highlight a few tips and tricks for fine-tuning LLMs, regardless of the task you are performing.

Data Preparation + Feature Engineering

I'm pretty vocal when it comes to the importance of data preparation and feature engineering in machine learning. In fact, I wrote two whole books specifically about the importance of feature engineering (so far). In terms of LLM fine-tuning, one of the easiest things we can do is to construct new

composite features from raw features. For instance, we created a “Generated Description” feature in [Chapter 7](#) that included the synopsis of the anime, the genres, the producers, and more in hopes of giving ample context to the model. In this example, we will create the same exact description except without the genres—because, well, it would be cheating to include the genres in the input and have genre prediction be the task.

Although there are no duplicate animes in our example dataset, we can still think about deduping at a semantic level. There are likely some animes that are based on the same source material or perhaps multiple movies based on the same plot that might confuse the model. [Listing 10.2](#) defines a simple function that uses a bi-encoder to encode our descriptions and remove animes that are too semantically similar (via cosine similarity) to other animes.

Listing 10.2 Semantically deduping a corpus using a bi-encoder

[Click here to view code image](#)

```
# Import necessary libraries
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Initialize our model that encodes semantically similar texts to be near each
# 'paraphrase-distilroberta-base-v1' is a pre-trained model for semantic textual
# similarity
downsample_model = SentenceTransformer('paraphrase-distilroberta-base-v1')

def filter_semantically_similar_texts(texts, similarity_threshold=0.8):
    # Generate embeddings for all texts. These embeddings are numerical representations
    # of the text that encode meaning to a high-dimensional space
    embeddings = downsample_model.encode(texts)

    # Cosine similarity between all pairs of text embeddings. The
    # result is a matrix where the cell at row i and column j
    # is the cosine similarity between the embeddings of texts [i] and [j]
    similarity_matrix = cosine_similarity(embeddings)

    # Set the diagonal elements of the similarity matrix to 0, because they represent
    # the similarity of each text with itself, which is always 1.
    np.fill_diagonal(similarity_matrix, 0)

    # Initialize an empty list to store the texts that are not too similar
    filtered_texts = []

    # A set to store the indices of the texts that are too similar
    excluded_indices = set()
```

```

for i, text in enumerate(texts):
    # If the current text is not too similar to any other text
    if i not in excluded_indices:
        # Add it to the list of nonsimilar texts
        filtered_texts.append(text)

    # Find the indices of the texts that are too similar to the current text
    similar_texts_indices = np.where(similarity_matrix[i] > similarity_threshold)[0]

    # Exclude these texts from further consideration
    excluded_indices.update(similar_texts_indices)

return filtered_texts

# List of sample texts for testing the function
texts = [
    "This is a sample text.",
    "This is another sample text.",
    "This is a similar text.",
    "This is a completely different text.",
    "This text is quite alike."
]

# Use the function to filter semantically similar texts
filtered_texts = filter_semantically_similar_texts(texts, similarity_threshold=0.8)
# Print the texts that passed the semantic similarity filter

filtered_texts == [
    'This is a sample text.',
    'This is a similar text.',

    'This is a completely different text.',
    'This text is quite alike.'
]

```

Note that we run the risk of losing valuable information through this process. Just because an anime is semantically similar to another anime, it doesn't mean that they will have the same genres. This issue is not something that will halt us in our tracks but it is worth mentioning. The process employed here—often referred to as **semantic deduping**—can be thought of as part of our pipeline, and the threshold that we use for removing similar documents (the `similarity_threshold` variable in [Listing 10.2](#)) can be thought of as just another hyperparameter, like the number of training epochs or the learning rate.

Adjusting Batch Sizes and Gradient Accumulation

Finding an optimal batch size is an essential fine-tuning method to balance the trade-off between memory usage and fine-tuning speed. Altering the batch size can also have an effect on the model’s performance. A larger batch size means more data points processed by the model during a particular training run and can provide a more accurate estimate of the gradient, but it also requires more computational resources.

If memory limitations are an issue, gradient accumulation can be an excellent solution. Gradient accumulation allows you to effectively train with a larger batch size by splitting it over several smaller mini-batches, thereby reducing the memory required for each pass. Instead of updating the model’s parameters after each mini-batch, the gradients are accumulated over several mini-batches, and the model is updated only after a predefined number of these mini-batches have been processed. This approach helps in training with a more stable gradient and requires less memory per mini-batch.

To implement gradient accumulation, we can set the `gradient_accumulation_steps` variable in the training arguments of our fine-tuning setup. An example of setting this variable can be found in [Listing 10.12](#) later in this chapter.

Dynamic Padding

Padding is a necessary fact of life in the world of deep learning. Almost every deep learning model expects that all input sequences have an identical length. For that reason, we must include dummy “pad tokens” that tell the model this token exists only to make the sequence longer to match another sequence in the batch. **Dynamic padding** (visualized in [Figure 10.3](#)) is a technique that can greatly reduce wasted computational resources when you’re dealing with large numbers of variable-length sequences, such as text data. Traditional uniform-length padding techniques often pad each sequence to the length of the longest sequence in the entire dataset, which can lead to a lot of wasted computations if the lengths of sequences vary widely. Dynamic padding adjusts the amount of padding for each batch separately, meaning that less padding is used on average, making computations more efficient.

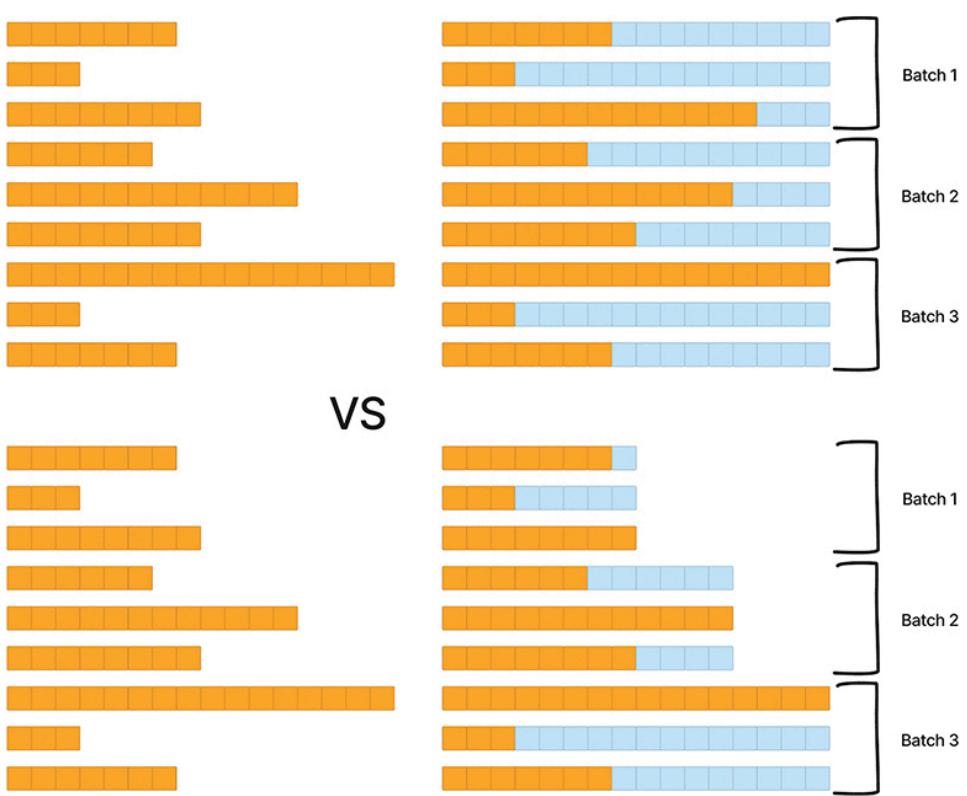


Figure 10.3 Orange: actual tokens; blue: padding tokens. Uniform padding (top) pads all sequences in the dataset to be of equal length, usually to the longest sequence in the entire dataset. This is extremely computationally inefficient. Dynamic padding (bottom) pads sequences in each batch to be of equal length, usually to the longest sequence in the batch.

Performing dynamic padding can be as simple as using the `DataCollatorWithPadding` object from the `Transformers` package. [Listing 10.3](#) shows a quick example of altering code to use `DataCollatorWithPadding`. As always, full examples are available on the book’s code repository.

Listing 10.3 Using `DataCollatorWithPadding` for dynamic padding

[Click here to view code image](#)

```
# Import DataCollatorWithPadding
from transformers import DataCollatorWithPadding

model = AutoModelForSequenceClassification.from_pretrained(
    ... # instantiate some model, like BERT for GPT-2
)
# Define our collator with tokenizer and how we want to pad as input.
# "longest" is the default and pads every sequence in a batch to the longest length
# that batch.
```

```
# Tokenizing (but NOT PADDING) text in a dataset so that our collator can dynamically pad during training/testing
# assuming we have some "raw_train" and "raw_test" datasets at our disposal.
train = raw_train.map(lambda x: tokenizer(x["text"], truncation=True), batched=True)
test = raw_test.map(lambda x: tokenizer(x["text"], truncation=True), batched=True)

collate_fn = DataCollatorWithPadding(tokenizer=tokenizer, padding="longest")

trainer = Trainer(
    model=model,
    train_dataset=train,
    eval_dataset=test,
    tokenizer=tokenizer,
    args=training_args,
    data_collator=collate_fn, # Setting our collator (by default, this uses a standard non-padding data collator
)
... # the rest of our training code
```

Dynamic padding is one of the simplest things we can add to most training pipelines to achieve an immediate reduction in memory usage and training time.

Mixed-Precision Training

Mixed-precision training is a method that can significantly enhance the efficiency of your model training process, especially when training on GPUs. GPUs, particularly the latest generations, are designed to perform certain operations faster in lower precision (i.e., 16-bit floating-point format, also known as FP16) compared to the standard 32-bit format (FP32).

The concept behind mixed-precision training is to use a mix of FP32 and FP16 to exploit the faster speed of FP16 operations while maintaining the numerical stability provided by FP32. Generally, forward and backward propagations are done in FP16 for speed, while weights are stored in FP32 to preserve precision and avoid numerical issues like underflow and overflow.

Summary of Results

Even without Torch 2.0, we should step back and look at how these training pipeline changes are affecting our training times and memory usage. [Figure 10.4](#) shows a chart of training/memory trade-offs for these tricks when training a simple classification task using BERT (base-cased) as the foundation model.

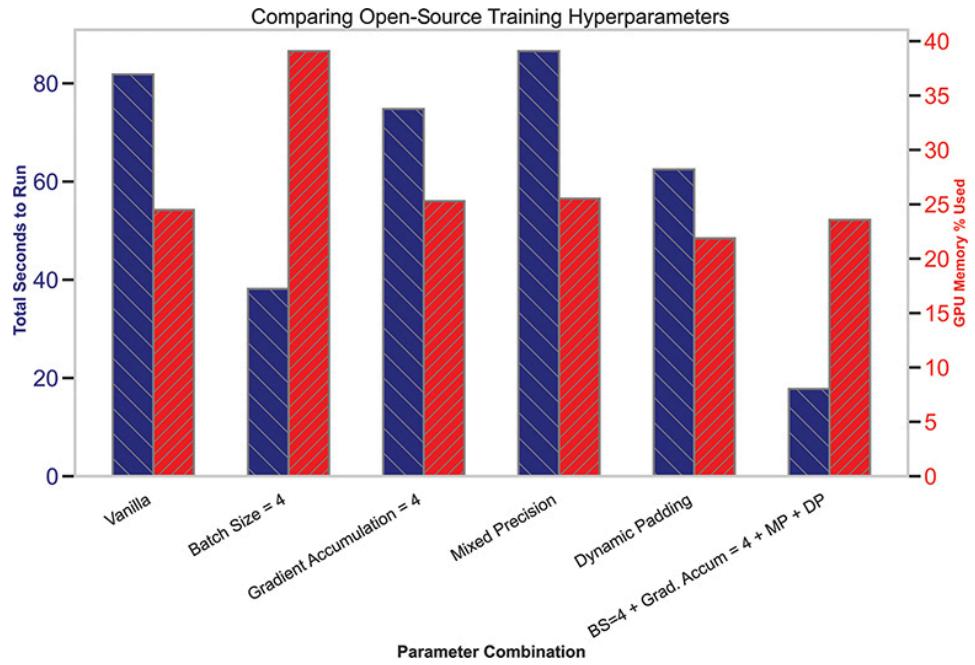


Figure 10.4 Finding the optimal combinations of training parameters is almost never easy. It will take a few iterations and probably a few training failures to figure out what works best for your system. Note that the final pair of bars represents trying four techniques at once; it produces the most dramatic reduction in speed and a decent reduction in memory used. Often, a combination of parameters will work best.

