11

Moving LLMs into Production

Introduction

As the power we unlock from large language models grows, so, too, does the necessity of deploying these models to production so we can share our hard work with more people. This chapter explores different strategies for considering deployments of both closed-source and open-source LLMs, with an emphasis on best practices for model management, preparation for inference, and methods for improving efficiency such as quantization, pruning, and distillation.

Deploying Closed-Source LLMs to Production

For closed-source LLMs, the deployment process typically involves interacting with an API provided by the company that developed the model. This model-as-a-service approach is convenient because the underlying hardware and model management are abstracted away. However, it also necessitates careful API key management.

Cost Projections

In previous chapters, we discussed costs to some extent. To recap, in the case of closed-source models, the cost projection primarily involves calculating the expected API usage, as this is typically how such models are accessed. The cost here will depend on the provider's pricing model and can vary based on several factors, including the following:

- **API calls:** This is the number of requests your application makes to the model. Providers usually base their charges on the number of API calls.
- **Using different models:** The same company may offer different models for different prices. Our fine-tuned Ada model is slightly more expensive than the standard Ada model, for example.
- **Model/prompt versioning:** If the provider offers different versions of the model or your prompts, there might be varying charges for each.

Estimating these costs requires a clear understanding of your application's needs and expected usage. For example, an application that

makes continuous, high-volume API calls will cost significantly more than one making infrequent, low-volume calls.

API Key Management

If you are using a closed-source LLM, chances are you will have to manage some API keys to use the API. There are several best practices for managing API keys. First, they should never be embedded in code, as this practice readily exposes them to version control systems or inadvertent sharing. Instead, use environment variables or secure cloud-based key management services to store your keys.

You should also regularly rotate your API keys to minimize the impact of any potential key leakage. If a key is compromised but is valid for only a short time, the window for misuse is limited.

Lastly, use keys with the minimum permissions necessary. If an API key is only needed to make inference requests to a model, it should not have permissions to modify the model or access other cloud resources.

Deploying Open-Source LLMs to Production

Deploying open-source LLMs is a different process, primarily because you have more control over the model and its deployment. However, this control also comes with additional responsibilities related to preparing the model for inference and ensuring it runs efficiently.

Preparing a Model for Inference

While we can use a model fresh from training in production, we can do a bit more to optimize our machine learning code for production inference. This usually involves converting the model to inference mode by calling the `.eval()` method in frameworks like PyTorch. Such a conversion disables some of the lower-level deep learning layers, such as the dropout and batch normalization layers, which behave differently during training and inference, making our model deterministic during inference. **Listing 11.1** shows how we can perform the `.eval()` call on our anime genre predictor from **Chapter 10** with a simple code addition.

Listing 11.1 **Setting an LLM to eval mode**

Click here to view code image

```
trained_model = AutoModelForSequenceClassification.from_pretrained(
 f"genre-prediction",
problem_type="multi_label_classification",
```

```
).eval() # Stops dropout layers from cutting off connections and makes the outp
  nondeterministic
```

Layers like dropout layers—which help prevent overfitting during training by randomly setting some activations to zero—should not be active during inference. Disabling them with `.eval()` ensures the model's output is more deterministic (i.e., stable and repeatable), providing consistent predictions for the same input while also speeding up inference and enhancing both the transparency and interpretability of the model.

Interoperability

It's beneficial to have your models be interoperable, meaning they can be used across different machine learning frameworks. One popular way to achieve this is by using ONNX (Open Neural Network Exchange), an open standard format for machine learning models.

ONNX

ONNX allows you to export models from one framework (e.g., PyTorch) and import them into another framework (e.g., TensorFlow) for inference. This cross-framework compatibility is very useful for deploying models in different environments and platforms. **Listing 11.2** shows a code snippet of using Hugging Face's `optimum` package—a utility package for building and running inference with an accelerated runtime such as ONNX Runtime—to load a sequence classification model into an ONNX format.

Listing 11.2 **Converting our genre prediction model to ONNX**

Click here to view code image

```
#!pip install optimum
from optimum.onnxruntime import ORTModelForSequenceClassification

ort_model = ORTModelForSequenceClassification.from_pretrained(
 f"genre-prediction-bert",
 from_transformers=True
)
```

Suppose you train a model in PyTorch but want to deploy it on a platform that primarily supports TensorFlow. In this case, you could first convert your model to ONNX format and then convert it to TensorFlow, thereby avoiding the need to retrain the model.

Quantization

We talked briefly about quantization in the last chapter when we were training SAWYER. We quantized the Llama-3 model to lower the precision of its weights to make training faster and take less memory. Let's take a closer look at how much quantization can influence our models.

As a refresher, quantization refers to the technique of representing models using fewer bits by reducing the precision of its parameters. This process involves converting continuous or high-precision values into a smaller set of discrete values, typically by mapping floating-point numbers to integers. The primary goal of quantizing LLMs is to decrease memory usage and accelerate inference.

There are several methods to quantize a model, but I wanted to focus on a specific use-case I'm asked about a lot as an AI consultant and teacher: deploying an off-the-shelf model using quantization with no fine-tuning. These models could be ones that were pre-trained by other organizations, such as Llama-3-8B, or ones that were previously fine-tuned on specific datasets without quantization.

The code it takes to quantize a model is relatively straightforward using popular packages like Transformers, which include implementations of algorithms like NF4 (**Listing 11.3**). NF4, which stands for NormalFloat 4, is a particularly effective strategy for maintaining the performance of AI models. Originally introduced in the LoRA paper, NF4 has become a preferred choice in modern quantization strategies.

