9

Moving Beyond Foundation Models

Introduction

In previous chapters, we have focused on using or fine-tuning pre-trained models such as GPT, BERT, and Claude to tackle a variety of natural language processing and computer vision tasks. While these models have demonstrated state-of-the-art performance on a wide range of benchmarks, they may not be sufficient for solving more complex or domain-specific tasks that require a deeper understanding of the problem.

In this chapter, we explore the concept of constructing novel LLM architectures by combining existing models. By combining different models, we can leverage their strengths to create a hybrid architecture that either performs better than the individual models or performs a task that wasn't possible previously.

We will be building a multimodal visual question-answering system, combining the text-processing capabilities of BERT, the image-processing capabilities of a Vision Transformer (yes, those exist), and the text-generation capabilities of the open-source GPT-2 to solve visual reasoning tasks. We will also explore the field of reinforcement learning and see how it can be used to fine-tune pre-trained LLMs. Let's dive in, shall we?

Case Study: Visual Q/A

Visual question-answering (VQA) is a challenging task that requires understanding and reasoning about both images and natural language (visualized in **Figure 9.1**). Given an image and a related question in natural language, the objective is to generate a textual response that answers the question correctly. We saw a brief example of using pre-trained VQA systems in **Chapter 6** in a prompt chaining example, but now we are going to make our own!

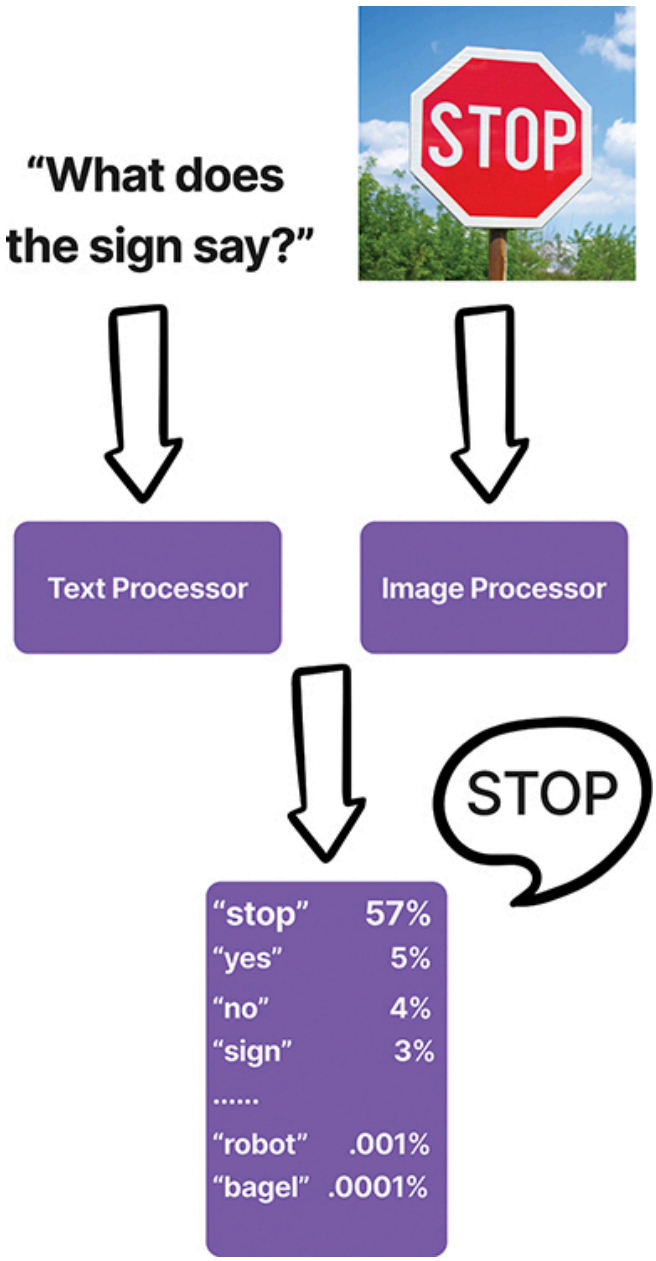


Figure 9.1 A visual question-answering (VQA) system generally takes in two modes (types) of data—image and text—and returns a human-readable answer to the question. This image outlines one of the most basic approaches to this problem, with the image and text being encoded by separate encoders and a final layer predicting a single word as an answer. Image: panicattack/123RF (stop sign)

In this section, we focus on constructing a VQA+LLM system by using existing models and techniques. We start by introducing the foundational models used for this task: BERT, ViT, and GPT-2. We then explore the combination of these models to create a hybrid architecture capable of processing both textual and visual inputs and generating coherent textual outputs.

We also demonstrate how to fine-tune the model using a dataset specifically designed for VQA tasks. We use the VQA v2.0 dataset, which contains

a large number of images along with natural language questions about the images and corresponding answers. We explain how to prepare this dataset for training and evaluation and how to fine-tune the model using the dataset.

Introduction to Our Models: The Vision Transformer, GPT-2, and DistilBERT

In this section, we introduce three foundational models that we will use in our constructed multimodal system: the Vision Transformer, GPT-2, and DistilBERT. These models, while not currently considered state-of-the-art options, are nonetheless powerful LLMs and have been widely used in various natural language processing and computer vision tasks. It's also worth noting that when we are considering which LLMs to work with, we don't always have to go right for the top-shelf LLMs, as they tend to be larger and slower to use. With the right data and the right motivation, we can make the smaller LLMs work just as well for our specific use-cases.

Our Text Processor: DistilBERT

DistilBERT is a distilled version of the popular BERT model that has been optimized for speed and memory efficiency. This pre-trained model uses knowledge distillation to transfer knowledge from the larger BERT model to a smaller and more efficient one. This allows it to run faster and consume less memory while still retaining much of the performance of the larger model.

DistilBERT should have prior knowledge of language that will help during training, thanks to transfer learning. This allows it to understand natural language text with high accuracy.

Our Image Processor: Vision Transformer

The Vision Transformer (ViT) is a Transformer-based architecture that is specifically designed for understanding images. ViT was developed by the same team that invented both the Transformer and BERT; it is used to extract features from images. A newer model that has gained popularity in recent years, it has been shown to be effective in various computer vision tasks.

Like BERT, ViT has been pre-trained on a dataset of images known as Imagenet, a large, publicly available database of annotated images. Just as BERT's pre-training helps it understand language for downstream tasks, ViT has prior knowledge of images that should help during training. This allows ViT to understand and extract relevant features from images with higher accuracy than other, non-pre-trained models.

When we use ViT, we should try to use the same image preprocessing steps that the model used during pre-training, so that it will have an easier time learning the new image sets. This is not strictly necessary and has both pros and cons.

Pros of reusing the same preprocessing steps are as follows:

1. **Consistency with pre-training:** Using data in the same format and distribution as was used during its pre-training can lead to better performance and faster convergence.
2. **Leveraging prior knowledge:** Since the model has been pre-trained on a large dataset, it has already learned to extract meaningful features from images. Using the same preprocessing steps allows the model to apply this prior knowledge effectively to the new dataset.
3. **Improved generalization:** The model is more likely to generalize well to new data if the preprocessing steps are consistent with its pre-training, as it has already seen a wide variety of image structures and features.

Cons of reusing the same preprocessing steps include the following:

1. **Limited flexibility:** Reusing the same preprocessing steps may limit the model's ability to adapt to new data distributions or specific characteristics of the new dataset, which may require different preprocessing techniques for optimal performance.
2. **Incompatibility with new data:** In some cases, the new dataset may have unique properties or structures that are not well suited to the original preprocessing steps, which could lead to suboptimal performance if the preprocessing steps are not adapted accordingly.
3. **Overfitting to pre-training data:** Relying too heavily on the same preprocessing steps might cause the model to overfit to the specific characteristics of the pre-training data, reducing its ability to generalize to new and diverse datasets.

We will reuse the ViT image preprocessor for now. **Figure 9.2** shows a sample of an image before preprocessing and the same image after it has gone through ViT's standard preprocessing steps.