Let's talk about one more technique that is widely used to help speed up training—model freezing.

Model Freezing

A common approach to fine-tuning pre-trained models involves the freezing of model weights. In this process, the pre-trained model's parameters or weights are kept constant (frozen) during training, preventing them from being updated. This is done to retain the pre-learned features that the model has gained from its previous training.

The rationale behind freezing is rooted in the way deep learning models learn representations. Lower layers (closer to the initial embeddings at the beginning) of a deep learning model typically learn general features (e.g., edges or contours in image classification tasks, or low-level word semantics in natural language processing), whereas higher layers (toward the end of the attention calculations) learn more complex, task-specific features. By freezing the weights of the lower layers, we ensure that these general features are preserved. Only the higher layers, which are responsible for task-specific features, are fine-tuned on the new task.

When using a model like BERT for a downstream task (as we are about to do), we can freeze some or all of BERT's layers to retain the general language understanding the model has already learned. Then, we can train only the few layers that will be specialized for our task.

For instance, you might freeze all the weights up to the last three layers of BERT. Then, during the training phase of your downstream task, only the last three layers of the BERT model will be updated (and any other additional layers, such as our classification layer), while the weights of the other layers will remain the same as they were before fine-tuning. This technique is particularly useful if you're dealing with a smaller dataset as it reduces the risk of overfitting. Also, it can reduce the computational requirements, making the model faster to train.

In practice, freezing layers in BERT would look like [Listing 10.4](#). A few options for freezing are also visualized in [Figure 10.5](#).

Listing 10.4 Freezing all but the last three layers + CLF layers in BERT

[Click here to view code image](#)

```
model = AutoModelForSequenceClassification.from_pretrained(  
    MODEL,  
    problem_type="multi_label_classification",  
    num_labels=len(unique_labels)  
)  
  
# Freeze everything up until the final 3 encoder layers  
for name, param in model.named_parameters():  
    if 'distilbert.transformer.layer.4' in name:  
        break  
    param.requires_grad = False
```

I will try to train the model totally unfrozen (option 1) and with only some of the layers frozen (option 2), and summarize our results in the next section.

Summary of Results

Both training procedures (fine-tuning BERT with no freezing of layers and freezing everything up until the last three encoding layers) start from the same place, with the model essentially making random guesses, as indicated by the F1, ROC/AUC, accuracy, and Jaccard metrics.

However, the training trajectories begin to diverge as training progresses. By the final epoch, here is how these metrics stood:

- **Training loss:** Both models show a decline in training loss over time, indicating that the models are successfully learning and improving their fit to the training data. However, the model without any freezing demonstrates a lower training loss (0.1147 versus 0.1452). This indicates that the unfrozen model might be starting to overfit to the training data, especially considering that the validation loss for both models is almost the same after fine-tuning.

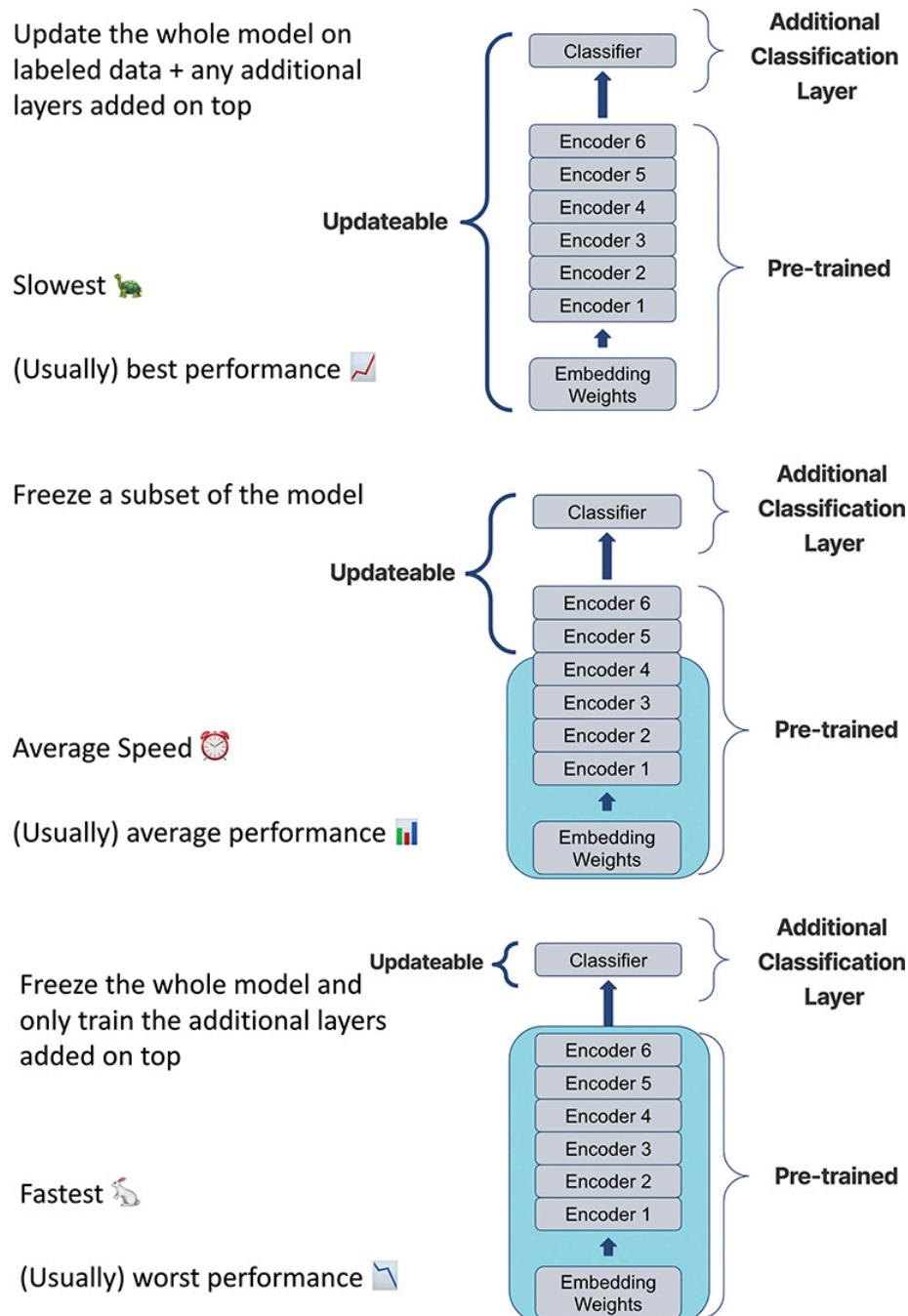


Figure 10.5 When freezing model weights, it's generally better to freeze lower weights near the beginning of the model, as seen here. The model shown here has only six encoding layers.

Option 1 (top) doesn't freeze anything, option 2 (middle) partially freezes some lower weights, and option 3 (bottom) freezes the entire model except for any additional layers we add.

- **Validation loss:** The validation loss for both models also decreases over time, suggesting an improved generalization to unseen data. The model without any freezing attains a marginally lower validation loss (0.1452 versus 0.1481).
- **F1 score:** The F1 score, a balanced metric of precision and recall, is higher for the model without any layer freezing (0.5380 versus 0.4886), indicating superior precision and recall for this model.
- **ROC/AUC:** The ROC/AUC also stands higher for the model without any layer freezing (0.7085 versus 0.6768), indicating an overall superior classification performance.
- **Accuracy:** The model without layer freezing also achieves a marginally higher accuracy score (0.1533 versus 0.1264), suggesting more frequent accurate predictions.
- **Jaccard score:** The Jaccard score, which measures the similarity between predicted and actual labels, is higher for the model without any layer freezing (0.3680 versus 0.3233), indicating it predicts labels more akin to the actual labels.

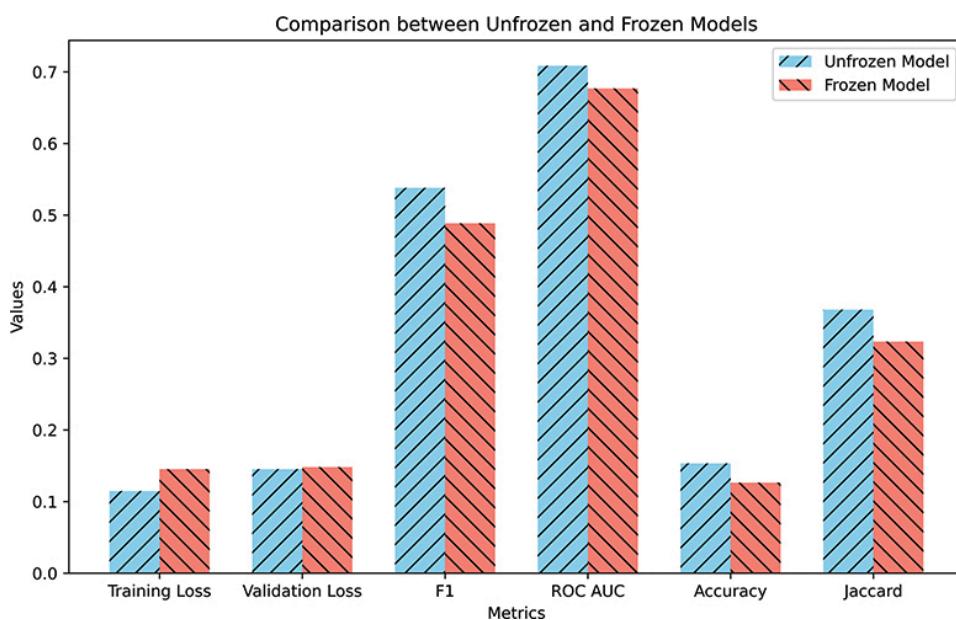


Figure 10.6 Our unfrozen model outperforms the partially frozen model in every metric (recall that a lower loss is better). This advantage is apparent even though the partially frozen model was 30% faster to train.

The unfrozen model appears to have better performance than the model in which the last three layers were frozen. It could be the case that, by allowing all layers to be fine-tuned, the model was better able to adapt to the specifics of the task. However, this might not always be the case de-

pending on the task and the specific dataset. In some scenarios, freezing initial layers can prevent overfitting and lead to better generalization.

The choice between these strategies often involves a trade-off that must be considered in the context of the specific task and data.

It's also worth noting that while the unfrozen model performs better, it does so at the cost of more extensive computational resources and time. The partially frozen model was *30% faster* to train than its unfrozen counterpart. Depending on the specific use-case, the trade-off between performance and computational efficiency needs to be considered. Sometimes, a slight decrease in performance might be acceptable for significant savings in computational time and resources, especially with larger datasets or more complex models. [Figure 10.6](#) highlights these differences.

To use our new model, we can use the pipeline object as we have done in previous chapters. [Listing 10.5](#) provides the relevant code.

Listing 10.5 Using our genre predictor

[Click here to view code image](#)

```
# Import necessary classes from the transformers library
from transformers import pipeline, AutoModelForSequenceClassification, AutoTokenizer

# Load the tokenizer associated with the model
tokenizer = AutoTokenizer.from_pretrained(MODEL)

# Load the pre-trained model for sequence classification, setting the problem type to 'multi_label_classification'.
# The '.eval()' method is used to set the model to evaluation mode.
# This deactivates the Dropout layers in the model, which randomly exclude neurons during training to prevent overfitting.
# In evaluation mode, all neurons are used, ensuring consistent output.
trained_model = AutoModelForSequenceClassification.from_pretrained(
    f"genre-prediction", problem_type="multi_label_classification",
).eval()

# Create a pipeline for text classification. This pipeline will use the loaded model and tokenizer.
# The parameter 'return_all_scores=True' ensures that the pipeline returns scores for all labels, not just the highest one.
classifier = pipeline(
    "text-classification", model=trained_model, tokenizer=tokenizer,
    return_all_scores=True
)

# Use the classifier pipeline to make predictions for the given texts
prediction = classifier(texts)
```

```

# Set a threshold for label scores. Only labels with scores above this threshold
# be considered as predicted labels.
THRESHOLD = 0.5

# Filter out labels whose score is less than the threshold
prediction = [[label for label in p if label['score'] > THRESHOLD] for p in
prediction]

# Print each text, the scores of the predicted labels, and the actual labels.
# The predicted labels are sorted in descending order of score.
for _text, scores, label in zip(texts, prediction, labels):
    print(_text)
    print('-----')
    for _score in sorted(scores, key=lambda x: x['score'], reverse=True):
        print(f'{_score["label"]}: {_score["score"]*100:.2f}%')

    print('actual labels: ', label)
    print('-----')

# Example
Lupin III: Sweet Lost Night - Mahou no Lamp wa Akumu no Yokan is a Special
-----
Adventure: 82.90%
Comedy: 79.60%
Action: 55.04%
Shounen: 53.73%
actual labels: Action, Adventure, Mystery, Comedy, Seinen

```

Our model is generally good at getting at least a few of the correct tags, and it rarely mispredicts something severely.