Listing 11.3 **Load Llama-3-8B-Instruct with and without quantization**

Click here to view code image

```
# Import necessary classes and functions from the transformers library
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfi

# Define the model name to load from Hugging Face's model hub
model_name = 'meta-llama/Meta-Llama-3-8B-Instruct'

# Configure the quantization settings using BitsAndBytesConfig
# Setting load_in_4bit to True enables 4-bit quantization
# bnb_4bit_use_double_quant enables double quantization for more precise contro
# bnb_4bit_quant_type specifies the NF4 quantization algorithm
# bnb_4bit_compute_dtype sets the data type for computation to bfloat16 for eff
bits_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
```

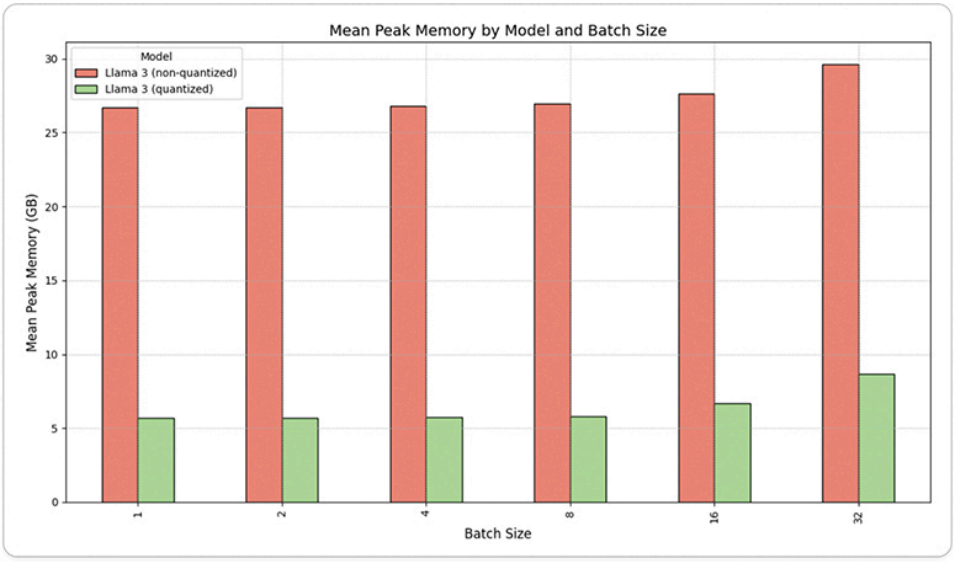
```
)

# Initialize the tokenizer for the model
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Load and configure the quantized model
qt_model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bits_config,
    device_map="auto"
).eval()  # Set the model to evaluation mode which disables training specific
operations like dropout

# Load the non-quantized version of the same model
non_qt_model = AutoModelForCausalLM.from_pretrained(
    model_name,
    device_map="auto"
).eval()  # Set the model to evaluation mode
```

Let's test both the quantized and the non-quantized models side by side on three considerations:

- **Optimizing inference:** Memory and latency reduction
- **Raw token output differences:** Measuring the raw differences between the next token prediction outputs
- **Performance on benchmarks/test sets:** Running generative benchmarks and comparing the two models

Optimizing Inference with Quantization

Probably the most well-known benefits of quantization are the inference gains both in memory usage and in latency/throughput. Lower parameter precision means smaller memory requirements for the model and faster computations. The memory usage and latency differences are dramatic between the two models and carry through for both small and larger batch sizes, as seen in **Figure 11.1**.



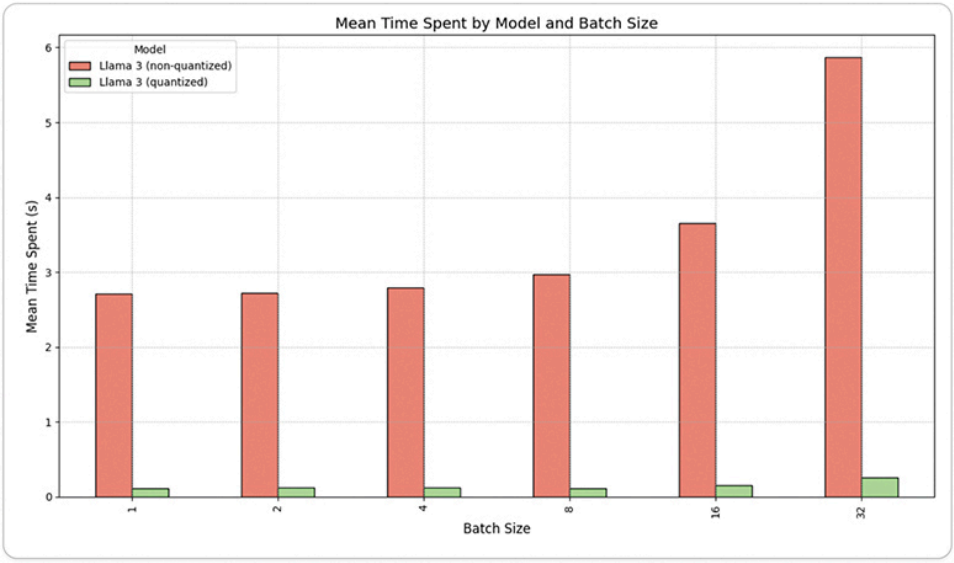


Figure 11.1 Measuring the peak memory usage and latency of the forward pass of Llama 3-8B shows striking differences. The non-quantized model (red) uses far more memory (top) and takes far longer to process inputs in batch sizes between 1 and 32 (bottom).

Quantized models are supposed to be faster and more memory efficient, so this is just the tip of the iceberg. But are they as reliable as their non-quantized cousin? Are they better? Worse? Let's see how we can find out.

Model Output Differences with Quantization

The rawest measure of language modeling output is to directly measure the differences in the next token prediction process. In **Figure 11.2**, I ask both versions of the Llama-3 model 163 questions from a subset of

MMLU-Virology (the benchmark content isn't as relevant here). I use the Jaccard index (similarity)—a similarity metric between two sets, measured as the number of items they have in common divided by the total number of unique items between them—to quantify the differences between the raw next-token predictions for each input at various token cutoff points: k = 1, 2, 3, etc. For example, if k = 3, I compare the top three token options for the first token prediction for each model and take the Jaccard similarity between them. I then average these values across the 163 examples.

This procedure is a relatively straightforward way to quantify the differences in the raw model output of quantized versus non-quantized models. I also chose the Jaccard index for its robustness in scenarios where the exact alignment of token sets is less important than the overall overlap—it is ideal for evaluating models where slight deviations in token predictions are acceptable. We can see that most tokens are shared in common, but a non-insignificant number of tokens are, in fact, different.

Given **Figure 11.2**, roughly speaking, we can expect about 75% to 83% of the tokens to match in the top 1, 3, 5, 10, and 20 predicted tokens for this test set, which can lead to performance differences (see the next section). These raw token outputs will not only affect performance on test sets, but also yield differences in the inference parameters that we set. For example, setting a top p (which affects token probabilities) for a non-quantized model might yield drastically different results on the quantized version.