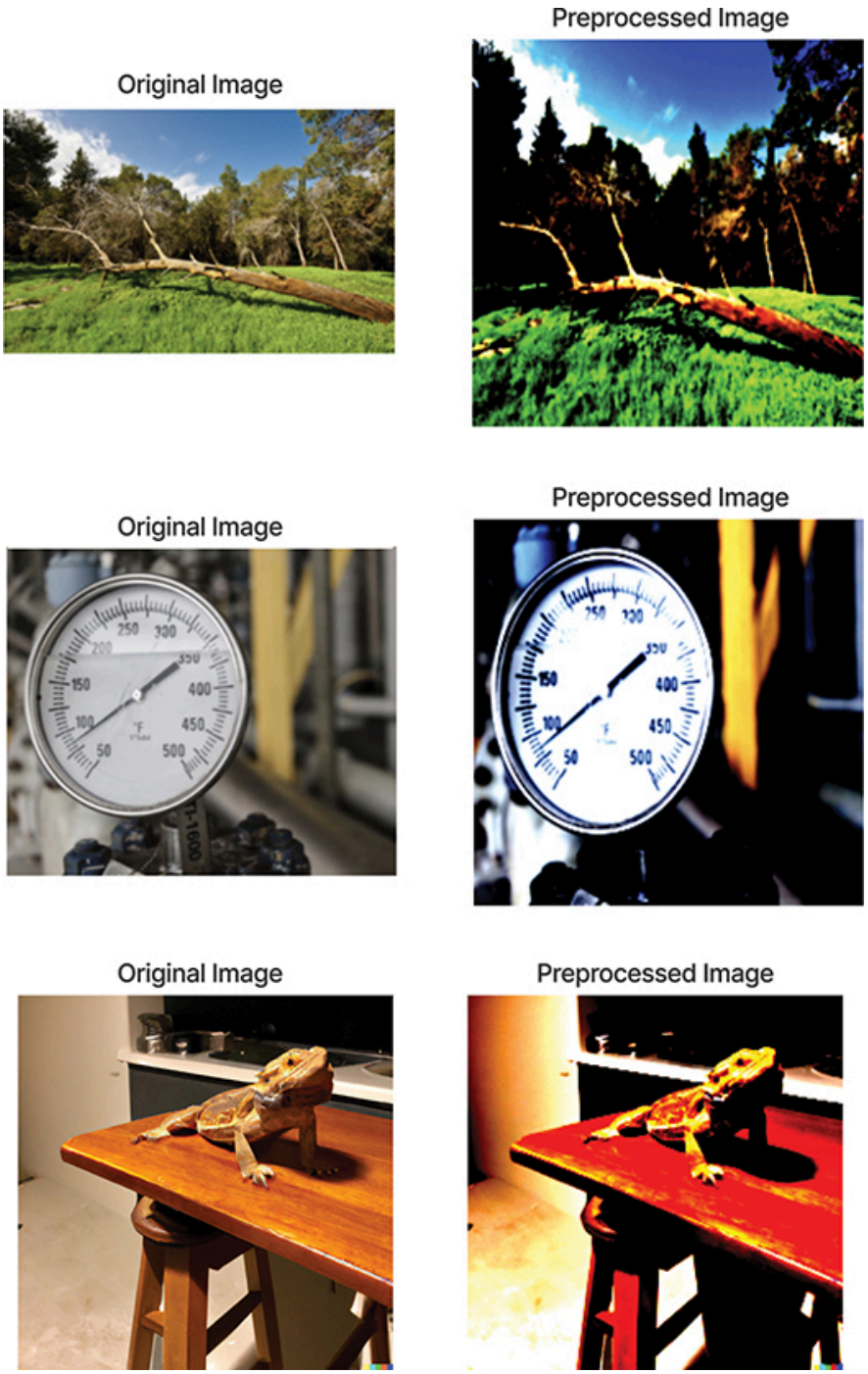


Figure 9.2 Image systems like the Vision Transformer (ViT) generally have to standardize images to a set format with predefined normalization steps so that each image is processed as consistently as possible. Images: Eaum M/Shutterstock (temperature gauge); gkuna/Shutterstock (broken tree)

Our Text Decoder: GPT-2

GPT-2 is OpenAI's precursor to GPT-3 and GPT-4 (probably obvious), but more importantly it is an open-source generative language model that is pre-trained on a relatively large corpus of text data. GPT-2 was pre-trained on approximately 40 GB of data, so it should also have prior knowledge of words that will help during training, again thanks to transfer learning.

The combination of these three models—DistilBERT for text processing, ViT for image processing, and GPT-2 for text decoding—will provide the basis for our multimodal system, as shown in **Figure 9.3**. These models all have prior knowledge, and we will rely on transfer learning capabilities to allow them to effectively process and generate highly accurate and relevant outputs for complex natural language and computer vision tasks.

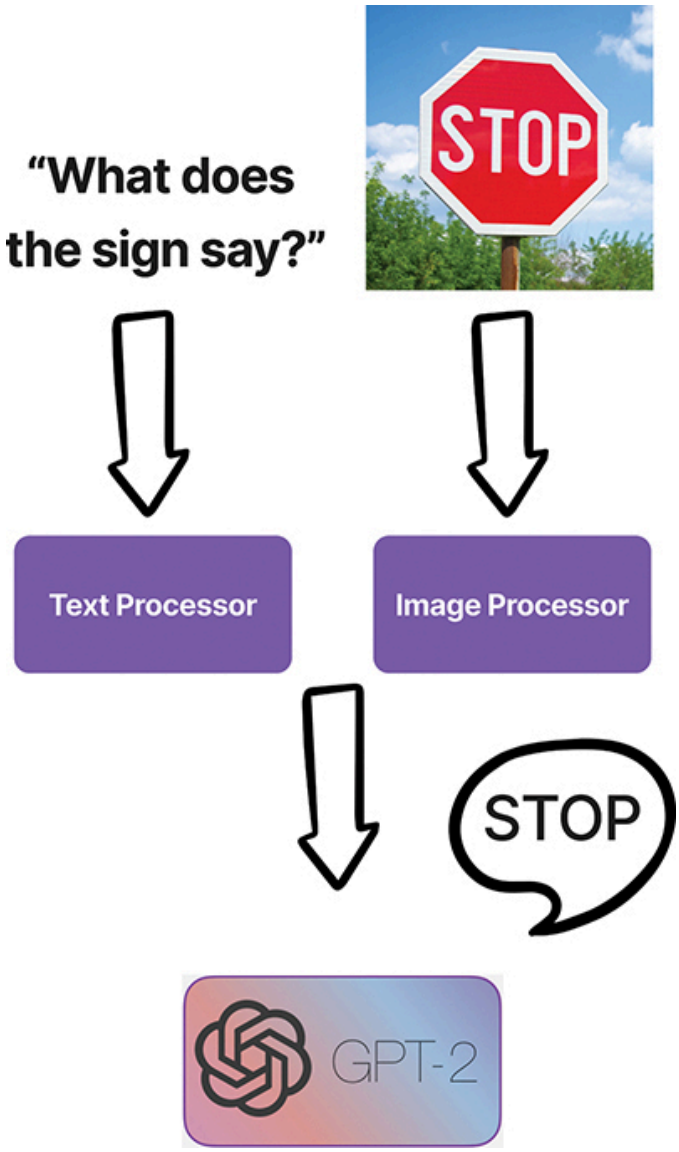


Figure 9.3 In a VQA system, the final single-token-prediction layer can be replaced with an entirely separate language model, such as the open-source GPT-2. The VQA system we will build has three Transformer-based models working side by side to solve a single, albeit very challenging, task. Image: panicattack/123RF (stop sign)

Hidden States Projection and Fusion

When we feed our text and image inputs into their respective models (DistilBERT and ViT), they produce output tensors that contain useful fea-

ture representations of the inputs. However, these features are not necessarily present in the same embedding space, and they may have different dimensionalities.

To address this mismatch, one option is to use linear projection layers (seen later in **Figure 9.5**) to transform the output tensors of the text and image models to have the same size as each other. This allows us to fuse the features extracted from the text and image inputs effectively. The shared dimensional space makes it possible to combine the text and image features (by averaging them, in our case) and feed them into the decoder (GPT-2) to generate a coherent and relevant textual response.

But how will GPT-2 accept these inputs from the encoding models? The answer to that question is a type of attention mechanism known as cross-attention.

Cross-Attention: What Is It, and Why Is It Critical?

Cross-attention is the mechanism that will allow our multimodal system to learn the interactions between our text and image inputs and the output text we want to generate. It is a critical component of the base Transformer architecture that allows it to incorporate information from inputs into outputs (the hallmark of a sequence-to-sequence model) effectively. The cross-attention calculation is actually much the same as the self-attention calculation, but occurs between two different sequences rather than within a single one. In cross-attention, the input sequence (or combined sequences in our case, because we will be inputting both text and images) will serve as the key and value input (which will be a combination of the queries from the image and text encoders), whereas the output sequence serves as the query input (our text-generating GPT-2).

Query, Key, and Value in Attention

The three internal components of attention—Query, Key, and Value—haven't really come up before in this book because we haven't really needed to understand why they exist. Instead, we simply relied on their ability to learn patterns in our data. Now, however, it's time to take a closer look at how these components interact so we can fully understand how cross-attention works.

In the self-attention mechanisms used by Transformers, the Query, Key, and Value components are crucial for determining the importance of each input token relative to others in the sequence. The Query represents the token for which we want to compute the attention weights, while the Keys and Values represent all tokens in the sequence. For example, in **Figure 9.4**, we are calculating attention for the token "like" (our Query) against every token in the sequence in the Key and Value space. The at-

tention scores are computed by taking the dot product between the Query and the Keys, scaling it by a normalization factor, and then multiplying the softmaxed resulting values by the Values to create a weighted sum of Value vectors.

In simpler terms, the Query is employed to extract pertinent information from other tokens, as determined by the attention scores. The Keys help identify which tokens are relevant to the Query, while the Values supply the corresponding information. This relationship is visualized in **Figure 9.4**.