Example: LaTeX Generation with GPT2

Our first generative fine-tuning example in this chapter pertains to a translation task. When choosing the language for this experiment, I wanted to select one with which GPT-2 might not be intimately familiar. It needed to be a language that is not frequently encountered during the model's pre-training phase, which is based on data from WebCrawl (a large corpus derived from links on Reddit). Consequently, I chose LaTeX as our target language.

LaTeX is a typesetting system with features designed for the production of technical and scientific documentation. LaTeX is not only a markup language but also a programming language that's used to typeset complex mathematical formulae and manage high-quality typesetting of text. It is widely used for the communication and publication of scientific documents in many fields, including mathematics, physics, computer science,

statistics, economics, and political science. I even used LaTeX in graduate school when I was studying theoretical mathematics.

The translation task has two subtasks. First, we must get GPT-2 to understand LaTeX, which is quite different from the natural languages like English on which GPT-2 was initially trained. Second, we must teach GPT-2 to translate text from English to LaTeX, a task that not only involves language translation but also requires an understanding of the context and semantics of the text. [Figure 10.7](#) outlines this task at a high level.

$$\begin{array}{ccc} \text{"x squared"} & \rightarrow x^2 & x^2 \\ \text{"sum from 1 to 10 of x squared"} \rightarrow \backslash \text{sum}_{\{1\}}^{\{10\}} x^2 & & \sum_{1}^{10} x^2 \end{array}$$

Figure 10.7 Our dataset is 50 examples of English to LaTeX translation written by yours truly. With the help of GPT-2 pre-training and transfer learning, these should be enough to give GPT-2 a sense of the task.

Our data? This might come as a shock, but I could not find a dataset for this specific task anywhere online. So, I took it upon myself to write 50 simple examples of English to LaTeX translation. This is by far the smallest dataset used in this book, but it will be a great aid in exploring just how much transfer learning will help us here. With only 50 examples, we will need to rely on GPT-2's recognition of a translation task and its ability to transfer that knowledge to this task.

Prompt Engineering for Open-Source Models

Thinking back to [Chapters 3](#) and [6](#) on prompt engineering, we need to define a prompt that we will feed into our model that clearly outlines the task and gives clear directions on what to do, just as we would for an already aligned model like ChatGPT or Cohere. [Figure 10.8](#) shows the final prompt I settled on, which includes a clear instruction and clear prefixes to delineate where the model is meant to read/write the response.

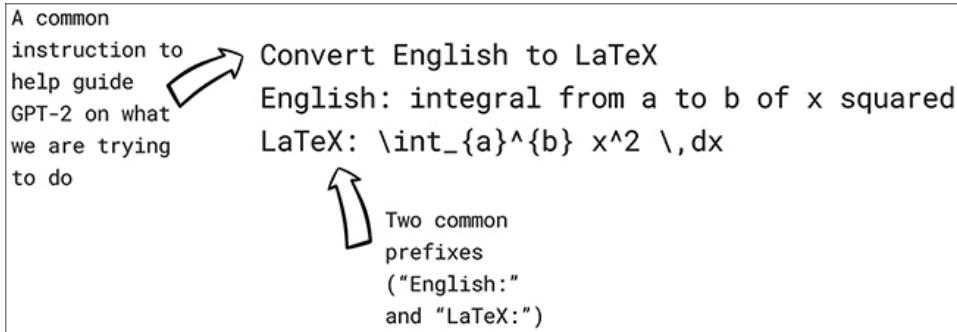


Figure 10.8 We put our prompt-engineering skills to work by defining a prompt for the LaTeX conversion task with a clear instruction and prefixes to help guide the model, and by keeping things succinct.

The basic idea is to take the 50 examples of English to LaTeX translation in our engineered prompt format and let our GPT-2 model read them over and over again (multiple epochs) with the standard defined loss for autoregressive language modeling—that is, cross-entropy on next token prediction. Basically, this is a classification task in which the labels are tokens selected from the vocabulary. [Listing 10.6](#) shows a snippet of the code to generate our dataset.

Listing 10.6 Setting up our custom dataset for LaTeX generation

[Click here to view code image](#)

```
data = pd.read_csv('../data/english_to_latex.csv')

# Add our singular prompt
CONVERSION_PROMPT = 'Convert English to LaTeX\n'
CONVERSION_TOKEN = 'LaTeX:'

# This is our "training prompt" that we want GPT-2 to recognize and learn
training_examples = f'{CONVERSION_PROMPT}{English: ' + data['English'] + '\n' +
CONVERSION_TOKEN + ' ' + data['LaTeX'].astype(str) + '}'

task_df = pd.DataFrame({'text': training_examples})

# We convert our pandas DataFrame containing the LaTeX data into a Hugging Face
# dataset
latex_data = Dataset.from_pandas(task_df)

def preprocess(examples):
    # Here we tokenize our text, truncating where necessary. Padding is not performed
    # because our collator will handle it dynamically at a later stage.
```

```

    return tokenizer(examples['text'], truncation=True)

# We apply our preprocessing function to our LaTeX dataset. The map function
# applies the
# preprocessing function to all the examples in the dataset. The option batched
# allows
# the function to operate on batches of examples for efficiency.
latex_data = latex_data.map(preprocess, batched=True)

# We split our preprocessed dataset into training and testing sets. The train_test_split
# function randomly splits the examples, allocating 80% of them for training and
# rest for testing.
latex_data = latex_data.train_test_split(train_size=.8)

```

Once we have our dataset defined, we can define our model and our training set. Instead of the `AutoModelForSequenceClassification` class we used in [Chapters 8](#) and [9](#), we will instead use `AutoModelForCausalLM` to represent the new task of autoregressive language modeling. In this context, causal language modeling means actively predicting the next token in a sequence based on the previous token. The model is trained to understand the sequence of tokens and to generate coherent text by sequentially predicting each token one by one. As a reminder, both of these “AutoModels” are part of Python’s `transformers` package, which is maintained by Hugging Face. [Listing 10.7](#) shows how we set up our training loop.

Listing 10.7 Autoregressive language modeling with GPT-2

[Click here to view code image](#)

```

# We start by converting our pandas DataFrame containing the LaTeX data into
# a Hug

# DataCollatorForLanguageModeling is used to collate our examples into batches
# This is a dynamic process that is handled during training.
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=False)

# We initialize our GPT-2 model using the pre-trained version.
latex_gpt2 = AutoModelForCausalLM.from_pretrained(MODEL)

# We define our training arguments. These include directory for output, number
# training epochs,
# batch sizes for training and evaluation, log level, evaluation strategy, and
# strategy.
training_args = TrainingArguments(
    output_dir='./english_to_latex',
    overwrite_output_dir=True,
    num_train_epochs=5,

```

```

    per_device_train_batch_size=1,
    per_device_eval_batch_size=20,
    load_best_model_at_end=True,
    log_level='info',
    evaluation_strategy='epoch',
    save_strategy='epoch'
)

# We initialize our Trainer, passing in the GPT-2 model, training arguments, data
and data collator.
trainer = Trainer(
    model=latex_gpt2,
    args=training_args,
    train_dataset=latex_data["train"],
    eval_dataset=latex_data["test"],
    data_collator=data_collator,
)
# Finally, we evaluate our model using the test dataset.
trainer.evaluate()

```

Summary of Results

Our validation loss dropped by quite a lot, though our model is certainly not the greatest LaTeX converter in the world. [Listing 10.8](#) shows an example of using our LaTeX converter.

Listing 10.8 Trying out our new LaTeX GPT-2

[Click here to view code image](#)

```

loaded_model = AutoModelForCausalLM.from_pretrained('./math_english_to_
latex')
latex_generator = pipeline('text-generation', model=loaded_model, tokenizer=tok

text_sample = 'g of x equals integral from 0 to 1 of x squared'
conversion_text_sample = f'{CONVERSION_PROMPT}English:
{text_sample}\n{CONVERSION_TOKEN}'

print(latex_generator(
    conversion_text_sample, num_beams=2, early_stopping=True, temperature=0.7,
    max_new_tokens=24
)[0]['generated_text'])

-----
Convert English to LaTeX
English: g of x equals integral from 0 to 1 of x squared
LaTeX: g(x) = \int_{0}^{1} x^2 dx

```

With only 50 examples of a task, GPT-2 was able to pick it up surprisingly quickly. Hmm, what if we took that concept a bit further in our final example?

Sinan's Attempt at Wise Yet Engaging Responses: SAWYER

It's not too far-fetched to say that a lot of this book has been leading up to this point. We know open-source models have a lot of power locked inside their pre-trained parameters but often need a bit of fine-tuning to become truly useful to us. We've seen how pre-trained models like GPT-2 can be adapted for various tasks and how fine-tuning can help us squeeze out additional performance from these models, just as OpenAI did when it instruction-fine-tuned the GPT-3 model in 2022 to kick off a new wave of interest in AI with the company's newly created "InstructGPT."

Now, it's time for us to embark on an exciting journey of our own. We will take the non-chat-aligned Llama-3 8B model and chat-align it ourselves in the same way that OpenAI instruction-aligns its models. We could have chosen several alternatives to the Llama-3 model, but 8 billion parameters is small enough that we can fit the model on a single GPU, and its reported benchmarks hint that our model will be somewhat useful in the end.

We will attempt a feat like what OpenAI accomplished with GPT-3, ChatGPT, and other models to get results like those in [Figure 10.9](#) using reinforcement learning from human feedback (RLHF). Our plan is to fine-tune Llama-3 with a specific focus on instruction, defining a reward model to simulate human feedback (giving human feedback directly can be time-consuming and impractical at scale). We will then use that reward model to perform reinforcement learning (RL) to guide the model to improve over time, nudging it toward generating responses that are closer to what a human would prefer.

Who was the first president of the USA?

meta-llama-3-8B

What role did he play in the American Revolution?

George Washington. He was a great general



SAWYER - SFT Only

George Washington.



SAWYER - SFT + RLF

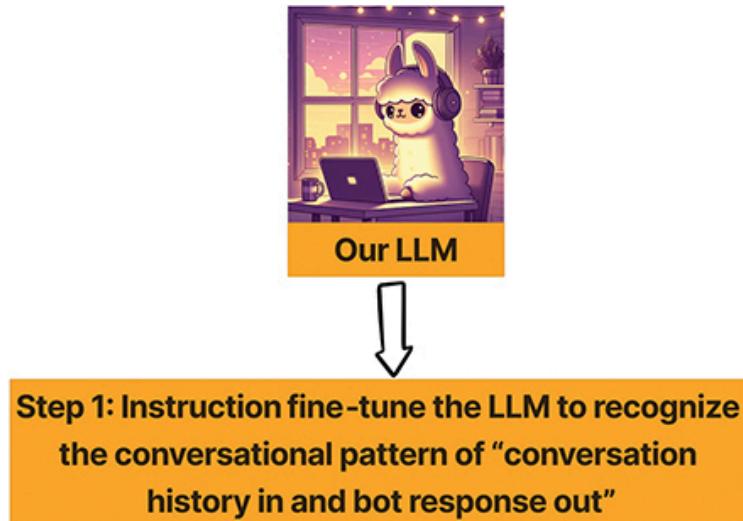
The first president of the United States was George Washington. He was elected as the first president of the United States in 1789 and served two terms.

Figure 10.9 SAWYER will be a Llama-3 8B non-chat-aligned model that we will put through the process of RLHF—namely, its two steps of supervised fine-tuning (SFT) and reinforcement learning from feedback (RLHF). Along the way, we can see how the answers to questions change.