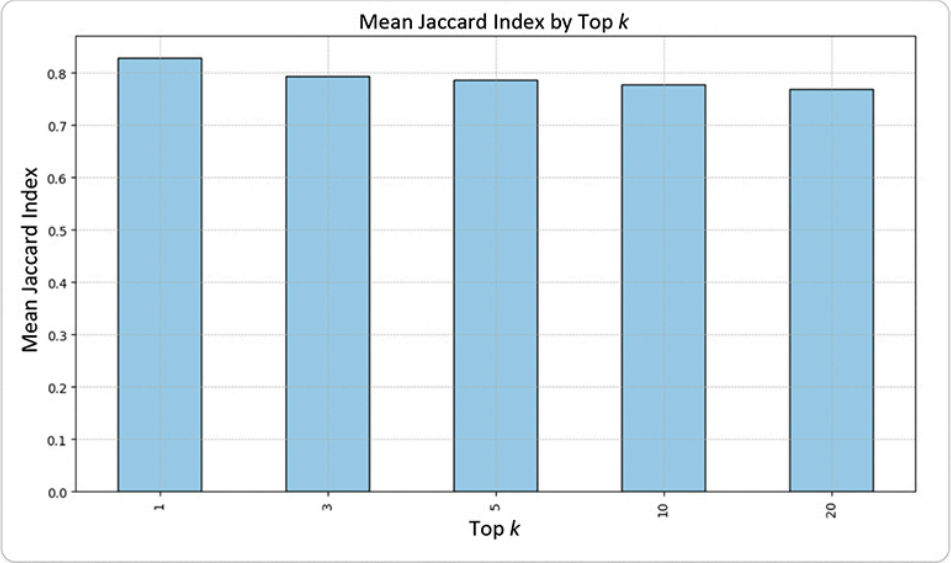


Figure 11.2 The Jaccard similarity between the top k predicted tokens of the quantized and non-quantized model on a subset of MMLU-Virology. Said another way, if we looked at the top 3 next

best tokens from each model on a sample of data, on average the Jaccard similarity of those sets of tokens would be roughly .8

Benchmarking Quantized Models

Considerations 1 and 2 focused on measuring the differences in raw next-token predictions both in similarity and in speed/memory usage, but neither addressed the accuracy of what those tokens represented. We saw non-insignificant differences between which tokens might be outputted, which suggests that there will be differences in benchmark performance.

Chapter 12 provides much more detail on benchmarking. For now, I'll just pass a very simple 0-shot prompt to each model on a subset of MMLU-Virology. This basically means I asked the question with no examples in the prompt and didn't ask for any chain of thought. I measured the words per minute (which I expected to be better for the quantized model) and the accuracy on the multiple-choice questions. The results are shown in **Figure 11.3**

Note

The only inference parameter I set was a temperature of 0.1 to induce some more consistency and reproducibility of the experiment. This choice will also highlight any token differences by making the differences in token probabilities sharper.

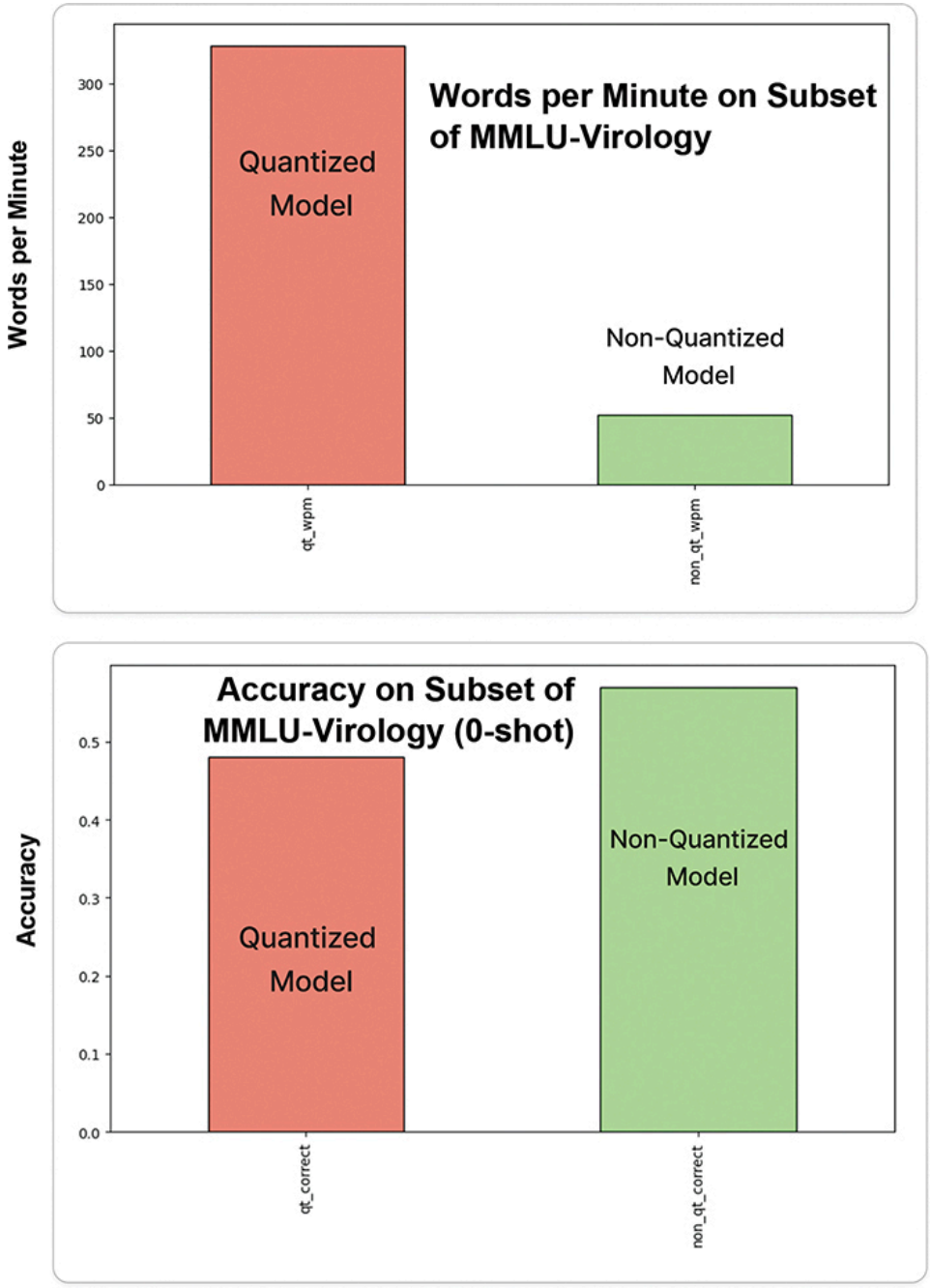


Figure 11.3 The quantized model (red in both graphs) has a better word per minute rate (top) but performs slightly worse in terms of accuracy on a subset of the MMLU benchmark (bottom).