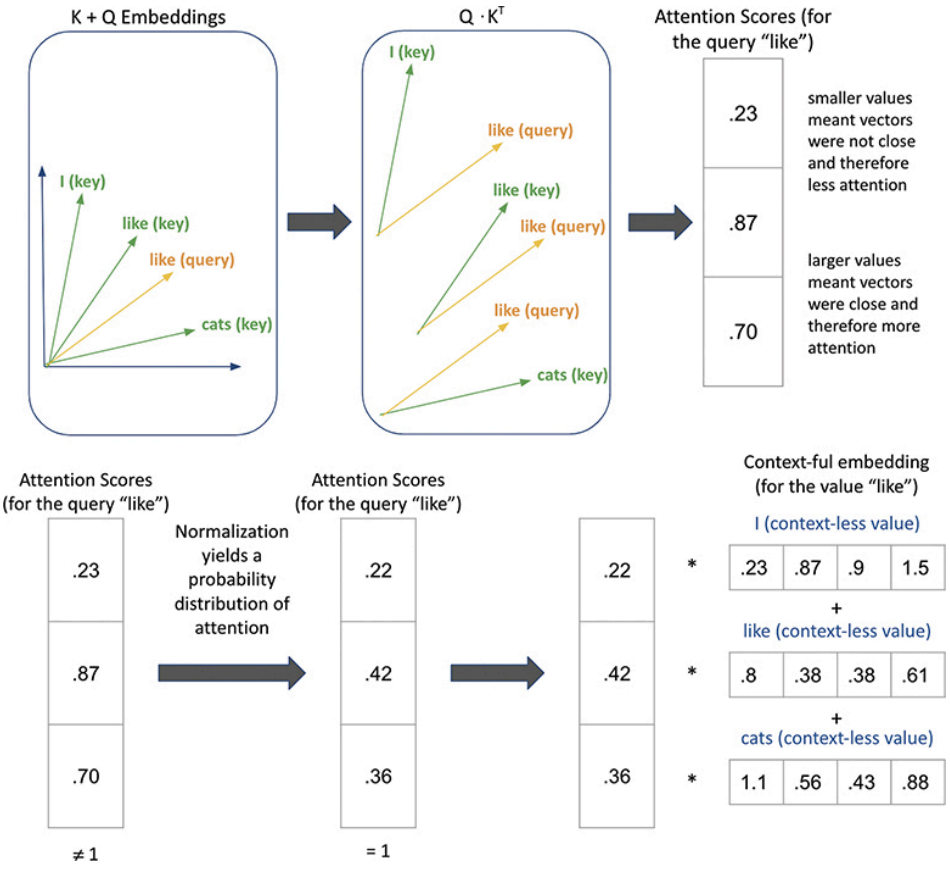


Figure 9.4 This figure represents the calculation of the scaled dot product attention value for the word "like" in the input "I like cats." Every input token to a Transformer-based LLM has an associated "query," "key," and "value" representation. The scaled dot product attention calculation generates attention scores for each Query token by taking the dot product with the Key tokens (top); those scores are then used to contextualize the Value tokens with proper weighting (bottom), yielding a final vector for each token in the input that is now aware of the other tokens in the input and how much it should be paying attention to them. In this case, the token "like" should be paying 22% of its attention to the token "I," 42% of its attention to itself (yes, tokens need to pay attention to themselves—as we all should—because they are part of the sequence and thus provide context), and 36% of its attention to the word "cats."

In cross-attention, the Query, Key, and Value matrices serve slightly different purposes. In this case, the Query represents the output of one modality (e.g., text), while the Keys and Values represent the outputs of another modality (e.g., image). Cross-attention is used to calculate attention scores that determine the degree of importance given to the output of one modality when processing the other modality.

In a multimodal system, cross-attention calculates attention weights that express the relevance between text and image inputs (illustrated in **Figure 9.5**). The Query is the output of the text model, while the Keys and Values are the output of the image model. The attention scores are computed by taking the dot product between the Query and the Keys and scaling it by a normalization factor. The resulting attention weights are then multiplied by the Values to create the weighted sum, which is utilized to generate a coherent and relevant textual response. **Listing 9.1** shows the hidden state sizes for our three models.

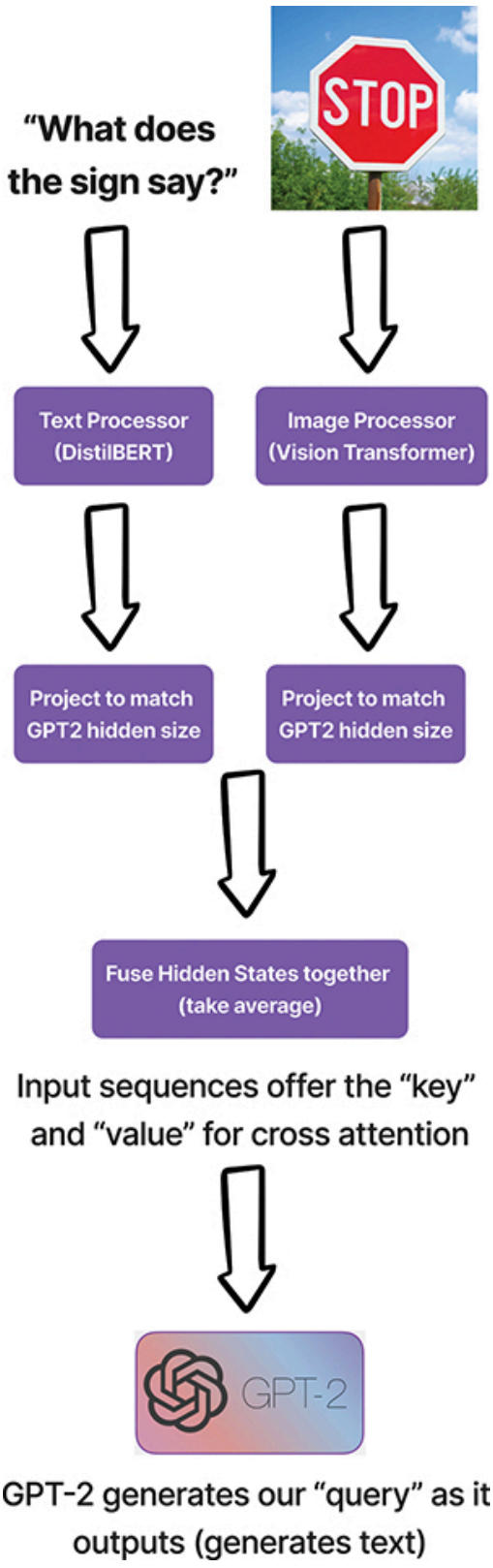


Figure 9.5 Our VQA system needs to fuse the encoded knowledge from the image and text encoders and pass that fusion to the GPT-2 model via the cross-attention mechanism. This mechanism takes the fused key and value vectors (see **Figure 9.4**) from the image and text encoders and passes them on to the decoder GPT-2, which uses the vectors to scale its own attention calculations. Image: panicattack/123RF (stop sign)

Listing 9.1 **Revealing LLMs' hidden states**

Click here to view code image

```
# Load the text encoder model and print the hidden size (number of hidden
units) in its configuration

print(AutoModel.from_pretrained(TEXT_ENCODER_MODEL).config.hidden_size)
# Load the image encoder model (using the Vision Transformer architecture) and
the hidden size in its configuration
print(ViTModel.from_pretrained(IMAGE_ENCODER_MODEL).config.hidden_size)

# Load the decoder model (for causal language modeling) and print the hidden si
its configuration
print(AutoModelForCausalLM.from_pretrained(DECODER_MODEL).config.hidden_size)

# 768
# 768
# 768
```

In our case, all models have the same hidden state size, so in theory we don't need to project anything. Nevertheless, it is good practice to include projection layers so that the model has a trainable layer that translates our text/image representations into something more meaningful for the decoder.

Initially, our cross-attention parameters will have to be randomized, and they will need to be learned during training. During the training process, the model learns to assign higher attention weights to relevant features while filtering out irrelevant ones. This way, the system can better understand the relationship between the text and image inputs, and generate more relevant and accurate textual responses. By assigning higher attention weights to relevant features while filtering out irrelevant ones, our system can better understand the relationship between the text and image inputs, generating more accurate and relevant textual responses.

With the ideas of cross-attention, fusion, and our models handy, let's move on to defining a multimodal architecture.

Our Custom Multimodal Model

Before getting deeper into the code, I'll point out that not all of the code that powers this example appears in these pages, but all of it lives in the notebooks on GitHub. I highly recommend following along using both!

When creating a novel PyTorch module (which is what we are doing), the main methods we need to define are the constructor (`init`), which will

instantiate our three Transformer models and potentially freeze layers to speed up training (more on that in **Chapter 10**), and the `forward` method, which will take in inputs and potentially labels to generate an output and a loss value. (Recall that loss is the same as error—the lower, the better.) The `forward` method will take the following inputs:

- **`input_ids`**: A tensor containing the input IDs for the text tokens. These IDs are generated by the tokenizer based on the input text. The shape of the tensor is `[batch_size, sequence_length]`.
- **`attention_mask`**: A tensor of the same shape as `input_ids` that indicates which input tokens should be attended to (value 1) and which should be ignored (value 0). It is mainly used to represent where padding tokens are located in the input sequence. For example, if we have a sequence of 10 tokens and we pad it to be 15 tokens long, the attention mask will have 5 0s, representing the pad tokens, and 10 1s, representing the 10 tokens we want to process.
- **`decoder_input_ids`**: A tensor containing the input IDs for the decoder tokens. These IDs are generated by the tokenizer based on the target text, which is used as a prompt for the decoder during training. The shape of the tensor during training is `[batch_size, target_sequence_length]`. At inference time, it will simply be a start token, so the model will have to generate the rest.
- **`image_features`**: A tensor containing the preprocessed image features for each sample in the batch. The shape of the tensor is `[batch_size, num_features, feature_dimension]`. These are generated by the pre-trained vision encoding model—in our case, the Vision Transformer.
- **`labels`**: A tensor containing the ground truth labels for the target text. The shape of the tensor is `[batch_size, target_sequence_length]`. These labels are used to compute the loss during training but won't exist at inference time. After all, if we had the labels, then we wouldn't need this model!