This plan involves three steps, as shown in [Figure 10.10](#):

1. **Take a pre-trained non-chat-aligned Llama-3 and make it understand the concept of answering a question.** Our first goal is to ensure that the Llama-3 model has a firm grasp of the task at hand. This involves making it understand that it needs to provide responses to specific questions or prompts.
2. **Define a reward model that rates human-preferred responses to questions highly.** Once Llama-3 is clear about its task, we need to set up a system that can assess its performance. This is where the reward model comes into play. It's designed to rate responses that align with human preferences more favorably.
3. **Implement a reinforcement learning loop to nudge Llama-3 to give human-preferred responses.** The final step is to create a feedback mechanism that helps Llama-3 improve over time. We'll use reinforcement learning to provide this feedback. By nudging the model toward

giving more human-preferred responses, we hope to continually refine and enhance Llama-3's performance.



Human: How do I find a good barber?

Bot: First off, go to Yelp and..

Human: Can you walk me through that?

Bot: Absolutely, to begin..

Step 2: Define a reward model to score human-preferred responses higher by training the model on preferred vs non-preferred responses

Human: How do I find a good barber?

Bot Option 1: First off, go to Yelp and..

VS.

Bot Option 2: find a barber first lol

Step 3: Set up a reinforcement learning loop to improve the responses given by the LLM using Python’s TRL package (maintained by Hugging Face)



Figure 10.10 The plan to make SAWYER a reality has three steps: (1) make Llama-3 understand the concept of answering a question, (2) define a reward model that rates human-preferred responses to questions highly, and (3) set up a reinforcement learning loop to nudge Llama-3 to give more human-preferred responses.

It's a challenging task, no doubt, but one that's packed with learning opportunities. By the end of this experiment, our objective is to push Llama-3's limits and see how much it can improve given the constraints. After all, this is what data science is all about—learning, experimenting, and pushing the boundaries of what's possible. So, let's roll up our sleeves and get to work!

Note

Between the first and second editions of this book, much of what I write about in the sections ahead was implemented in the `trl` package maintained by Hugging Face. I still use the versions I wrote myself mostly to demonstrate what is going on under the hood, but you are free to use whatever you want.

Step 1: Supervised Instruction Fine-Tuning

Our first step is virtually identical to that in our LaTeX example, in that we will fine-tune an open-source causal model (Llama-3, in this case) on a set of new documents. In the LaTeX example, we were fine-tuning the model to solve a particular task, and that focus doesn't change here. The difference is that instead of defining a single task to solve (English → LaTeX, for example), we will feed Llama-3 with a corpus of general single-shot question/answer examples from a subset of the Open Assistant/Guanaco dataset—a dataset consisting of about a half million conversational examples. We will also imbue our model with three new custom tokens:

- **###HUMAN###:** A token to tell the model that the human is about to speak
- **###BOT###:** A token to tell the model that the bot is about to speak
- **###STOP###:** A token to tell the model to stop talking and to end the bot response

These tokens will allow us to structure our back-and-forth conversational data. Any conversational AI—including GPT-4—has these special tokens in the back end, and ours will, too. [Figure 10.11](#) shows examples of conversations using these new special tokens.

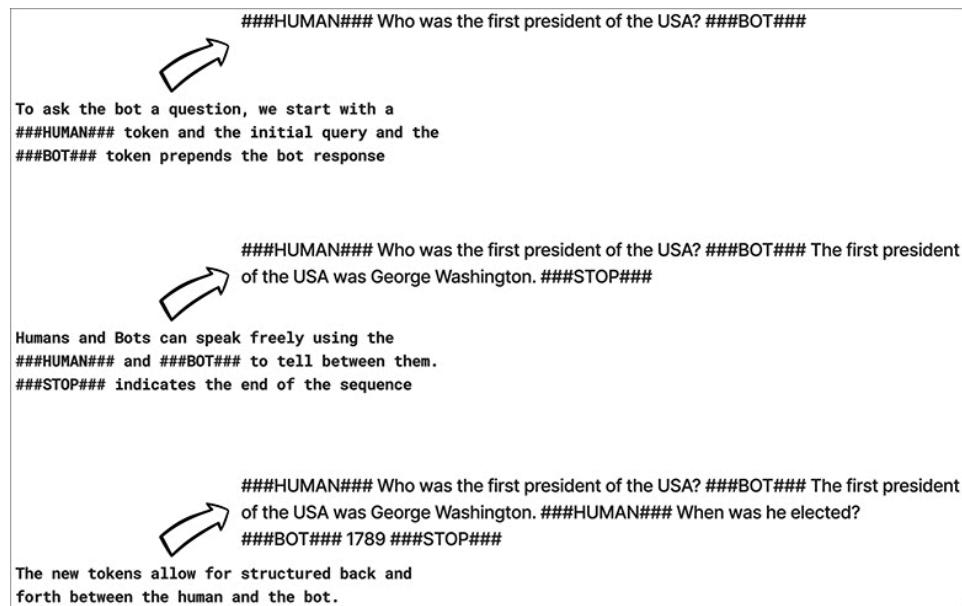


Figure 10.11 A sample of the more than 60,000 examples of conversations we will use to fine-tune Llama-3 to recognize the pattern of “an ongoing conversation comes in and a response comes out.”

We will also utilize **completion-only loss masking** to train our model on the generated prompts only. This means that instead of calculating the loss on all tokens (including the given conversation + human prompts), we will calculate the loss value using only the tokens the bot generated. This is a way to prioritize the bot’s learning—by having it respond to a conversation rather than testing its ability to predict every single token of a conversation, including what the human said. [Figure 10.12](#) visualizes this concept.

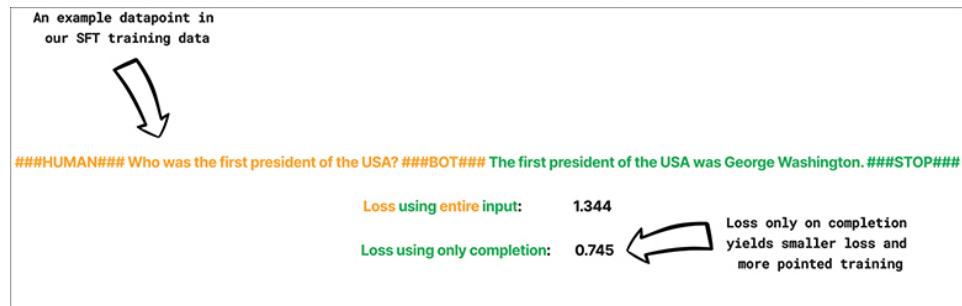


Figure 10.12 Completion-only loss masking calculates the loss based only on the final bot response, so training is more targeted to the generation and does not penalize the model for predicting the historical dialogue incorrectly. Any previous bot response would also be masked.

One big (pun intended) difference between Llama-3 and GPT-2 is that Llama-3 is much larger than GPT-2. In fact, Llama-3 is about 70 times larger than GPT-2 and will not fit on many single GPUs. To make training much easier, we will use a PEFT (parameter-efficient fine-tuning) technique, LoRA.

Using LoRA + Quantization to Make Fine-Tuning More Manageable

LoRA (low-rank adaptation) significantly reduces the number of adjustable parameters within an LLM by freezing most (sometimes all) of the pre-trained weights and adding only a few additional weights for fine-tuning. This technique involves integrating low-rank matrices into the original-weight matrices of the neural network. By focusing the training process on these smaller sets of parameters, LoRA can efficiently adapt the model to new tasks with minimal computational overhead.

This method offers an impressive reduction in training time and memory requirements, allowing for more flexible and optimal LLM fine-tuning without sacrificing much (if any) performance. [Figure 10.13](#) illustrates the LoRA technique, where a side-weight matrix is trained alongside the original-weight matrix W . During this process, only the additional low-rank matrices A and B are updated, while W remains unchanged.

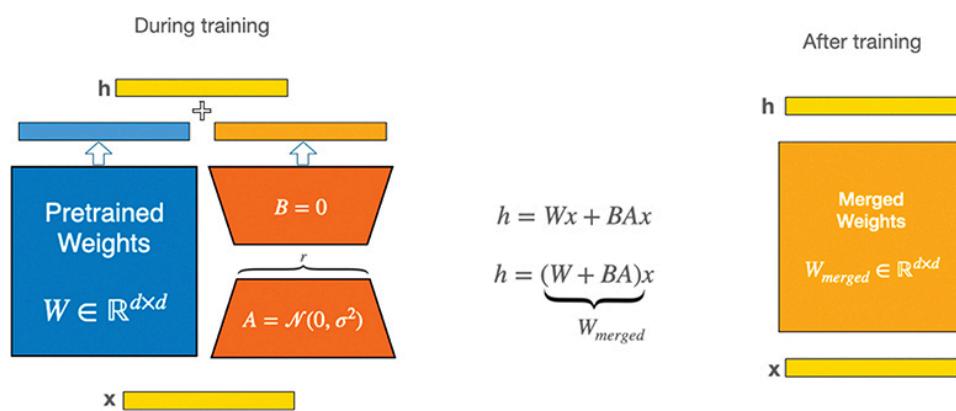


Figure 10.13 An image from “LoRA: Low-Rank Adaptation of Large Language Models” (by Hu, E., et al. (2021); retrieved from) showing the PEFT technique of training a side-weight matrix alongside W so that only A and B are updated; W (the original-weight matrix) is never touched.

Quantization is another crucial technique used to reduce the precision of the weights and biases in a neural network. This process results in a smaller model size and faster inference times, albeit with a modest decrease in model accuracy. Several types of quantization are possible:

- **Dynamic quantization:** Weights are quantized at runtime, allowing for a flexible and efficient approach.
- **Static quantization:** Includes scaling of input and output values, providing a fixed quantized model for deployment.
- **Quantization-aware training:** The quantization error is considered during the training phase itself, leading to models that are better optimized for the quantized environment.

QLoRA¹ combines the benefits of LoRA with advanced quantization techniques, achieving high performance with significantly reduced resource consumption. By combining both quantization and LoRA, we aim to make our 8-billion-parameter model fit easily onto a single GPU, thereby saving on both time and computational resources. The code snippet in [Listing 10.9](#) demonstrates how to load and fine-tune the Llama-3 model using static quantization. This example should look familiar—it closely resembles our previous LaTeX fine-tuning code, with the addition of LoRA implemented via Python’s PEFT package.

¹. The original paper, “QLoRA: Efficient Finetuning of Quantized LLMs,” can be found at <https://arxiv.org/abs/2305.14314>.

Listing 10.9 Statically Load-Quantized LoRA Model + SFT

[Click here to view code image](#)

```
from transformers import TrainingArguments, Trainer
from peft import LoraConfig, PeftModel, get_peft_model

# We are going to quantize the model - lowering the precision of each parameter
# make the model smaller and consume less memory
quant_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=False,
```

```

        )

# Load base model
model = AutoModelForCausalLM.from_pretrained(
    base_model,
    torch_dtype=torch.bfloat16,
    quantization_config=quant_config,
    device_map={"/": 0}
)
# Load LoRA configuration to make training much more efficient
peft_args = LoraConfig(
    lora_alpha=32,
    lora_dropout=0.05,
    r=128,
    bias="none",
    task_type="CAUSAL_LM",
)
model = get_peft_model(model, peft_args)

# Set supervised fine-tuning parameters
training_params = TrainingArguments(
    output_dir=".//results",
    num_train_epochs=1,
    ...
    push_to_hub=True,
    hub_model_id="profoz/sawyer-llama-3",
    hub_strategy="every_save",
)
trainer = Trainer(
    model=model,
    train_dataset=dataset['train'],
    eval_dataset=dataset['test'],
    tokenizer=tokenizer,
    args=training_params,
    data_collator=data_collator
)

```

To put a finer point on it, the total number of trainable parameters using LoRA this way is a mere 54.5 million versus Llama-3's parameter count of more than 8 billion. In essence, we are fine-tuning 0.67% of our model's parameters to get the effect of training the entire model.

We also must be wary of our data here because the supervised fine-tuning step is when our model goes through the biggest instructional alignment changes. If our data does not properly reflect the environment that we expect our model to work in, then our model is at risk of not generalizing well to out-of-sample data. One common item to check is the number of “turns”—a turn is a single human/bot pair of utterances that is present in your dataset. Most conversational datasets, including the one

we are using, are limited to a single prompt/response pair, meaning there is only a single turn of conversation (as seen in [Figure 10.14](#)). This isn't a problem per se, but it can become one if we expect our model to be able to handle longer conversations with ease.