Right out of the gate, the non-quantized model is performing slightly better on this benchmark subset but has a much lower word per minute rate; that comes as no surprise given the forward pass calculations we performed to address consideration 1. The difference in performance comes down to the fact that quantization is objectively altering the model from how it was trained, which will likely lead to degradation on test set performance. It won't always be true that the quantized version of a

model will perform worse on a test dataset, but it's always good to test for it (**foreshadows the next chapter**).

Quantization offers tangible benefits in terms of reducing memory usage and enhancing the speed of computations. This has been demonstrated effectively in the case of Llama-3-8B, where quantized models significantly outperform their non-quantized counterparts in memory efficiency and processing speed during inference. However, quantization does come with built-in trade-offs. The alterations in precision can lead to differences in token output and potentially affect performance on benchmarks and practical applications. The balance between efficiency and accuracy must be carefully tested and managed.

To make our models smaller and faster while retaining their performance characteristics, we can use our fine-tuning knowledge to transfer knowledge from larger LLMs into smaller, more lightweight versions of themselves. That results in two versions of the same model: one larger and one smaller. This process is known as knowledge distillation.

Knowledge Distillation

Distillation is a process used to create a smaller (student) model that tries to mimic the behavior of a larger (teacher) model or an ensemble of models. This results in a more compact model that can match the performance of the teacher and run more efficiently, which is very beneficial when deploying in resource-limited environments, such as on a browser or a smartphone.

We have seen distilled models elsewhere in this book. Notably, we have trained DistilBERT—a distilled version of BERT—as a faster and cheaper (computationally) alternative to the original model. We often use distilled LLMs to get more AI bang for our buck.

Task-Specific Versus Task-Agnostic Distillation

Suppose we have a complex LLM that has been trained to take in anime descriptions and output genre labels (the teacher), and we want to create a smaller, more efficient model (the student) that can generate similar descriptions. We could simply train the student model (e.g., DistilBERT) from scratch using labeled data to predict the output of the teacher model. This involves adjusting the student model's weights based on both the teacher model's output and the ground truth labels. This approach is called **task-agnostic distillation**, as the model was distilled prior to seeing any task-related data. We could also perform **task-specific distillation**, in which the student model is fine-tuned on both ground truth labels *and* the teacher model's output to get more performance from the student model

by giving it multiple sources of knowledge. **Figure 11.4** outlines the high-level differences between our two distillation approaches.