Listing 9.2 shows a snippet of the code it takes to create a custom model from our three separate Transformer-based models (BERT, ViT, and GPT2). The full class can be found in the book's repository for your copy-and-pasting needs.

Listing 9.2 **A snippet of our multimodal model**

Click here to view code image

```
class MultiModalModel(nn.Module):
 ...

 # Freeze the specified encoders or decoder
```

```
 def freeze(self, freeze):
 ...
 # Iterate through the specified components and freeze their parameters
 if freeze in ('encoders', 'all') or 'text_encoder' in freeze:
 ...
 for param in self.text_encoder.parameters():
 param.requires_grad = False

 if freeze in ('encoders', 'all') or 'image_encoder' in freeze:
 ...
 for param in self.image_encoder.parameters():
 param.requires_grad = False

 if freeze in ('decoder', 'all'):
 ...
 for name, param in self.decoder.named_parameters():
 if "crossattention" not in name:
 param.requires_grad = False

 # Encode the input text and project it into the decoder's hidden space
 def encode_text(self, input_text, attention_mask):
 # Check input for NaN or infinite values
 self.check_input(input_text, "input_text")

 # Encode the input text and obtain the mean of the last hidden state
 text_encoded = self.text_encoder(input_text, attention_mask=attention_mask)
.last_hidden_state.mean(dim=1)

 # Project the encoded text into the decoder's hidden space
 return self.text_projection(text_encoded)

 # Encode the input image and project it into the decoder's hidden space
 def encode_image(self, input_image):
 # Check input for NaN or infinite values
 self.check_input(input_image, "input_image")

 # Encode the input image and obtain the mean of the last hidden state
 image_encoded = self.image_encoder(input_image).last_hidden_state.mean(dim=1)

 # Project the encoded image into the decoder's hidden space
 return self.image_projection(image_encoded)

 # Forward pass: encode text and image, combine encoded features, and decode wi
 def forward(self, input_text, input_image, decoder_input_ids, attention_mask,
labels=None):
 # Check decoder input for NaN or infinite values
 self.check_input(decoder_input_ids, "decoder_input_ids")

 # Encode text and image
 text_projected = self.encode_text(input_text, attention_mask)
 image_projected = self.encode_image(input_image)

 # Combine encoded features
 combined_features = (text_projected + image_projected) / 2
```

```
# Set padding token labels to -100 for the decoder
if labels is not None:
labels = torch.where(labels == decoder_tokenizer.pad_token_id, -100, labels)

# Decode with GPT-2
decoder_outputs = self.decoder(
input_ids=decoder_input_ids,
labels=labels,
encoder_hidden_states=combined_features.unsqueeze(1)
)
return decoder_outputs
...
```

With a model defined and properly adjusted for cross-attention, let's look at the data that will power our engine.

Our Data: Visual QA

Our dataset, which comes from Visual QA (**Figure 9.6**), contains pairs of open-ended questions about images with human-annotated answers. The dataset is meant to produce questions that require an understanding of vision, language, and just a bit of commonsense knowledge to answer.

Figure 9.6 The **VisualQA.org** website has a dataset with open-ended questions about images. Source: Visual Question Answering (2024); **https://visualqa.org**.

Parsing the Dataset for Our Model

Listing 9.3 shows a function I wrote to parse the image files and creates a dataset that we can use with Hugging Face's Trainer object.

Listing 9.3 **Parsing the Visual QA files**

Click here to view code image

```
# Function to load VQA data from the given annotation and question files
def load_vqa_data(annotations_file, questions_file, images_folder, start_at=Non
```

```
end_at=None, max_images=None, max_questions=None):
 # Load the annotations and questions JSON files
 with open(annotations_file, "r") as f:
 annotations_data = json.load(f)
 with open(questions_file, "r") as f:
 questions_data = json.load(f)

 data = []
 images_used = defaultdict(int)
 # Create a dictionary to map question_id to the annotation data
 annotations_dict = {annotation["question_id"]: annotation for annotation in
annotations_data["annotations"]}

 # Iterate through questions in the specified range
 for question in tqdm(questions_data["questions"][start_at:end_at]):
 ...
 # Check if the image file exists and has not reached the max_questions limit
 ...

 # Add the data as a dictionary
 data.append(
 {
 "image_id": image_id,
 "question_id": question_id,
 "question": question["question"],
 "answer": decoder_tokenizer.bos_token + ' ' + annotation["multiple_choice_
answer"]+decoder_tokenizer.eos_token,
 "all_answers": all_answers,
 "image": image,
 }
 )
 ...
 # Break the loop if the max_images limit is reached
 ...

 return data

# Load training and validation VQA data
train_data = load_vqa_data(
 "v2_mscoco_train2014_annotations.json",
"v2_OpenEnded_mscoco_train2014_questions.json", "train2014",
)
val_data = load_vqa_data(
 "v2_mscoco_val2014_annotations.json", "v2_OpenEnded_mscoco_val2014_questions.j
"val2014"
)
from datasets import Dataset

train_dataset = Dataset.from_dict({key: [item[key] for item in train_data] for
train_data[0].keys()})

# Optionally save the dataset to disk for later retrieval
train_dataset.save_to_disk("vqa_train_dataset")
```

```
# Create Hugging Face datasets
val_dataset = Dataset.from_dict({key: [item[key] for item in val_data] for key
val_data[0].keys()})

# Optionally save the dataset to disk for later retrieval
val_dataset.save_to_disk("vqa_val_dataset")
```

The VQA Training Loop

Training in this case study won't be different from what we have done in earlier chapters. Most of the hard work was done in our data parsing, to be honest. We get to use Hugging Face's Trainer and TrainingArguments objects with our custom model, and training will simply come down to expecting a drop in our validation loss. The full code can be found in the book's repository, and a snippet is shown in **Listing 9.4**.

Listing 9.4 **Training loop for VQA**

Click here to view code image

```
# Define the model configurations
DECODER_MODEL = 'gpt2'
TEXT_ENCODER_MODEL = 'distilbert-base-uncased'
IMAGE_ENCODER_MODEL = "facebook/dino-vitb16" # A version of ViT from Facebook

# Initialize the MultiModalModel with the specified configurations
model = MultiModalModel(
 image_encoder_model=IMAGE_ENCODER_MODEL,
 text_encoder_model=TEXT_ENCODER_MODEL,
 decoder_model=DECODER_MODEL,
 freeze='nothing'
)

# Configure training arguments
training_args = TrainingArguments(
 output_dir=OUTPUT_DIR,
 optim='adamw_torch',
 num_train_epochs=1,
 per_device_train_batch_size=16,
 per_device_eval_batch_size=16,
 gradient_accumulation_steps=4,
 evaluation_strategy="epoch",
 logging_dir="./logs",
 logging_steps=10,
 fp16=device.type == 'cuda', # This saves memory on GPU-enabled machines
 save_strategy='epoch'
)

# Initialize the Trainer with the model, training arguments, and datasets
```

```
Trainer(
 model=model,
 args=training_args,
 train_dataset=train_dataset,
 eval_dataset=val_dataset,
 data_collator=data_collator
)
```

There's a lot of code that powers this example. As noted earlier, I highly recommend following along with the notebook on GitHub for the full code and comments!

Summary of Results

Figure 9.7 shows a sample of images with a few questions asked of our newly developed VQA system. Note that some of the responses are more than a single token, which is an immediate benefit of having the LLM as our decoder as opposed to outputting a single token as in standard VQA systems.