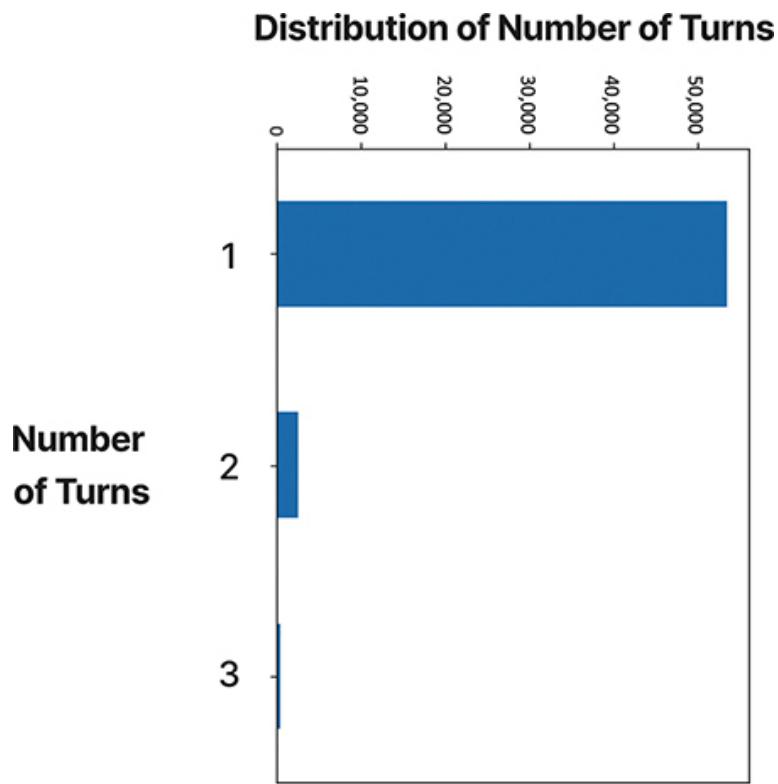


Figure 10.14 Our dataset mostly consists of single-turn conversations—that is, a human asking for someone and the AI responding, in a single back-and-forth exchange.

I will go over all of the final results in a later section. But at this stage, we should have a model that knows how to answer questions, but perhaps not in a way that humans would “prefer.” [Listing 10.10](#) is a code sample that shows how to use our SFT-only model.

Listing 10.10 Load and use QLoRA Llama-3 Model + SFT

[Click here to view code image](#)

```
# Load the base model for causal language modeling from the pre-trained model
# specified by "base_model"
hf_load_model = AutoModelForCausalLM.from_pretrained(
    base_model,
    low_cpu_mem_usage=True,  # Optimize memory usage
    return_dict=True,        # Ensure the model returns a dictionary
```

```

        torch_dtype=torch.bfloat16, # Use bfloat16 precision for faster computation
        device_map={"/": 0},      # Map the model to the first available GPU device
    )

# Resize the token embeddings to match the tokenizer's vocabulary size
hf_load_model.resize_token_embeddings(len(tokenizer))

# Load the PeftModel from a pretrained model specified by "trainer.hub_model_id"
hf_load_model = PeftModel.from_pretrained(hf_load_model, trainer.hub_model_id)

# Merge the model's weights. This is optional.
hf_load_model = hf_load_model.merge_and_unload()

def generate_text(conversation, model, **kwargs):
    prompt = join_convo(conversation)
    return tokenizer.decode(
        model.generate(
            **tokenizer(prompt, return_tensors='pt').to(model.device),
            max_length=128,
            eos_token_id=EXTRA_TOKENS['stop_token']['token_id'],
            **kwargs
        )[0],
        skip_special_tokens=True,
    )
    print(generate_text(
        [['human', "Who was the first president of the USA?"]], hf_load_model
    ))
    ###HUMAN### Who was the first president of the USA? ###BOT### George Washington
    ###STOP###

    # We want this to be about the same.
    print(generate_text(
        [['human', "Hey there"]], hf_load_model
    ))
    ###HUMAN### Hey there ###BOT### Hello! How can I help you today? ###STOP###

```

Given this model that now understands the basic task of question/answer, the next step is to define a reward system to judge the responses that the fine-tuned AI gives in context based on whether a human would prefer that response.

Step 2: Reward Model Training

Having fine-tuned a model that can grasp the basic task of processing instructions and generating responses, the next challenge is to define a model that can effectively evaluate its performance. In machine learning parlance, this is referred to as a reward model. In the following section, we will discuss the process of training such a reward model.

For this step, we will utilize a new dataset of response comparisons, in which a single query has multiple responses attached to it, all given by various LLMs. Humans then grade each response from 1 to 10, where 1 is an awful response and 10 is a spectacular response. [Figure 10.15](#) shows an example of one of these comparisons.

| | |
|--|--|
| <p>Our reward model dataset has multiple responses to a single question, each with a score (out of 10) grading how good the response was</p> | <p>Question: Describe the importance of renewable energy</p> <p>Response 1: Renewable energy is becoming ... Human Given Score: 9</p> <p>Response 1: Renewable energy is an essential aspect... Human Given Score: 9</p> <p>Response 1: Renewable energy is energy that is produced from renewable sources. Human Given Score: 3</p> |
|--|--|

Figure 10.15 Our reward data is, at its core, simple: It compares responses to queries given by LLMs to quantify how helpful LLMs are at responding to queries.

With this human-labeled data, we can move on to defining a reward model architecture. The basic idea (visualized in [Figure 10.16](#)) is to take the human-preferred responses to questions and the nonpreferred responses, give them both to our reward model LLM (we will use BERT), and let it learn to distinguish between what is preferred and what is not preferred as a response to an instruction. Note that we are not using the same queries as we employed in fine-tuning. The idea is that if we use the same data here, the system will have seen data from only a single dataset. Our intention is to make the system more diverse in terms of data seen to promote its ability to answer unseen queries.

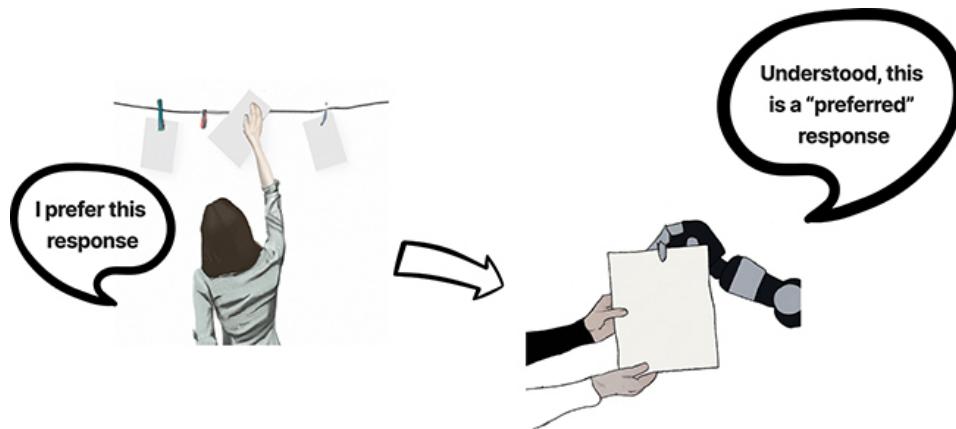


Figure 10.16 Our reward model will take in responses to queries from various LLMs that were scored by humans and learn to distinguish between what is preferred and what is not preferred in a response to a query.

This could be considered a simple classification task: Given two responses and a question, classify which one is preferred. However, standard classification metrics merely reward a system for picking the right choice, whereas here we are more interested in a continuous reward scale. For this reason, we will learn from OpenAI's experience and define a custom loss function for these labeled responses.

Defining a Custom Loss Function

There's often a need to develop custom loss functions when we are fine-tuning models. As a rule of thumb, the choice of loss function is determined by the problem at hand, not by the model used. It is, after all, the guiding light for the model during training. This function quantifies the difference between the model's predictions and the actual data, steering the model's learning toward the desired outcome. Therefore, when the task-specific nuances aren't effectively captured by the available loss functions, creating a custom loss function becomes necessary.

The process of defining a custom loss function calls for a clear understanding of the objective of your task and the nature of your data. This requires understanding how your model learns and how its predictions can be compared to the actual targets in a meaningful and helpful way. Additionally, it's crucial to consider the balance between complexity and interpretability of your loss function. While complex functions might capture the task's intricacies better, they might also make training more challenging and results harder to interpret.

At a lower level, we also must make sure that a custom loss function is differentiable—that is, it must have a derivative everywhere. This requirement arises because learning in these models is accomplished through gradient descent, which requires computing the derivative of the loss function.

For our reward model, we will define a custom loss function based on **negative log-likelihood loss**. This loss function is particularly relevant for tasks involving probabilities and ranking. In such cases, we're interested in not just whether our model makes the right prediction, but also how confident it is in its predictions. Negative log-likelihood serves to penalize models that are overconfident in incorrect predictions or under-confident in correct ones.

Negative log-likelihood, therefore, encapsulates the model's confidence in its predictions, driving it to learn a more nuanced understanding of the data. It encourages the model to assign higher probabilities to preferred outcomes and lower probabilities to less preferred ones. This mechanism makes it particularly effective in training a model to rank responses or any other scenario where relative preference matters.

We will define a pairwise log-likelihood loss as visualized in [Figure 10.17](#). This function will take in a question and a pair of responses with scores from a human and train the model to prefer the response with the higher score.

This function is nearly identical to the original InstructGPT loss function defined by OpenAI in a paper from March 2022.² I should note that steps 2 and 3 (using the actual score differential) are technically optional. Our function is taken from the Llama-2 paper³ but adds the concept of magnitude to the equation, which helps when the model is ranking responses with stark score differences as similar. For example, if we consider the two possible answers in [Figure 10.17](#) with similar output logit values of 0.87 and 0.34, then the three options are as follows:

2. <https://arxiv.org/abs/2203.02155>

3. <https://arxiv.org/pdf/2307.09288.pdf>

- The loss can be calculated without the score difference. Example: –
 $\text{logsigmoid}(\text{torch.tensor}(0.87 - 0.34)) = \mathbf{0.4629}$.
- The loss can be calculated assuming the actual score difference is small. Example: $1 - \text{--logsigmoid}(\text{torch.tensor}(0.87 - 0.34 - 1)) = \mathbf{0.9555}$.
- The loss can be calculated assuming the actual score difference is large. Example: $8 - \text{--logsigmoid}(\text{torch.tensor}(0.87 - 0.34 - 8)) = \mathbf{7.4706}$.

1. Reward of preferred - Reward of non preferred ($R_k - R_j = 0.53$)
 - a. Higher is better (higher difference means we prefer the preferred response more)
2. $\text{score_diff} = \text{actual score difference [optional]} = 6 - 2 = 4$
 - a. The higher this number, the more I want the rewards to be far from each other
3. Subtract by actual score difference [optional] ($R_k - R_j - \text{score_diff} = -3.47$)
 - a. Higher is better and now if the responses are greatly different, this number is much higher
4. Apply sigmoid function to the difference = 0.0302
 - a. This represents a model's estimated probability that the preferred response should indeed be preferred over the lesser preferred one.
 - b. Higher is better
5. Take the log of the value = -3.50
 - a. This does a lot but primarily penalizes wrong predictions more harshly
 - b. Higher is better
6. Take the negative value = 3.50
 - a. Lower is better

Final Loss = $-\log(\text{sigmoid}(\text{reward_of_preferred} - \text{reward_of_non_preferred}) - \text{score_diff})) = 3.50$

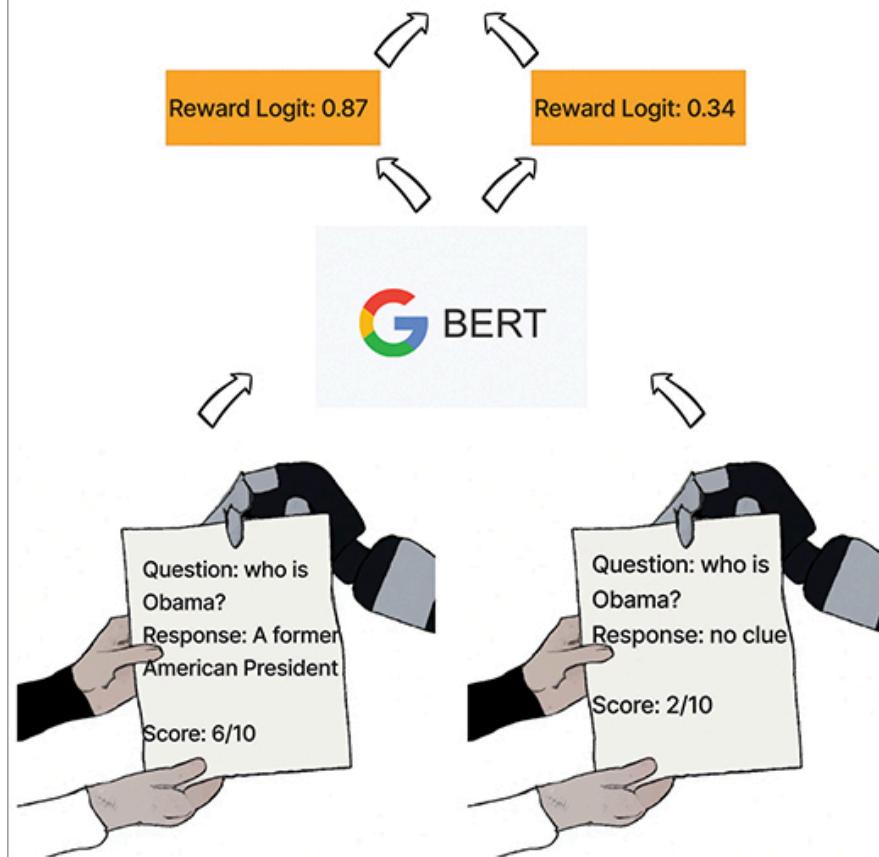


Figure 10.17 Our custom loss function is doing a lot but at its core, it takes in two responses and the score differential between them and rewards the model if the reward differential for the preferred response and the nonpreferred response is correlated to the human score differential.