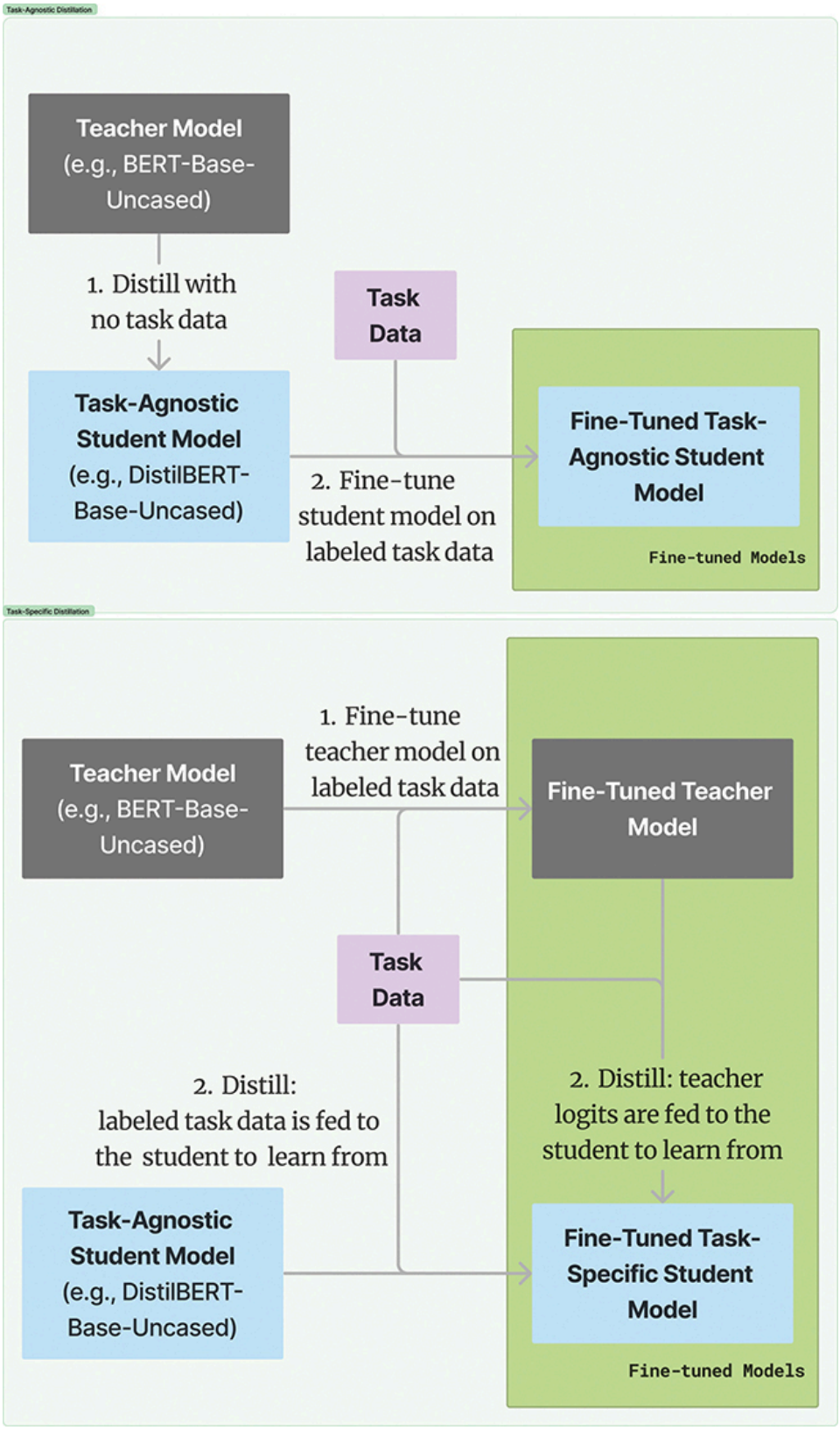


Figure 11.4 Task-agnostic distillation (top) first distills an un-fine-tuned model first and then fine-tunes the resulting distilled smaller model on task-specific data. The labeled data is used only once to fine-tune the student model. In contrast, task-specific distillation (bottom) distills a larger fine-tuned teacher model into a smaller student model by training a student model to both model the training data and match the teacher model's predictions on the same data. This way, the labeled data is used both to fine-tune the teacher and to fine-tune the student.

Both methods have their merits, and the choice between them depends on factors such as the available computational resources, the complexity of the teacher model, and the performance requirements of the student model. Let's see an example of performing a task-specific distillation using our handy-dandy anime genre predictor from **Chapter 10**.

Case Study: Distilling Our Anime Genre Predictor

In this example, we will define a custom subclass of a Hugging Face Trainer object as well as the training arguments needed to define two new hyperparameters. **Listing 11.4** expands the `Trainer` and `TrainingArguments` classes to support knowledge distillation. The code contains several key features:

- **`DistillationTrainingArguments`:** This class extends the `TrainingArguments` class of the Transformers library, adding two additional hyperparameters specific to knowledge distillation: `alpha` and `temperature`. **`alpha`** is a weighting factor that controls the balance between the original task loss (e.g., cross-entropy loss for classification tasks) and the distillation loss, whereas **`temperature`** is a hyperparameter used to control the "softness" of the probability distributions of model outputs, with higher values leading to softer distributions. **Figure 11.5** shows an example of softening a probability distribution using the temperature hyperparameter.
- **`DistillationTrainer`:** This class extends the `Trainer` class of the Transformers library. It adds a new argument `teacher_model`, which refers to the pre-trained model from which the student model learns.
- **Custom loss computation:** In the `compute_loss` function of `DistillationTrainer`, the total loss is computed as a weighted combination of the student's original loss and a distillation loss. The distillation loss is calculated as the Kullback–Leibler (KL) divergence between the softened output distributions of the student and teacher models. This is the same KL-divergence we discussed in **Chapter 10**.

These modified training classes leverage the knowledge contained in the larger, more complex model (the teacher) to improve the performance of a smaller, more efficient model (the student), even when the student model is already pre-trained and fine-tuned on a specific task.

Listing 11.4 **Defining distillation training arguments and trainer**

Click here to view code image

```
from transformers import TrainingArguments, Trainer
import torch
import torch.nn as nn
import torch.nn.functional as F

# Custom TrainingArguments class to add distillation-specific parameters
class DistillationTrainingArguments(TrainingArguments):
 def __init__(self, *args, alpha=0.5, temperature=2.0, **kwargs):
 super().__init__(*args, **kwargs)

 # alpha is the weight for the original student loss
 # Higher value means more focus on the student's original task
 self.alpha = alpha

 # temperature softens the probability distributions before calculating distill
loss
 # Higher value makes the distribution more uniform, carrying more information
the teacher model's outputs
 self.temperature = temperature

# Custom Trainer class to implement knowledge distillation
class DistillationTrainer(Trainer):
 def __init__(self, *args, teacher_model=None, **kwargs):
 super().__init__(*args, **kwargs)

 # The teacher model, a pre-trained model that the student model will learn fro
 self.teacher = teacher_model

 # Move the teacher model to the same device as the student model
 # This is necessary for the computations in the forward pass
 self._move_model_to_device(self.teacher, self.model.device)

 # Set teacher model to eval mode because we want to use it only for inference,
for training
 self.teacher.eval()

 def compute_loss(self, model, inputs, return_outputs=False):
 # Compute the output of the student model on the inputs
 outputs_student = model(**inputs)
 # Original loss of the student model (e.g., cross-entropy for classification)
 student_loss = outputs_student.loss

 # Compute the output of the teacher model on the inputs
 # We don't need gradients for the teacher model, so we use torch.no_grad to av
unnecessary computations
 with torch.no_grad():
 outputs_teacher = self.teacher(**inputs)

 # Check that the sizes of the student and teacher outputs match
 assert outputs_student.logits.size() == outputs_teacher.logits.size()

 # Kullback-Leibler divergence loss function, comparing the softened output
```

```
distributions of the student and teacher models
 loss_function = nn.KLDivLoss(reduction="batchmean")

 # Calculate the distillation loss between the student and teacher outputs
 # We apply log_softmax to the student's outputs and softmax to the teacher's o
before calculating the loss
 # This is due to the expectation of log probabilities for the input and probab
for the target in nn.KLDivLoss
 loss_logits = (loss_function(
 F.log_softmax(outputs_student.logits / self.args.temperature, dim=-1),
 F.softmax(outputs_teacher.logits / self.args.temperature, dim=-1)) * (self.arg
temperature ** 2))

 # The total loss is a weighted combination of the student's original loss and
distillation loss
 loss = self.args.alpha * student_loss + (1. - self.args.alpha) * loss_logits

 # Depending on the return_outputs parameter, return either the loss alone or t
and the student's outputs
 return (loss, outputs_student) if return_outputs else loss
```

A Bit More on Temperature

We have seen the temperature variable before, when it was used to control the "randomness" of GPT-like models. In general, temperature is a hyperparameter that is used to control the "softness" of the probability distribution. Let's break down the role of the temperature in the context of knowledge distillation:

- **Softening the distribution:** The `softmax` function is used to transform the logits from the teacher and student models into a probability distribution. When you divide the logits by the temperature before applying `softmax`, this effectively "softens" the distribution. A higher temperature will make the distribution more uniform (i.e., closer to equal probabilities for all classes), whereas a lower temperature will make it more "peaked" (i.e., a higher probability for the most likely class and lower probabilities for all other classes). In the context of distillation, a softer distribution (higher temperature) carries more information about the relative probabilities of the non-maximum classes, which can help the student model learn more effectively from the teacher. Conversely, a sharper distribution (lower temperature) will enforce the discrimination between classes by making the most likely class even more likely. Put another way, the higher the temperature, the more the student will be able to capture subtle differences between classes, whereas a lower temperature emphasizes the correct class more strongly, potentially aiding in precise discrimination. **Figure 11.5** shows how the temperature visually affects our `softmax` values.