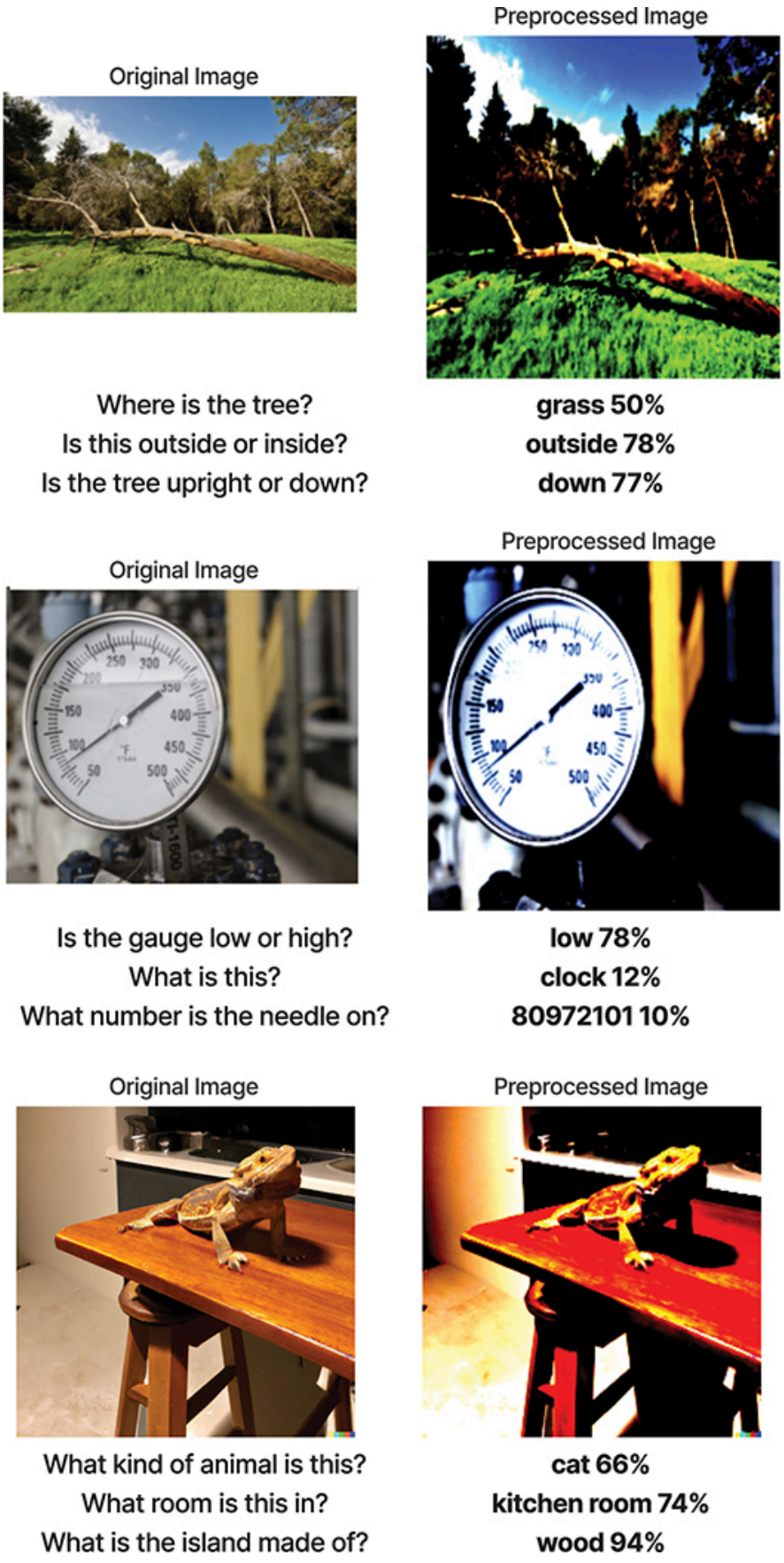
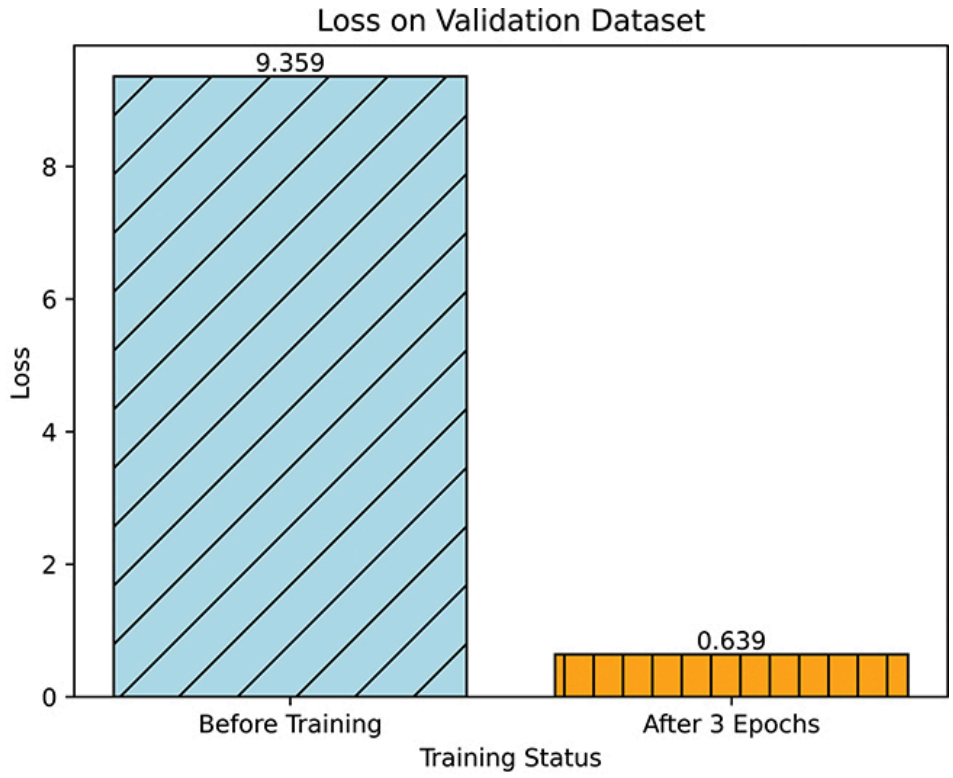


Figure 9.7 Our VQA system is not half bad at answering sample questions about images, even though we used relatively small models (in terms of number of parameters and especially compared to the state-of-the-art systems available today). Each percentage is the aggregated token prediction probabilities that GPT-2 generated while answering the given questions. Clearly, it is getting some questions wrong. With more training on more data, we can reduce the number of errors

even further. Images: Eaum M/Shutterstock (temperature gauge); gkuna/Shutterstock (broken tree)

This is only a sample of data and not a very holistic representation of performance. To showcase how our model training went, **Figure 9.8** shows the drastic change in our language modeling loss on the validation set and the accuracy on a hold-out testing dataset after only three epochs.



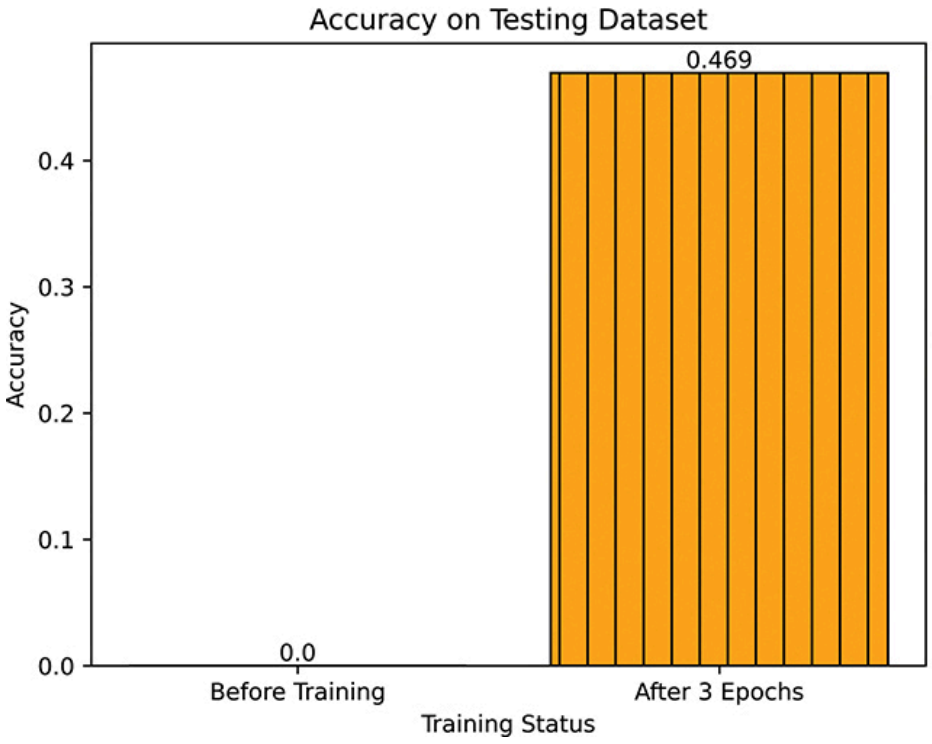


Figure 9.8 After only three epochs, our VQA system showed a massive drop in validation loss (top) and an increase in testing accuracy (bottom), which is great!

Our model is far from perfect. It will require more advanced training strategies and lots more training data before it can really be considered state of the art. Even so, using free data, free models, and (mostly) free compute power (my own laptop) yielded a not half-bad VQA system.

Let's step away from the idea of pure language modeling and image processing for just a moment. We'll next explore a novel way of fine-tuning language models using this approach's powerful cousin—reinforcement learning.

Case Study: Reinforcement Learning from Feedback

We have seen over and over the remarkable capabilities of language models in this book. Usually, we have dealt with relatively objective tasks such as classification. When the task was more subjective, such as semantic retrieval and anime recommendations, we had to take some time to define an objective quantitative metric to guide the model's fine-tuning and overall system performance. In general, defining what constitutes "good" output text can be challenging, as it is often subjective and task/context-dependent. Different applications may require different "good" attributes, such as creativity for storytelling, readability for summarization, or code functionality for code snippets.

When we fine-tune LLMs, we must design a loss function to guide training. The loss function calculates the error or difference between the predicted output and the actual target output for a particular batch of data. But designing a loss function that captures these more subjective attributes can seem intractable, and most language models continue to be trained using a simple next-token prediction loss (autoregressive language modeling), such as cross-entropy. As for as evaluating the outputs of LLMs (which we will dive into in **Chapter 12**), metrics exist that are designed to compare generated text to ground truth reference texts using very simple rules and heuristics like matching keywords and phrases. We could use an embedding similarity to compare outputs to ground truth sequences, but this approach considers only semantic information, which isn't always the only thing we need to compare. We might want to consider the style of the text, for example.