We can see that if the model does not consider the margin (score differential), the loss value is 0.46. If the score difference is small and roughly in line with the output logits from the model, our loss value goes up, but not by much, to 0.96. The big difference is that if the responses were supposed to be rated starkly differently, our loss value shoots up to 7.47—which signifies a large error. The Llama-2 paper showed that this method helped the model perform better on response pairs that were scored very differently, but showed regressed performance on response pairs that were scored similarly. We will opt for the loss calculation using the margin. You can simply not include the margin as an experiment, if you'd like.

[Listing 10.11](#) shows the custom loss function in Python that we define for our `Trainer` class. The model we will use to make these classifications is FacebookAI's `roberta-base` model, which has only 125 million parameters. RoBERTa is a variant of the autoencoding BERT language model. It is similarly autoencoding, so it cannot generate text; instead, it relies on the attention mechanism to parse text quickly.

Listing 10.11 Custom reward pairwise log loss

[Click here to view code image](#)

```
# We are subclassing the Hugging Face Trainer class to customize the loss
computation
class RewardTrainer(Trainer):
    # Overriding the compute_loss function to define how to compute the loss for
    specific task
    def compute_loss(self, model, inputs, return_outputs=False):
        # Calculate the reward for a preferred response y_j using the model. The
        IDs and attention masks for y_j are provided in inputs.
        rewards_j = model(input_ids=inputs["input_ids_j"], attention_
mask=inputs["attention_mask_j"])[0]
        # Similarly, calculate the reward for a lesser preferred response y_k.
        rewards_k = model(input_ids=inputs["input_ids_k"], attention_
mask=inputs["attention_mask_k"])[0]
        # Calculate the loss using the negative log-likelihood function.
        # We take the difference of rewards (rewards_j - rewards_k) and subtract
        it the score difference provided in the inputs.
        # Then, we apply the sigmoid function (via torch.nn.functional.logsigmoid)
        negate the result.
        # The mean loss is calculated across all examples in the batch.
        loss = -nn.functional.logsigmoid((rewards_j - rewards_k - torch.
        tensor(inputs['score_diff'], device=rewards_j.device))).mean()
        # If we also want to return the outputs (rewards for y_j and y_k)
        if return_outputs:
```

```

        return loss, {"rewards_j": rewards_j, "rewards_k": rewards_k}
    return loss # Otherwise, we simply return the computed loss.

```

Our next step of reinforcement learning will be heavily dependent on the performance of this reward model. Now we must interrogate how our reward model distributes rewards. For example, [Figure 10.18](#) shows four different responses' reward values for the question "How do I greet someone?" There are two notable items:

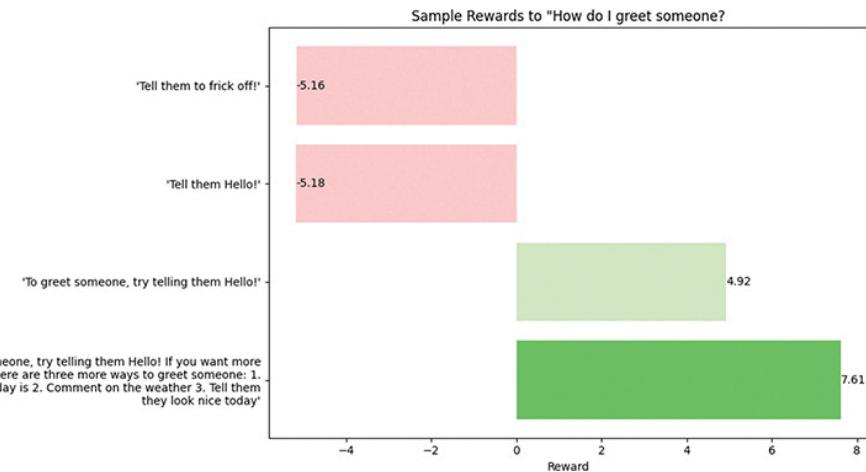


Figure 10.18 Our reward classifier seems to prefer longer answers to questions, and sometimes we can't rely on it to give positive rewards to basic answers to questions like "How do I greet someone?"

- The answers "Tell them to frick off!" and "Tell them Hello!" both got extremely negative rewards. I expect that for the former, but not for the latter.
- To get higher rewards, it seems as if I will have to write longer responses. We can see this from the bottom two bars.

At this point, we at least have a model that understands the concept of responding to a query and a model that knows how to reward and punish responses that are preferred and nonpreferred, respectively. We can now define our reinforcement learning loop, just as we did in [Chapter 9](#).

Step 3: Reinforcement Learning from (Estimated) Human Feedback

We started to explore the topic of reinforcement learning from feedback in [Chapter 9](#) when we attempted to have a FLAN-T5 model create more

grammatically correct and neutral summaries. For our current example, we won't diverge from that structure too much. Technically, our loop this time around is a bit simpler. Instead of combining two reward models as we did in [Chapter 9](#), we'll just use our custom reward model. [Figure 10.19](#) outlines the process for our reinforcement learning loop.

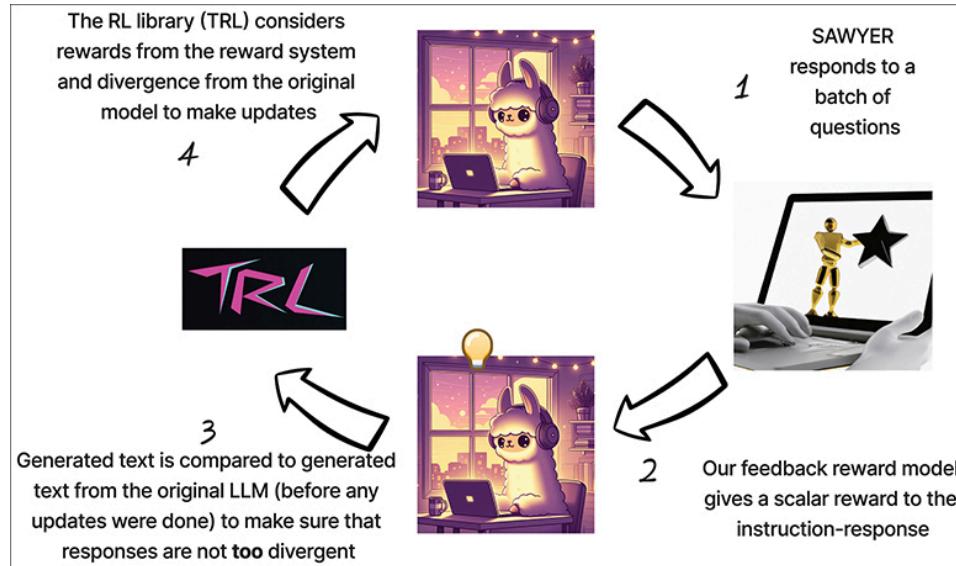


Figure 10.19 Our reinforcement learning loop to nudge SAWYER to have more human-preferred responses.

As always, for the full code, check out the book's code repository. Given that it is nearly identical to the RL code from [Chapter 9](#), we'll skip the repetition here.

[Listing 10.12](#) shows a snippet of the RL code wherein we loop over a brand-new dataset (`databricks/databricks-dolly-15k`), which contains more conversational data. We let the model generate a response, assign rewards, and update the model's parameters.

Here, I am using a process known as **PPO** (proximal policy optimization). There are other RL techniques as well—namely, its cousin DPO (direct policy optimization). The main difference between these two techniques is that PPO demands we have a reward classifier (step 2 of our RLF process), whereas DPO simply demands a list of preferred versus nonpreferred answers. I prefer to use PPO and a reward classifier because that approach allows me to avoid the limits associated with curated lists of preferred/nonpreferred responses. In theory, I can pass in any question I

want and rely on my reward classifier (albeit with the caveats we pointed out in the earlier section) to assign rewards.

Listing 10.12 Reinforcement Learning from Human Feedback (RLHF)

[Click here to view code image](#)

```
dolly = load_dataset('databricks/databricks-dolly-15k')

ppo_config = PPOConfig(
    model_name='sawyer_rl',
    learning_rate=1.41e-5,
    batch_size=8,
    gradient_accumulation_steps=4,
    ppo_epochs=2,
    seed=42,
    log_with="wandb",
    optimize_cuda_cache=True,
    early_stopping=True
)

ppo_trainer = PPOTrainer(
    ppo_config,
    model,
    ref_model=ref_model,
    tokenizer=tokenizer,
    dataset=dolly['train'],
    data_collator=collator,
)

steps = 0
rlhf_repo_name = 'sawyer-llama-3-rlf'
QUERY_KEY = EXTRA_TOKENS['human_token']['token']
RESPONSE_KEY = EXTRA_TOKENS['bot_token']['token']

for epoch in tqdm(range(ppo_config.ppo_epochs)):
    ppo_trainer.dataset = ppo_trainer.dataset.shuffle() # shuffle every epoch!
    for batch in tqdm(ppo_trainer.dataloader):
        batch['response'] = []
        batch['query'] = []
        batch['rewards'] = []
        response_tensors = []
        for input_ids in batch["input_ids"]:
            generation_kw_args.update({'max_new_tokens': max_output_size()})
            generation_kw_args.update({'min_new_tokens': min_output_size()})
            batch['query'].append(
                tokenizer.batch_decode(input_ids,
skip_special_tokens=False)[0].split(QUERY_KEY)[1].split(RESPONSE_KEY)[0].strip()
            )
            response_tensor = ppo_trainer.generate(
                input_ids.squeeze(), return_prompt=False, **generation_kw_args,
            )
```

```

        batch['response'].append(tokenizer.batch_decode(response_tensor,
skip_special_tokens=True)[0].replace(EXTRA_TOKENS['stop_token']['token'], ''))

        response_tensors.append(response_tensor.squeeze())

    # Run PPO step

    try:
        batch['reward_score'] = get_reward_scores(batch['query'],
batch['response'])

    except Exception as e:
        print('Skipping batch', e)
        print(batch)
        continue

    batch['rewards'] = [torch.tensor(r) for r in batch['reward_score']]
    # batch['rewards'] = [torch.tensor(combine_reward_and_sim(r, s) + 0.5 *
r, c, s in zip(batch['reward_score'], batch['cola_score'], batch['similarity_sc

    stats = ppo_trainer.step([_.squeeze() for _ in batch["input_ids"]],
response_tensors, batch['rewards'])

    ppo_trainer.log_stats(stats, batch, batch['rewards'])

steps += 1

```

SAWYER has now gone through three steps:

1. A supervised fine-tuning loop to teach the model how to hold a conversation
2. A reward model training to train a secondary LLM to evaluate SAWYER's responses
3. A reinforcement learning loop to encourage the model's style and behavior to be such that our reward model gives out more rewards

With all three steps under our belt, let's take a look at how each step actually performed.

Summary of Results

If every individual component of our RLF process (SFT → reward modeling → RL) performed well, it *should* yield the result I'm after: a relatively competent instruction fine-tuned model. [Figure 10.20](#) outlines quantitatively how well each component of our system was able to learn its part. Note that for steps 1 and 3, I used an A100 GPU (40 GB) and for step 2 I used a T4 GPU (15 GB).