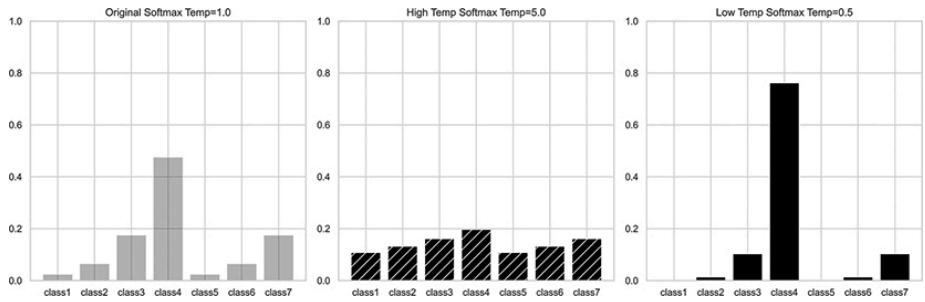


Figure 11.5 Illustrating the effect of the temperature on the `softmax` output of a set of example logits. The leftmost graph, titled "Original Softmax Temp=1.0," depicts the `softmax` probabilities using a default temperature of 1.0. These are our original `softmax` values for classes—for example, tokens to predict when autoregressively language modeling. The middle graph, "High Temp Softmax Temp=5.0," shows the distribution with a relatively high temperature setting of 5.0, which softens the probability distribution, making it appear more uniform. In a language modeling example, this effect makes tokens that would have been less likely to be chosen from the original distribution, more likely to be chosen. For an AI product, this change is often described as making the LLM more deterministic and "creative." The rightmost graph, "Low Temp Softmax Temp=0.5," shows the output of the `softmax` function with a lower temperature setting of 0.5. This creates a more "peaked" distribution, assigning a higher probability to the most likely class while all other classes receive significantly lower probabilities. As a result, the model is considered less deterministic and less "creative."

- **Temperature-squared in the loss function:** The Kullback–Leibler divergence part of the loss function includes a temperature-squared term. This term can be seen as a scaling factor for the distillation loss, which corrects for the change in scale of the logits caused by dividing them by the temperature. Without this correction, the gradients during back-propagation would be smaller when the temperature is higher, potentially slowing down training. By including the temperature-squared term, the scale of the gradients is kept more consistent regardless of the temperature value.
- **Dividing by the temperature in the loss function:** As mentioned earlier, dividing the logits by the temperature before applying `softmax` is used to soften the probability distributions. This is done separately for both the teacher and student model's logits in the loss function.

The temperature is used to control the balance between transferring knowledge about the hard targets (e.g., genre prediction labels) and the soft targets (the teacher's predictions for genre) during the distillation process. Its value needs to be carefully chosen and may require some experimentation or validation on a development set.

Running the Distillation Process

Running the training process with our modified classes is a breeze. We simply have to define a teacher model (which I trained off-screen using a BERT large-uncased model), a student model 289(a DistilBERT model), a

tokenizer, and a data collator. Note that I'm choosing teacher and student models that share a tokenizing schema and token IDs. Although distilling models from one token space to another is possible, it's much more difficult—so I chose the easier route here.

Listing 11.5 highlights some of the major code snippets to get the training going.

Listing 11.5 **Running our distillation process**

Click here to view code image

```
# Define teacher model
trained_model = AutoModelForSequenceClassification.from_pretrained(
 f"genre-prediction", problem_type="multi_label_classification",
)

# Define student model
student_model = AutoModelForSequenceClassification.from_pretrained(
 'distilbert-base-uncased',
 num_labels=len(unique_labels),
 id2label=id2label,
 label2id=label2id,
)

# Define training args
training_args = DistillationTrainingArguments(
 output_dir='distilled-genre-prediction',
 evaluation_strategy = "epoch",
 save_strategy = "epoch",
 num_train_epochs=10,
 logging_steps=50,
 per_device_train_batch_size=16,
 gradient_accumulation_steps=4,
 per_device_eval_batch_size=64,
 load_best_model_at_end=True,
 alpha=0.5,
 temperature=4.0,
 fp16=True
 )

distil_trainer = DistillationTrainer(
 student_model,
 training_args,
 teacher_model=trained_model,
 train_dataset=description_encoded_dataset["train"],
 eval_dataset=description_encoded_dataset["test"],
 data_collator=data_collator,
 tokenizer=tokenizer,
 compute_metrics=compute_metrics,
)
```

```
distil_trainer.train()
```

Summary of Distillation Results

We have three models to compare here:

- **The teacher model:** A BERT large-uncased model trained on the standard loss to predict genres.
- **The task-agnostic distilled student model:** A DistilBERT model that was distilled from the BERT base-uncased model, and then fed training data in a manner identical to the teacher model.
- **The task-specific distilled student model:** A DistilBERT model that was distilled from both the BERT base-uncased model and the teacher's knowledge. It is fed the same data as the other two models but is judged on two fronts—the loss from the actual task and the loss from being too different from the teacher (the KL divergence).

Figure 11.6 shows the Jaccard score (a measure where a higher value indicates a greater similarity and therefore greater accuracy) for our three models trained over 10 epochs. We can see that the task-specific student model excels over the task-agnostic student model and even performs better than the teacher model in earlier epochs. The teacher model still performs the best in terms of Jaccard similarity over three epochs, but that won't be our only metric.