But what if we could use live feedback (human or automated) for evaluating generated text as a performance measure or even as a loss function to optimize the model? That's where **reinforcement learning from feed-**

back (RLF)—RLHF for human feedback and RLAIF for AI feedback—comes into play. By employing reinforcement learning methods, RLF can directly optimize a language model using real-time feedback, allowing models trained on a general corpus of text data to align more closely with nuanced human values.

ChatGPT is one of the first notable applications of RLHF. While OpenAI provides an impressive explanation of RLHF in its paper "Training Language Models to Follow Instructions with Human Feedback,"[1] it doesn't cover everything, so I'll try to fill in the gaps.

---

1 https://arxiv.org/abs/2203.02155

The training process basically breaks down into three core steps (shown in Figure 9.9):

1. **Pre-training a language model:** Pre-training a language model involves training the model on a large corpus of text data, such as articles, books, and websites, or even a curated dataset. With reinforcement learning, the language model in question is almost always a generative one. This means we are expecting to work with models like Llama, Mistral, or T5. During this phase, the model learns to generate text for general corpora or in service of a task. This process helps the model to learn grammar, syntax, and some level of semantics from the text data. The objective function used during pre-training is typically the cross-entropy loss, which measures the difference between the predicted token probabilities and the true token probabilities. Pre-training allows the model to acquire a foundational understanding of the language, which can later be fine-tuned for specific tasks.
2. **Defining (potentially training) a reward model:** After pre-training the language model, the next step is to define a reward model that can be used to evaluate the quality of the generated text. This involves gathering human feedback, such as rankings or scores for different text samples, which can be used to create a dataset of human preferences. The reward model aims to capture these preferences, and can be trained as a supervised learning problem, where the goal is to learn a function that maps generated text to a reward signal (a scalar value) representing the quality of the text according to human feedback. The reward model serves as a proxy for human evaluation and is used during the reinforcement learning phase to guide the fine-tuning process.
3. **Fine-tuning the LM with reinforcement learning:** With a pre-trained language model and a reward model in place, the final step is to fine-tune the language model using reinforcement learning techniques. In this phase, the model generates text, receives feedback from the re-

ward model, and updates its parameters based on the reward signal. The objective is to optimize the language model such that the generated text aligns closely with human preferences. Fine-tuning with reinforcement learning allows the model to adapt to specific tasks and generate text that better reflects human values and preferences.

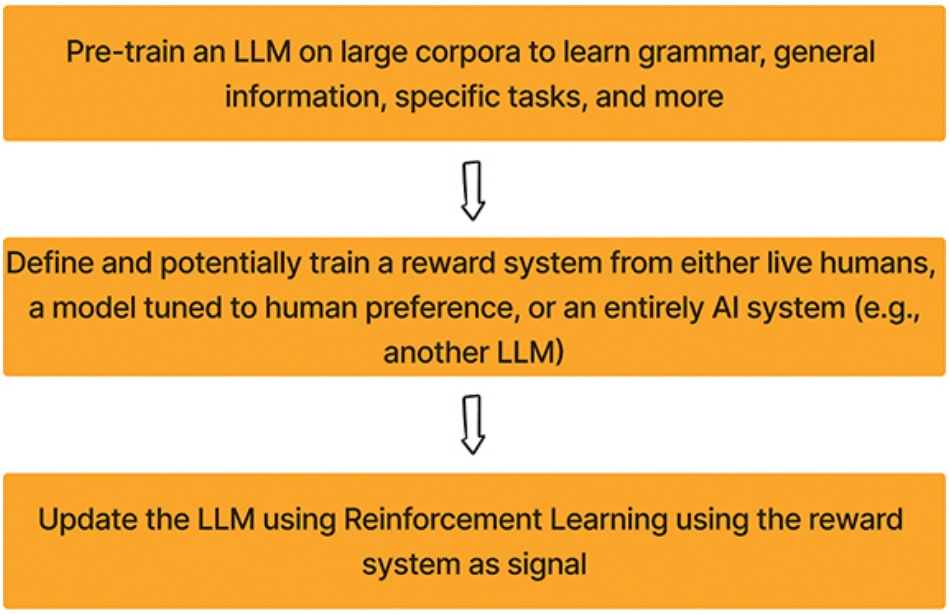


Figure 9.9 The core steps of reinforcement learning-based LLM training include pre-training the LLM, defining and potentially training a reward model, and using that reward model to update the LLM from step 1.

We will perform this process in its entirety in **Chapter 10**. For now, to set up this relatively complicated process, I'll outline a simpler version. In this version, we will take a pre-trained LLM off the shelf (FLAN-T5), use an already defined and trained reward model, and really focus on step 3, the reinforcement learning loop.

Our Model: FLAN-T5

We have seen and used FLAN-T5 (visualized in an image taken from the original FLAN-T5 paper in **Figure 9.10**) before, so this discussion is really just a refresher. FLAN-T5 is an encoder–decoder model (effectively a pure Transformer model), which means it has built-in trained cross-attention layers and offers the benefit of instruction fine-tuning (as GPT-3.5, ChatGPT, and GPT-4 do). We'll use the open-source "small" version of the model.

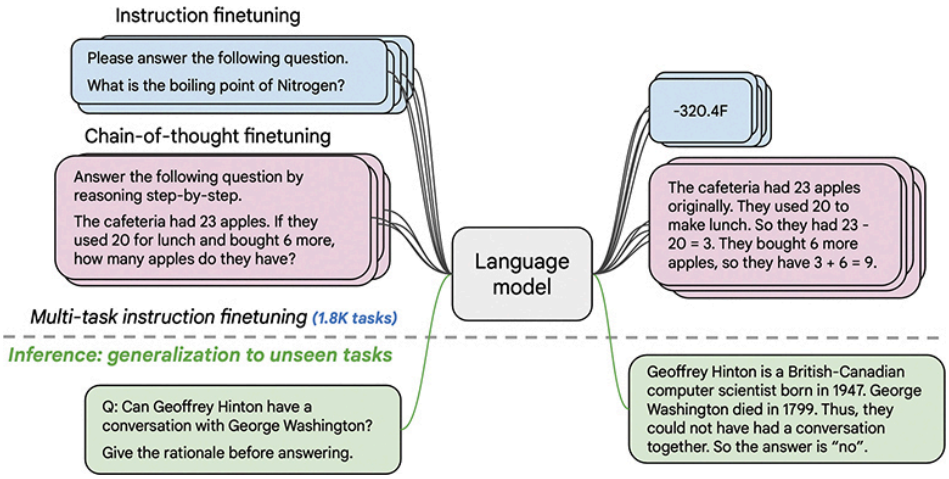


Figure 9.10 FLAN-T5 is an open-source encoder–decoder architecture that has been instruction fine-tuned.

In **Chapter 10**, we will perform our own version of instruction fine-tuning. For now, we will borrow this already instruction-fine-tuned LLM from the good people at Google AI and move on to define a reward model.

Our Reward Model: Sentiment and Grammar Correctness

A reward model has to take in the output of an LLM (in our case, a sequence of text) and return a scalar (single number) reward, which should numerically represent feedback on the output. This feedback can come from an actual human, which would be very slow to run. Alternatively, it could come from another language model or even a more complicated system that ranks potential model outputs, with those rankings then being converted to rewards. As long as we are assigning a scalar reward for each output, either approach will yield a viable reward system.

In **Chapter 10**, we will be doing some really interesting work to define our own reward model. Here, though, we will again rely on the hard work of others and use the following prebuilt LLMs:

- **Sentiment from the `cardiffnlp/twitter-roberta-base-sentiment` LLM:** The idea is to promote summaries that are neutral in nature, so the reward from this model will be defined as the logit value of the **"neutral"** class. As we've seen in previous chapters, logit values are the raw, un-normalized scores that a model assigns to each class before applying a softmax function to obtain probabilities. The higher

the logit score for a class, the more confident the model is that the class is the correct one.

- **A "grammar score" from the `textattack/roberta-base-CoLA` LLM:** We want our summaries to be grammatically correct, so using a score from this model should promote summaries that are easier to read. The reward will be defined as the logit value of the **"grammatically correct"** class.

Note that by choosing these classifiers to form the basis of our reward system, we are implicitly trusting in their performance. I checked out their descriptions on the Hugging Face model repository to see how they were trained and which performance metrics I could find. In general, the reward systems play a big role in this process—so if they are not aligned with how you truly would reward text sequences, you are in for some trouble.

A snippet of the code that translates generated text into scores (rewards) using a weighted sum of logits from our two models can be found in **Listing 9.5**.