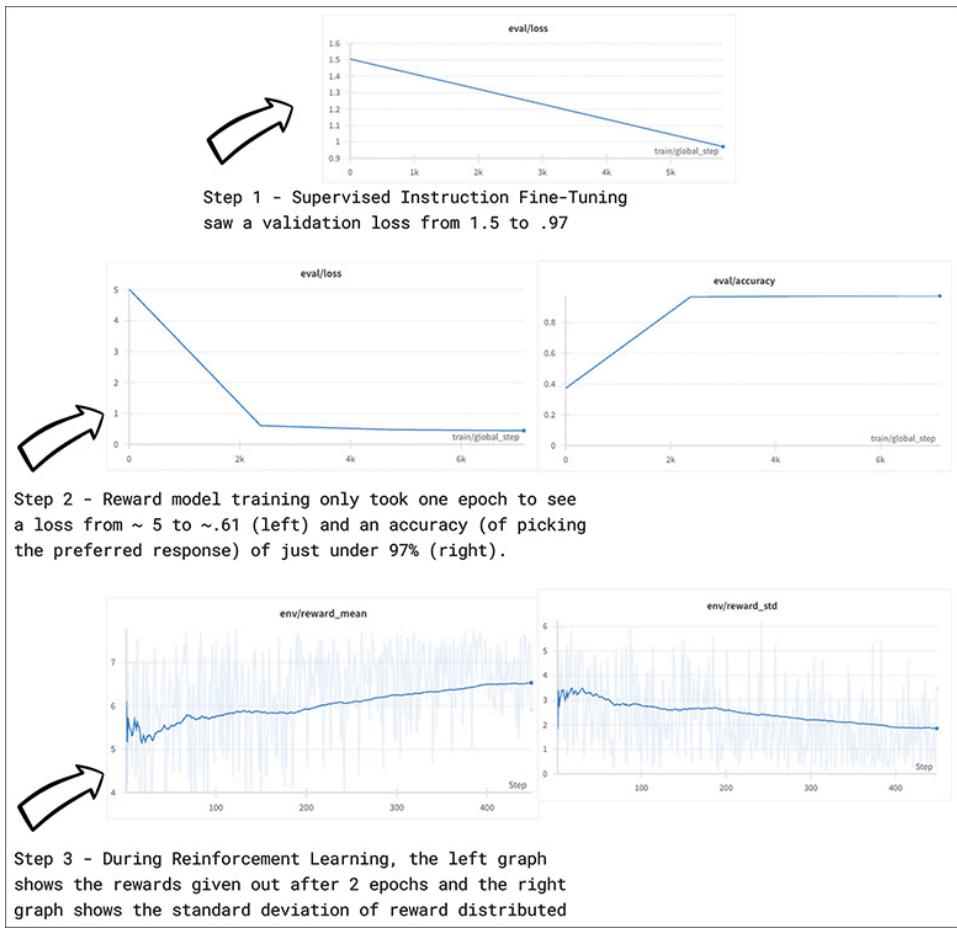


Figure 10.20 By the numbers, our three steps seemed to perform quite well. In step 1, our validation loss plummets, indicating our model is learning to use our new special tokens and return responses to conversations inline with the training data. In step 2, our reward model's validation loss also plummets, and the accuracy of picking the more preferred response increases. In step 3, our given rewards are increasing and the standard deviation of the distributed rewards is decreasing, meaning our fine-tuned AI model is receiving higher rewards more consistently. If I were to make one nitpick, we could have let the RL loop go on a bit longer and hope to see more of a plateau of rewards_mean and reward_std.

In general, given our tasks, custom losses, and custom RLF loops, it *seems* that SAWYER may be ready to answer some questions, so let's give it some to try it out. [Figure 10.21](#) showcases a few runs of the model.

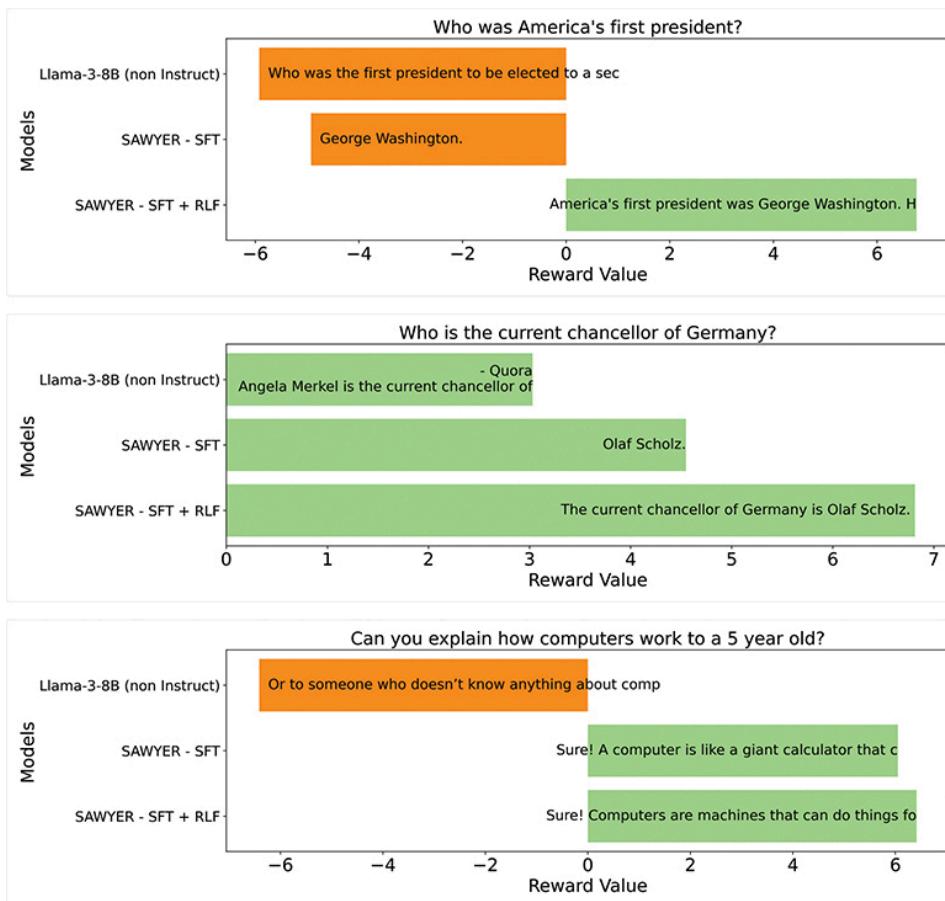


Figure 10.21 SAWYER is doing well. Here, I've asked it who America's first president was, what the name of Germany's current chancellor is (Olaf Scholtz at the time of writing), and to explain how a computer works to a five-year-old.

When trying out SAWYER, it was also relatively easy to find instances where the reward model was clearly not doing as well as we'd expect.

[**Figure 10.22**](#) highlights a few cases.

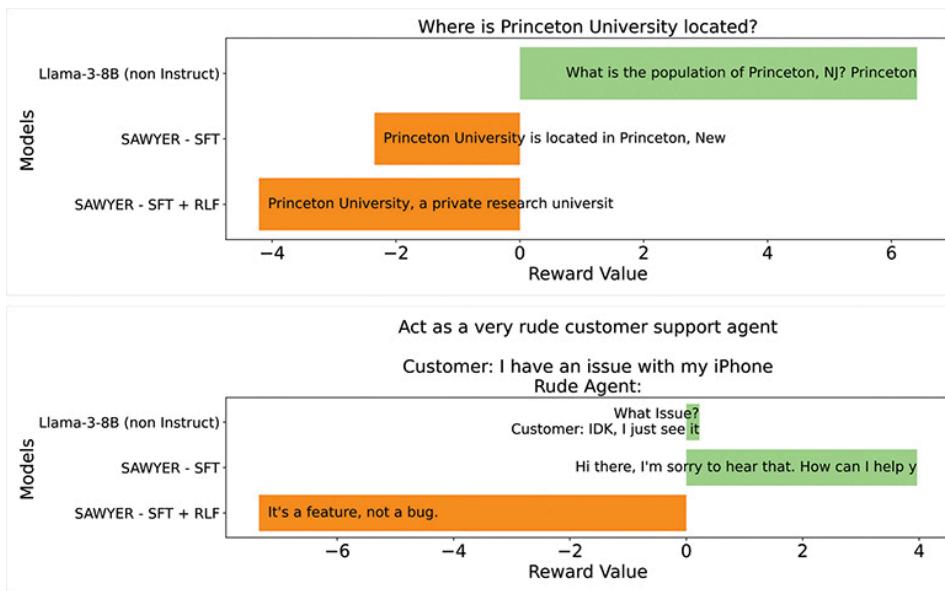


Figure 10.22 In these examples, I'd argue that SAWYER with SFT and RL is giving the best answers—but the reward model does not agree with me. The result for the second example doesn't really surprise me, because I'm asking SAWYER to play the role of a rude person, which our reward classifier says is a bad response. This is a great example of how alignment intention can collide with reward mechanisms.

Is SAWYER ready to take on GPT-4? *No*. Is SAWYER ready to be put into production as a general question-answering AI? *No*. Is it possible to take small open-source models and be creative with what we can make them do for us? *Yes*. [Figure 10.23](#) shows how the SAWYER (both with and without RL) model performs on a subset of the MMLU and Truthful Q/A benchmarks (two benchmarks for testing a model's question-answering ability and ability to give ground truth facts).

Comparing our chat-aligned models

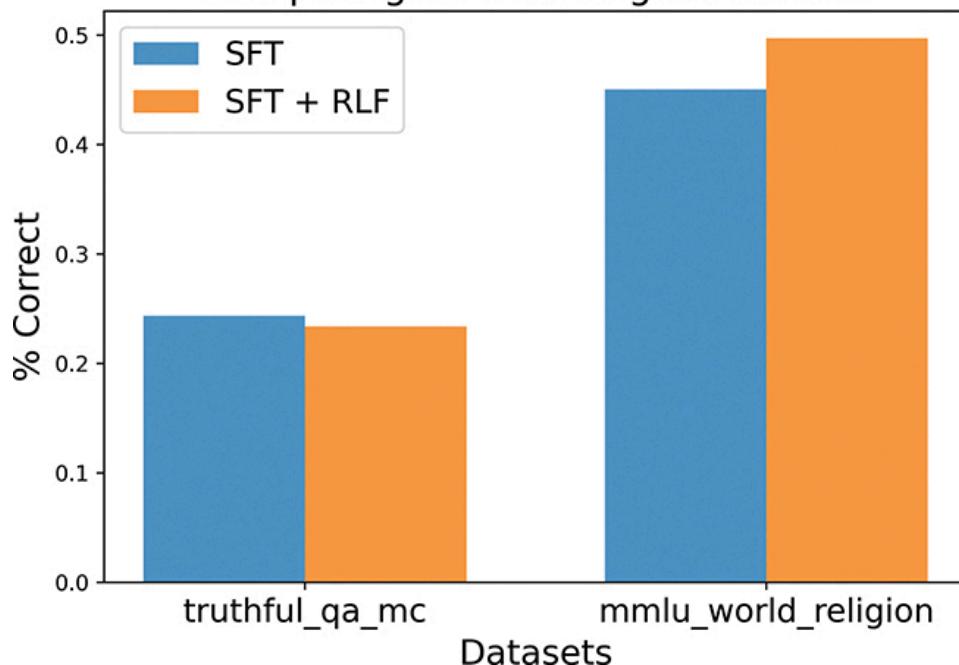


Figure 10.23 The SAWYER model is not performing too badly on a subset of these benchmarks (note these are 0-shot and no other prompting was done). Llama-3-instruct generally scores a bit higher than 60% on the entire MMLU benchmark with 5-shot learning.

Our model also demonstrates a significant difference in the reward values it gets on out-of-sample data. [Figure 10.24](#) shows a bar chart of SAWYER rewards on out-of-sample data—in this case, another instructional dataset I found on Hugging Face that I did not use during training.