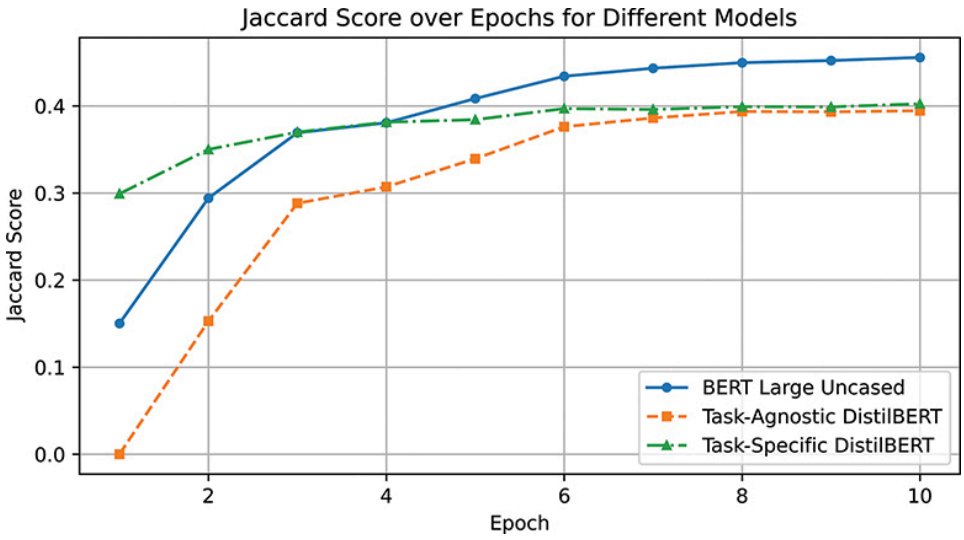


Figure 11.6 Our teacher model performs the best of all three models, which comes as no surprise. Note that our task-specific DistilBERT model performs better than our task-agnostic DistilBERT model.

Performance on genre prediction may not be our only consideration. **Figure 11.7** highlights just how similar the task-specific model is to the teacher model in terms of performance, and also shows the difference in memory usage and speed of the models.

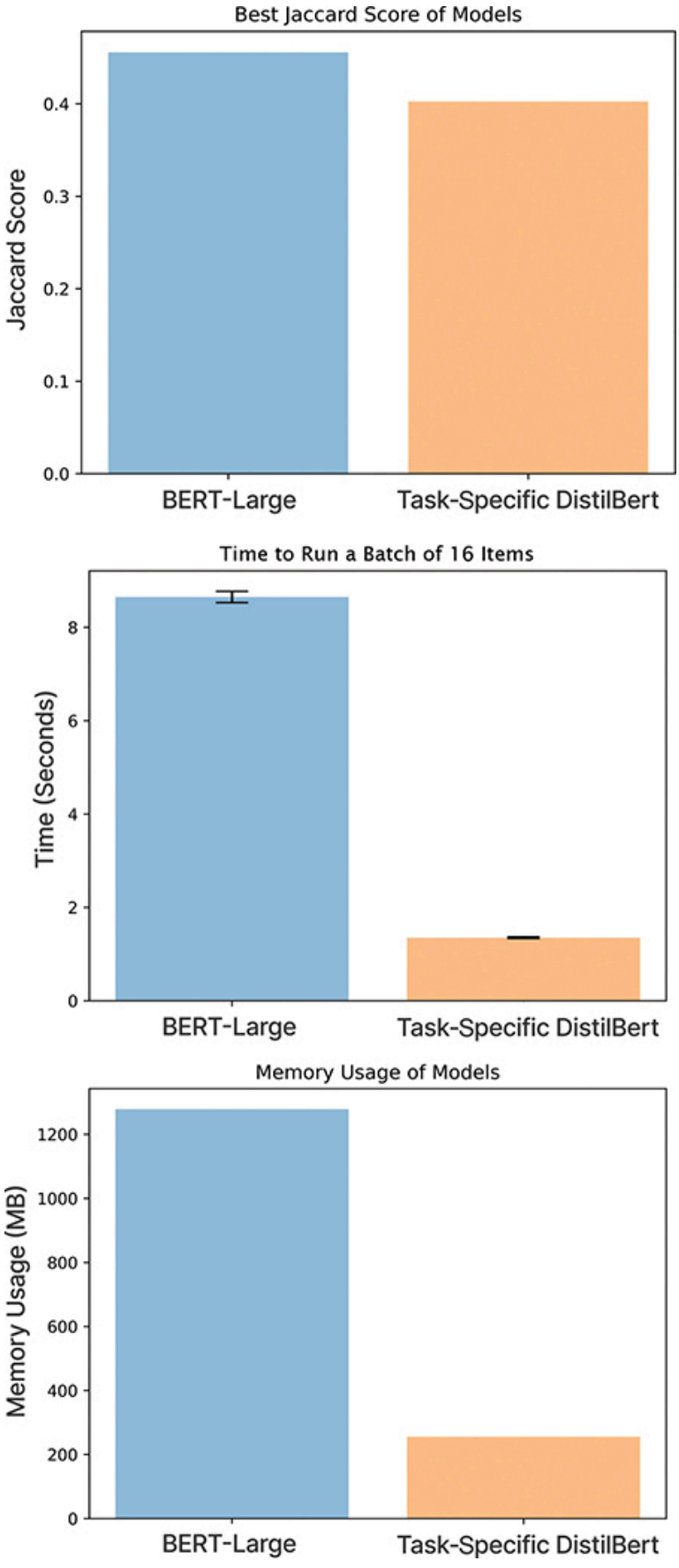


Figure 11.7 Our student model is 4 to 6 times faster and more memory efficient, while being only slightly less accurate.

Overall, our task-specific distilled model performs better than our task-agnostic model and is about 4 to 6 times more efficient than our teacher model in terms of memory usage and speed. It can even be the case that the student model outperforms the teacher.

The Student Becomes the Master

Although most cases of distillation yield a less accurate student model, this isn't always the case. In a separate instance of distillation using a different dataset, I trained a BERT-large-cased model against the `go_emotions` dataset from Hugging Face—58,000 curated Reddit comments labeled with 27 emotion categories. In this case, the task-specific distilled model outperformed the teacher model with only 3 epochs of training (**Figure 11.8**).