Listing 9.5 **Defining our reward system**

Click here to view code image

```
from transformers import pipeline

# Initialize the CoLA pipeline
tokenizer = AutoTokenizer.from_pretrained("textattack/roberta-base-CoLA")
model = AutoModelForSequenceClassification.from_pretrained("textattack/roberta-
CoLA")
cola_pipeline = pipeline('text-classification', model=model, tokenizer=tokenize

# Initialize the sentiment analysis pipeline
sentiment_pipeline = pipeline('text-classification', 'cardiffnlp/twitter-robert
sentiment')

# Function to get CoLA scores for a list of texts
def get_cola_scores(texts):
 scores = []
 results = cola_pipeline(texts, function_to_apply='none', top_k=None)
 for result in results:
 for label in result:
 if label['label'] == 'LABEL_1': # Good grammar
 scores.append(label['score'])
 return scores

# Function to get sentiment scores for a list of texts
def get_sentiment_scores(texts):
 scores = []
```

```
  results = sentiment_pipeline(texts, function_to_apply='none', top_k=None)
  for result in results:
  for label in result:
  if label['label'] == 'LABEL_1': # Neutral sentiment
  scores.append(label['score'])
  return scores

texts = [
  'The Eiffel Tower in Paris is the tallest structure in the world, with a heigh
1,063 metres',
  'This is a bad book',
  'this is a bad books'
]

# Get CoLA and neutral sentiment scores for the list of texts
cola_scores = get_cola_scores(texts)
neutral_scores = get_sentiment_scores(texts)

# Combine the scores using zip
transposed_lists = zip(cola_scores, neutral_scores)

# Calculate the weighted averages for each index
rewards = [1 * values[0] + 0.5 * values[1] for values in transposed_lists]

# Convert the rewards to a list of tensors
rewards = [torch.tensor([_]) for _ in rewards]

## Rewards are [2.52644997, -0.453404724, -1.610627412]
```

With a model and a reward system ready to go, we just need to introduce one more new component, our reinforcement learning library: TRL.

Transformer Reinforcement Learning

Transformer Reinforcement Learning (TRL) is an open-source library we can use to train Transformer models with reinforcement learning. This library is integrated with our favorite package: Hugging Face's `transformers`.

The TRL library supports pure decoder models like Llama-3 and Mistral (more on that in **Chapter 10**) as well as sequence-to-sequence models like FLAN-T5. These models can be optimized using **proximal policy optimization (PPO)**. The inner workings of PPO aren't covered in this book, but the long and short of it is that PPO helps the model learn effectively by balancing two important processes: exploration and exploitation. Exploration means trying out new actions to discover potentially better strategies, whereas exploitation means using known actions that give good results. PPO ensures that the model doesn't change too drastically in any single update, making the learning process smoother and more sta-

ble. TRL also has many examples on its GitHub page if you want to see even more applications.

Figure 9.11 shows the high-level process of our (for now) simplified RLF loop.

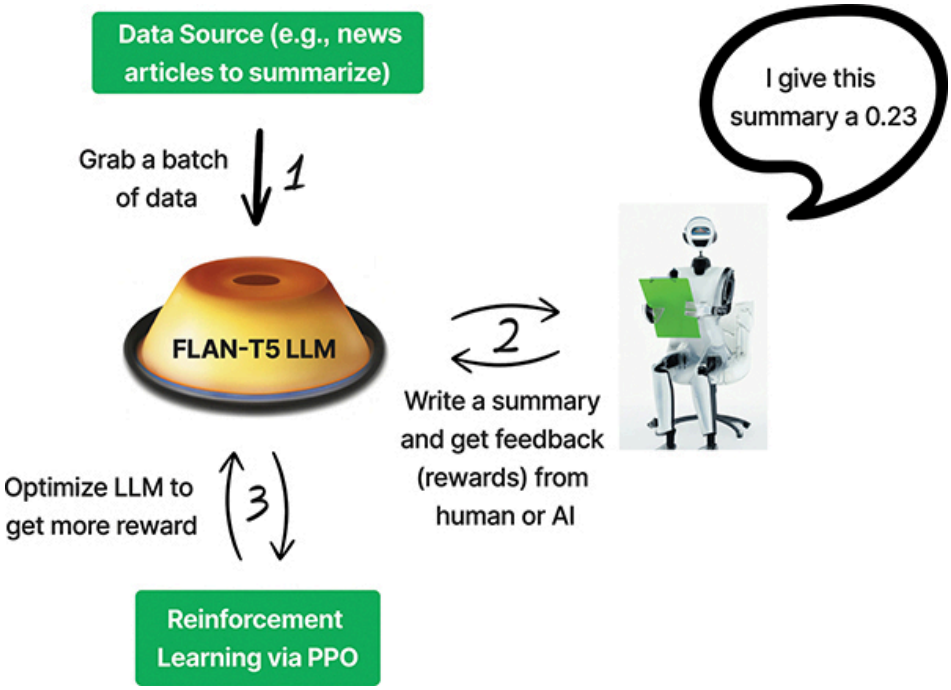


Figure 9.11 Our first reinforcement learning from feedback loop has our pre-trained LLM (FLAN-T5) learning from a curated dataset and a prebuilt reward system. In **Chapter 10**, we will see this loop performed with much more customization and rigor.

Let's jump into defining our training loop with some code to really see some results here.

The RLF Training Loop

Our RLF fine-tuning loop has a few steps:

1. Instantiate *two* versions of our model:
    1. Our "reference" model, which is the original FLAN-T5 model and will *never* be updated
    2. Our "current" model, which will have its parameters updated after every batch of data
2. Grab a batch of data from a source (in our case, a corpus of news articles from Hugging Face).
3. Calculate the rewards from our two reward models and aggregate them into a single scalar (number) as a weighted sum of the two

rewards.

4. Pass the rewards to the TRL package, which calculates two things:
    1. How to update the model slightly based on the reward system.
    2. How divergent the text is from text generated from the reference model—that is, the **KL-divergence** between our two outputs. We won't go deep into this calculation, but simply note that the KL-divergence, or Kullback–Leibler divergence, is a way to measure how one set of probabilities differs from another. Imagine you have two ways to guess something, such as two different sets of predictions. KL-divergence helps us understand how much the second set of guesses differs from the first set. In this context, it helps ensure that the new text generated by the model stays relatively similar to the original model's (the model pre-RL) text, maintaining consistency and preventing unexpected or off-track outputs.
5. TRL updates the "current" model from the batch of data, logs anything to a reporting system (I like the free Weights & Biases platform), and starts over from step 1.

This training loop is illustrated in **Figure 9.12**.

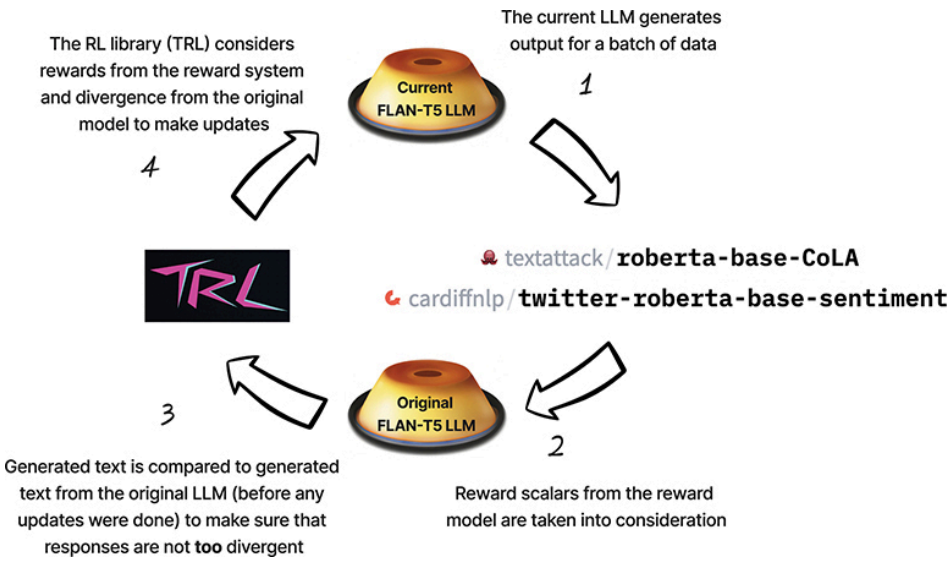


Figure 9.12 Our RLF training loop has four main steps: (1) The LLM generates an output; (2) the reward system assigns a scalar reward (positive for good, negative for bad); (3) the TRL library factors in rewards and divergence before doing any updating; and (4) the PPO policy updates the LLM.

A snippet of code for this training loop appears in **Listing 9.6**; the entire loop is defined in this book's code repository.