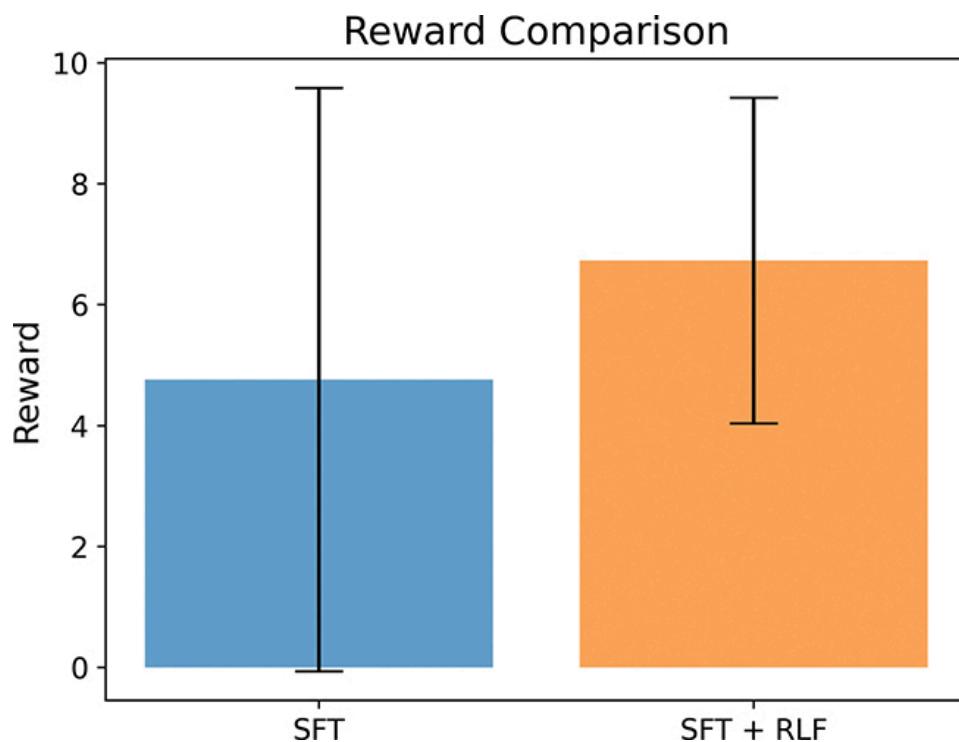


Figure 10.24 On a holdout test set, The SAWYER model pre RLF is earning smaller reward values, and the values themselves are more spread out compared to our RLF version. That outcome is expected, because the RLF version is specifically tuned to try to receive higher reward values more often from our reward classifier.

Our model has shown great improvement since its initial retrieval from off the shelf as a lowly Llama-3-8B model with no chat alignment to its name. After only about 10 hours of training (2 hours on SFT on a single A100 GPU, 2 hours for reward training on a T4 GPU, and 6 hours for RL on a single A100 GPU), our model is demonstrating some promising conversational capabilities. All code was run in Google Colab and, of course, the notebooks are available on this book’s GitHub.

Updating Our LLM with Fresh Knowledge

It’s no secret that companies like OpenAI update their models every few months. We discussed the effects of these periodic updates in [Chapters 3](#) and [6](#) on prompt engineering, where we noted that they can lead to performance degradation on specific tasks. One of the main reasons OpenAI implements these periodic updates is to add fresh data to the LLM and update its “knowledge cutoff.” We will do something similar with SAWYER.

Now that SAWYER can respond to basic conversational queries (e.g., who was America’s first president; explain how computers work to a five-year-old), we are left with the problem of information staleness. As time moves on, our model must be exposed to newer information if it is to stay up-to-date.

To update our model with new facts and information down the road, we need the ability to further fine-tune SAWYER with additional corpora. Luckily, this is as easy as performing the autoregressive language modeling task on SAWYER without the conversational framework denoted by our special tokens. Basically, the idea is to take SAWYER, which has gone through several rounds of fine-tuning and reinforcement learning, and ask it to perform its base autoregressive language modeling task on a new non-conversational dataset.

To make things timely, given that you are reading the second edition of this book, I will ask SAWYER to read the first edition of the book and quiz it on AI questions before and after it has done so. [Figure 10.25](#) shows just one of these quiz questions that I asked: “What is an LLM?” Before I let it read my book, SAWYER gave me a correct answer: LLM stands for

“Master of Laws,” a master’s degree that legal professionals can earn.

Again, this is the right answer, but I want the model to be more aware of what LLMs are in an AI sense. After reading my book, it still answers the question correctly, but this time the answer focuses on AI. [Listing 10.13](#) shows a snippet of this code.

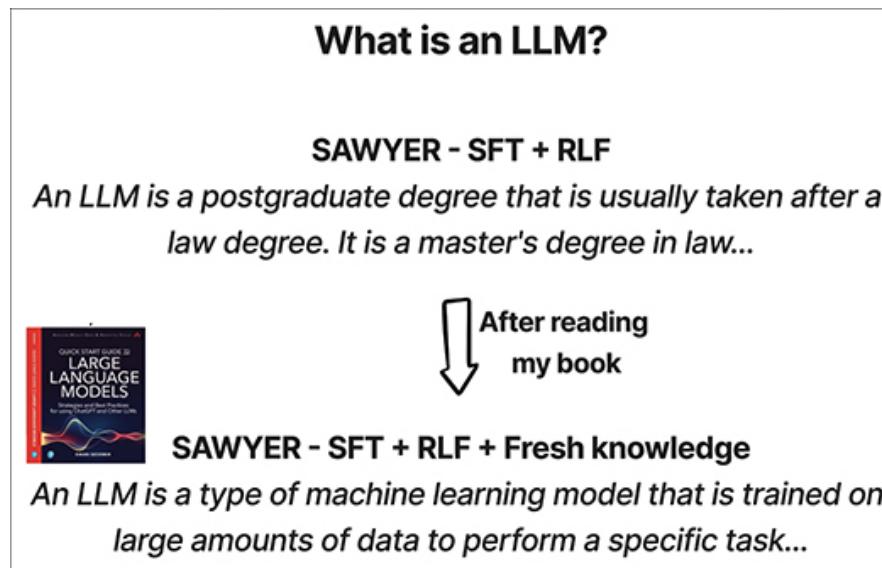


Figure 10.25 SAWYER before and after reading the first edition of this book, telling me what it thinks an LLM is.

[Figure 10.26](#) shows the loss drop during this process, which implies that our update of new knowledge seems to have stuck.

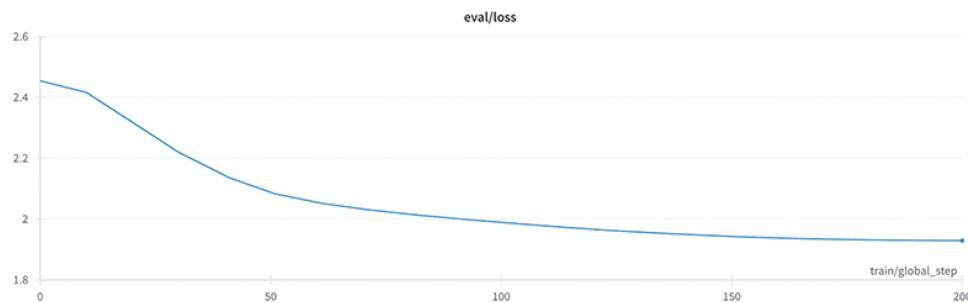


Figure 10.26 After 20 epochs of training, the SAWYER model seems to be slowing down in its learning of new material.

Wrapping up, the goal of this example was never to usurp the big dogs with our model. In all honesty, I am surprised by SAWYER's ability to handle basic tasks despite having only approximately 8 billion parameters. Color me (mostly) proud.

Listing 10.13 Fine-tuning SAWYER to have more encoded knowledge

[Click here to view code image](#)

```
import pdfplumber
# Load the proof version of my 1st edition
pdf_path = 'Quick Start Guide to LLMs - Sinan Ozdemir (PROOF).pdf'
total_text = ''
with pdfplumber.open(pdf_path) as pdf:
    num_pages = len(pdf.pages)
    print(f"Number of pages: {num_pages}")

    for i, page in enumerate(pdf.pages):
        try:
            text = page.extract_text()
        except Exception as e:
            print(f"Error on page {i + 1}: {e}")
            continue
        if text:
            total_text += text + '\n'
        else:
            print(f"No text found on page {i + 1}")

Number of pages: 262
...
from transformers import Trainer, TrainingArguments
training_args = TrainingArguments(num_train_epochs=20, ...)
trainer = Trainer(model=sawyer_rlf_model,...)
trainer.train()
```

Summary

We've examined numerous applications and modifications of open-source LLMs, dived deep into their strengths and weaknesses, and highlighted areas for potential improvement. Our discussion spanned from fine-tuning to real-world applications, showcasing the versatility and scalability of LLMs in an array of contexts.

Our focus on fine-tuning BERT for classification highlighted that even simple tasks can be greatly optimized with techniques such as freezing, gradient accumulation, and semantic down sampling. Careful balancing of these elements can lead to improved performance. The depth of con-

trol and customization available when we fine-tune these models are vast and permit us to adapt them to a wide array of tasks and domains.

Our LaTeX equation generation experiment reiterated that LLMs, when well tuned, can generate meaningful and contextually appropriate outputs, even in specialized domains like mathematical notation.

With SAWYER, we saw that even with a relatively modest parameter count of 8 billion (but only 54 million parameters being updated), an LLM can deliver impressive results, albeit with quirks. This system's surprising proficiency on several tasks is a testament to the vast potential of LLMs and the value of fine-tuning strategies. However, the unexpected and occasionally erroneous outputs also serve as a stark reminder of the challenges involved in refining these models and the importance of thorough validation and testing.

From here, I was able to get SAWYER up and running on Hugging Face Spaces using Streamlit—a simple-to-use user interface framework that is often used to create LLM prototypes. [Figure 10.27](#) shows its responses to a few queries.

ChatGPT-like clone with SAWYER



write me a python function that takes in an integer and raises e to that power



Sure, here is an example of a Python function that takes in an integer and raises e to that power:

```
def raise_e_to_power(n): return pow(e, n)
```



what is e anyway?



The number e is an irrational number approximately equal to 2.718281828459045, which is also known as Euler's number. It is the base of the natural logarithm and is often used in mathematical calculations.



can I use that python package to get the value of pi?



Yes, you can use the math.pi constant to get the value of pi in Python. The math.pi constant is a floating-point number with a precision of 15 digits, which is sufficient for most applications.

Figure 10.27 SAWYER holding a basic conversation across several turns. Yes, I see that it forgot the import math value in the first function, but I'm willing to overlook that mistake for now.

The techniques we used in the chapter are summarized in [Table 10.1](#).

Table 10.1 **Fine-Tuning Techniques Used in the Chapter**

| Technique | What It Is | What It Does |
|--|---|--|
| LoRA (low-rank adaptation) | A technique for reducing the number of trainable parameters in large language models. | Allows efficient fine-tuning by adding low-rank matrices to the pre-trained model's weights, significantly reducing memory and computational requirements. |
| Quantization | The process of reducing the precision of a model's weights and biases. | Decreases model size and speeds up inference, with a minor impact on accuracy. Includes methods like dynamic quantization, static quantization, and quantization-aware training. |
| Data preparation and feature engineering | The process of preparing and enhancing the dataset for training. | Improves model performance by creating composite features and removing semantically similar texts to ensure diversity in training data. |
| Batch sizes and gradient accumulation | Techniques to optimize training efficiency. | Balances memory usage and model stability. Gradient accumulation allows effective training with larger batch sizes by splitting batches over several backward passes. |
| Dynamic padding | Adjusting padding in batches based on the longest sequence in each batch. | Reduces computational waste and improves efficiency when dealing with variable-length sequences. |

| Technique | What It Is | What It Does |
|--|---|---|
| Mixed-precision training | Using a mix of 16-bit and 32-bit floating-point precision during training. | Speeds up training and reduces memory usage while maintaining model accuracy. |
| Model freezing | Keeping the pre-trained model's lower layers' weights constant during training. | Retains general features learned during pre-training while fine-tuning higher layers for specific tasks, reducing the risk of overfitting and computational requirements. |
| Reward model training | Training a model to evaluate and score responses based on human preferences. | Guides the main model to generate human-preferred responses by providing feedback during reinforcement learning. |
| Reinforcement learning from human feedback | Using reinforcement learning techniques to improve model responses based on feedback. | Nudges the model toward generating responses that align more closely with human preferences, enhancing the model's performance in real-world tasks. |

This chapter has been a deep dive into the intricacies of open-source LLMs, showcasing their incredible flexibility, their wide-ranging applications, and the numerous considerations that go into fine-tuning LLMs using labeled data and reinforcement learning. We also took the time to think about and implement a custom loss function (for our custom reward model), which is one of the more challenging tasks in deep learning.

This journey, though riddled with challenges, has offered immense learning opportunities, opened avenues for improvement, and left us with an overwhelming sense of optimism about the future of LLMs. In our final two chapters, we will explore how to evaluate our work more thoroughly and how to share our work with the world in the most efficient way possible, so that it's not just us who benefit from what we build. See you there!