Listing 9.6 **Defining our RLF training loop with TRL**

Click here to view code image

```
from datasets import load_dataset
from tqdm.auto import tqdm

# Set the configuration
config = PPOConfig(
 model_name="google/flan-t5-small",
 batch_size=4,
 learning_rate=2e-5,
 remove_unused_columns=False,
 log_with="wandb",
 gradient_accumulation_steps=8,
)

# Set random seed for reproducibility
np.random.seed(42)

# Load the model and tokenizer
flan_t5_model = AutoModelForSeq2SeqLMWithValueHead.from_pretrained(config.model
flan_t5_model_ref = create_reference_model(flan_t5_model)
flan_t5_tokenizer = AutoTokenizer.from_pretrained(config.model_name)

# Load the dataset
dataset = load_dataset("argilla/news-summary")

# Preprocess the dataset
dataset = dataset.map(
 lambda x: {"input_ids": flan_t5_tokenizer.encode('summarize: ' + x["text"],
return_tensors="pt")},
 batched=False,
)

# Define a collator function
def collator(data):
 return dict((key, [d[key] for d in data]) for key in data[0])

# Start the training loop
for epoch in tqdm(range(2)):
 for batch in tqdm(ppo_trainer.dataloader):
 game_data = dict()
 # Prepend the "summarize: " instruction that T5 works well with
 game_data["query"] = ['summarize: ' + b for b in batch["text"]]
 # Get response from Flan-T5
 input_tensors = [_.squeeze() for _ in batch["input_ids"]]
 response_tensors = []
 for query in input_tensors:
 response = ppo_trainer.generate(query.squeeze(), **generation_kwargs)
 response_tensors.append(response.squeeze())

 # Store the generated response
 game_data["response"] = [flan_t5_tokenizer.decode(r.squeeze(),
```

```
 skip_special_tokens=False) for r in response_tensors]

 # Calculate rewards from the cleaned response (no special tokens)
 game_data["clean_response"] = [flan_t5_tokenizer.decode(r.squeeze(),
skip_special_tokens=True) for r in response_tensors]
 game_data['cola_scores'] = get_cola_scores(game_data["clean_response"])
 game_data['neutral_scores'] = get_sentiment_scores(game_data["clean_response"]
 rewards = game_data['neutral_scores']
 transposed_lists = zip(game_data['cola_scores'], game_data['neutral_scores'])
 # Calculate the averages for each index
 rewards = [1 * values[0] + 0.5 * values[1] for values in transposed_lists]
 rewards = [torch.tensor([_]) for _ in rewards]

 # Run PPO training
 stats = ppo_trainer.step(input_tensors, response_tensors, rewards)

 # Log the statistics (I use Weights & Biases)
 stats['env/reward'] = np.mean([r.cpu().numpy() for r in rewards])
 ppo_trainer.log_stats(stats, game_data, rewards)

# After the training loop, save the trained model and tokenizer
flan_t5_model.save_pretrained("t5-align")
flan_t5_tokenizer.save_pretrained("t5-align")
```

Let's see how it does after two epochs!

Summary of Results

Figure 9.13 shows how rewards were given over the training loop of two epochs. As the system progressed, it gave out more rewards, which is generally a good sign. Note that the rewards started out relatively high, indicating FLAN-T5 was already providing relatively neutral and readable responses, so we should not expect drastic changes in the summaries.

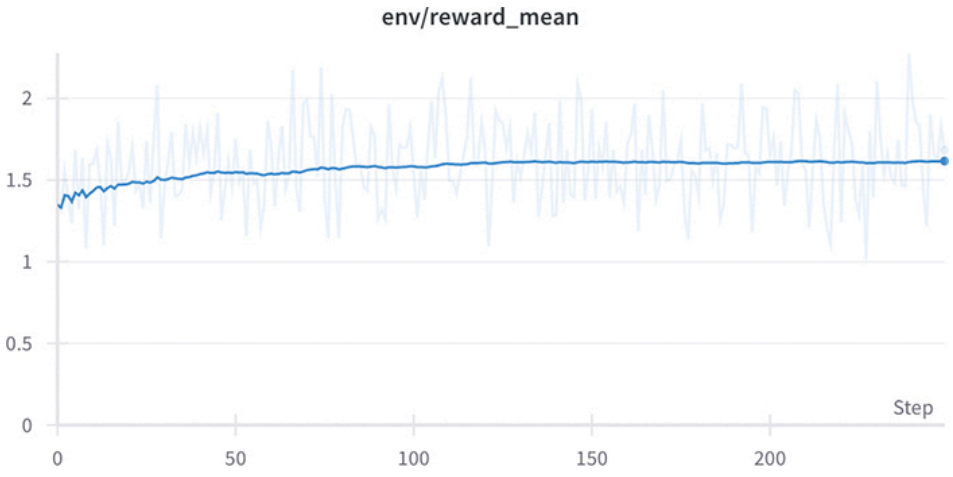


Figure 9.13 Our system is giving out more rewards as training progresses (the graph is smoothed to see the overall movement).

But what do these adjusted generations look like? **Figure 9.14** shows a sample of generated summaries before and after our RLF fine-tuning.

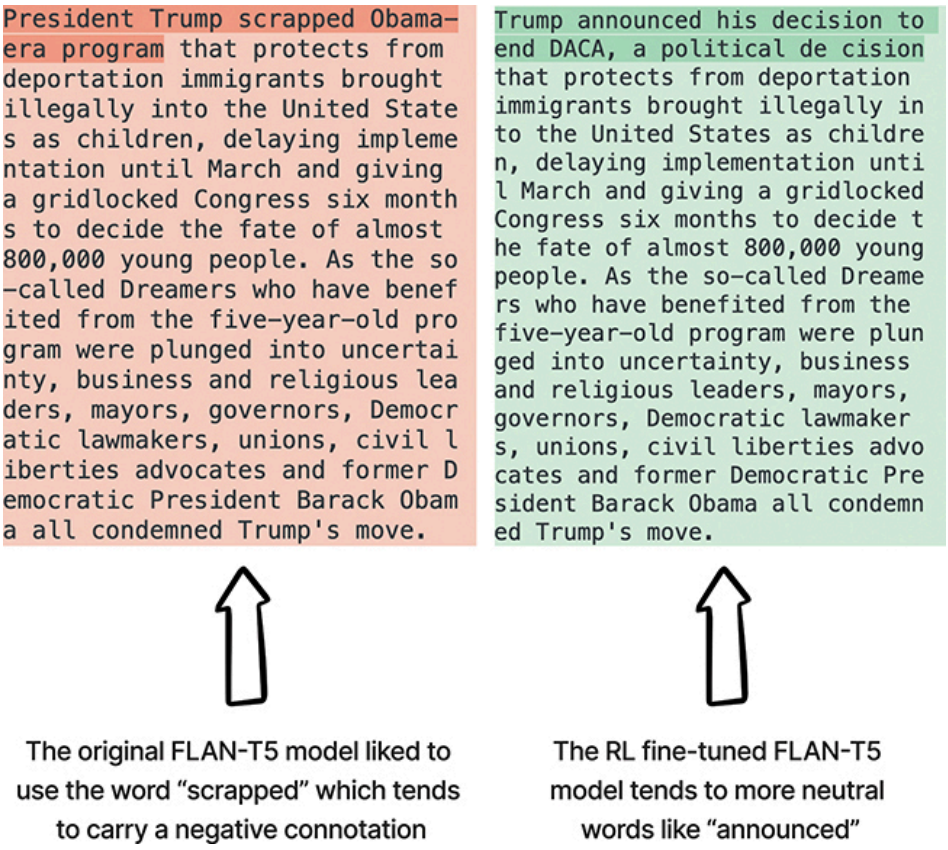


Figure 9.14 Our fine-tuned model barely differs in most summaries but does tend to use more neutral-sounding words that are grammatically correct and easy to read.

Figure 9.15 shows a broader out-of-sample dataset's awarded values for our reward classifiers. We can see an increase in both the mean and the median of rewards coming in from both classifiers. Great!

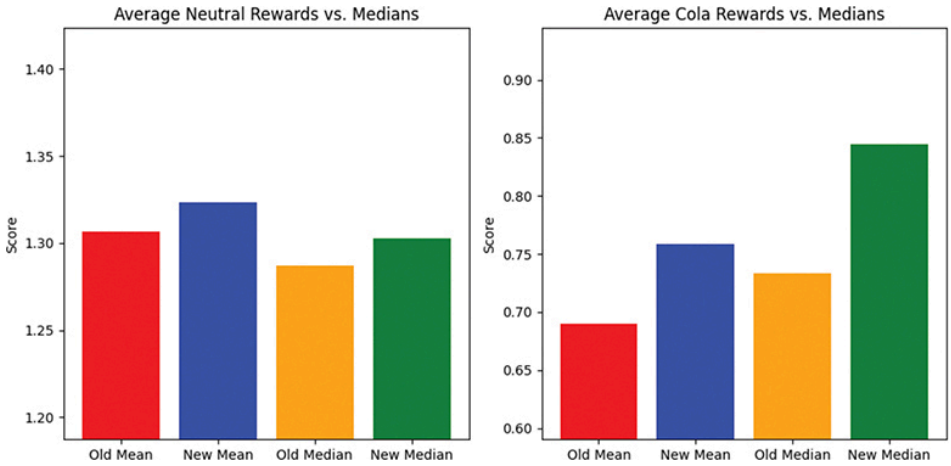


Figure 9.15 Our FLAN-T5 model is getting higher average and median rewards from both our Cola model (the grammatical correctness detector) and the neutral detection model on an out-of-sample dataset of 200 items. This is a great sign that our model will perform well on unseen data.

This is our first example of a nonsupervised data fine-tuning of an LLM. We never gave FLAN-T5 (article, summary) example pairs to help it learn *how* to summarize articles—and that's important. FLAN-T5 has already seen supervised datasets on summarization, so it should already know how to do that. All we wanted to do was to nudge the responses to be more aligned with a reward metric that we defined. **Chapter 10** provides a much more in-depth example of this process, in which we train an LLM with supervised data, train our own reward system, and perform this same TRL loop with much more interesting results.

Summary

Foundational models like FLAN-T5, ViT, ChatGPT, GPT-4, Meta's Llama family of models, GPT-2, and BERT can be wonderful starting points for solving a wide variety of tasks. Fine-tuning them with supervised labeled data to tweak classifications and embeddings can get us even further, but some tasks require us to get creative with our fine-tuning processes, with our data, and with our model architectures. This chapter merely scratches the surface of what is possible. The next few chapters will dive even deeper into ways to modify models and use data more creatively, and will even start to answer the question of how we can share our models with the world.