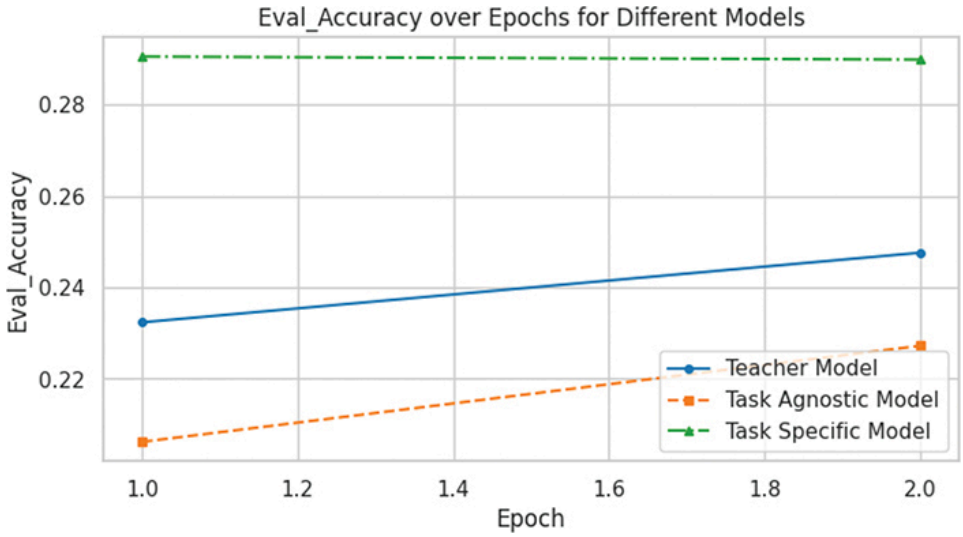


Figure 11.8 Sometimes a student model can outperform a teacher, but this is rare! It often means that the task itself didn't require as many parameters as the teacher has. In this case, a BERT-large model trained on the `go_emotions` dataset underperformed against the task-specific distilled model, but still outperformed the task-agnostic distilled model.

With any LLM, the training, testing, and experimentation can often be the fun part—but what comes next will make or break our ability to use a model in the long run. We need to think about cost-projecting inference, licensing data, and deploying options.

Cost Projections with LLMs

In the case of open-source models, cost projections involve considering both the compute and storage resources required to host and run the model:

- **Compute costs:** Include the costs of the machines (virtual machines or dedicated hardware) where the model will be running. Factors such as the machine's CPU, GPU, memory, and network capabilities, as well as the region and the running time, will affect this cost.
- **Storage costs:** Include the costs to store the model's weights and biases and any data that the model needs for inference. These costs will depend on the size of the model and data, the storage type (e.g., SSD versus HDD), and the region. If you store multiple versions of the model, they can really add up.
- **Scaling costs:** If you intend to serve a high volume of requests, you may need to use load balancing and auto-scaling solutions, which come with additional costs.
- **Maintenance costs:** The costs associated with monitoring and maintaining your deployment, such as logging, alerting, debugging, and updating the model.

Predicting these costs accurately requires a comprehensive understanding of your application's requirements, the chosen cloud provider's pricing structure, and the model's resource needs. Often, it's wise to leverage cost estimation tools provided by cloud services, perform small-scale tests to gather metrics, or consult with cloud solution architects to obtain a more accurate projection.

Pushing to Hugging Face

We have been using Hugging Face's models enough to finally consider sharing our open-source, fine-tuned models to the world via Hugging Face's platform, with the aim of providing wider visibility of the models and their ease of use to the community. If you are inclined to use Hugging Face as a repository, you'll need to follow the steps outlined here.

Preparing the Model

Before you can push your model, ensure that it's appropriately fine-tuned and saved in a format compatible with Hugging Face. You can use the

`save_pretrained()` function (shown in **Listing 11.6**) in the Hugging Face Transformers library for this purpose.

Listing 11.6 **Saving models and tokenizers to disk**

Click here to view code image

```
from transformers import BertModel, BertTokenizer

# Assuming you have a fine-tuned model and tokenizer
model = BertModel.from_pretrained("bert-base-uncased")
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

# Save the model and tokenizer
model.save_pretrained("<your-path>/my-fine-tuned-model")
tokenizer.save_pretrained("<your-path>/my-fine-tuned-model")
```

Think About Licensing

You have to specify a license for your model when you upload it to a repository. The license informs users about what they can and cannot do with your model. Popular licenses include Apache 2.0, MIT, and GNU GPL v3. You should include a LICENSE file in the model repository.

Here is a bit more information on each of the three licenses just mentioned:

- **Apache 2.0:** The Apache License 2.0 allows users to freely use, reproduce, distribute, display, and perform the work, as well as make derivative works. The conditions are that any distribution should include a copy of the original Apache 2.0 license, state any changes made, and include a NOTICE file if one exists. In addition, while it allows the use of patent claims, this license does not provide an express grant of patent rights from contributors.
- **MIT:** The MIT License is a permissive free software license, which means it permits reuse within proprietary software provided all copies of the licensed software include a copy of the MIT License terms. This means that you can use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the software, provided you include the necessary copyright and permission notices.
- **GNU GPL v3:** The GNU General Public License (GPL) is a copyright license that requires any work that is distributed or published, and that in whole or in part contains or is derived from the program or any part of it, to be licensed as a whole at no charge to all third parties under the terms of GPL v3. This license ensures that all users who re-

ceive a copy of the work also receive the freedoms to use, modify, and distribute the original work. However, it requires that any modifications also be licensed under the same terms, which is not required by the MIT or Apache licenses.

Writing the Model Card

A model card serves as the primary documentation for your model. It provides information about the model's purpose, capabilities, limitations, and performance. Essential components of a model card include the following items:

- **Model description:** Details about what the model does and how it was trained.
- **Dataset details:** Information about the data used to train and validate the model.
- **Evaluation results:** Details about the model's performance on various tasks.
- **Usage examples:** Code snippets showing how to use the model.
- **Limitations and biases:** Any known limitations or biases in the model.

The model card, a markdown file named README.md, should be located in the model's root directory. The Hugging Face trainer also offers a way to automatically create these using `trainer.create_model_card()`. You should plan to add more to this automatically generated markdown file, as otherwise it will include only basic information like the model name and final metrics.

Pushing the Model to a Repository

The Hugging Face Transformers library has a `push_to_hub` feature that allows users to easily upload their models directly to the Hugging Face Model Hub. **Listing 11.7** provides an example of this feature's use.

Listing 11.7 **Pushing models and tokenizers to Hugging Face**

Click here to view code image

```
from transformers import BertModel, BertTokenizer

# Assuming you have a fine-tuned model and tokenizer
model = BertModel.from_pretrained("bert-base-uncased")
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

# Save the model and tokenizer to a directory
model.save_pretrained("my-fine-tuned-model")
```

```
tokenizer.save_pretrained("my-fine-tuned-model")

# Push the model to the Hub
model.push_to_hub("my-fine-tuned-model")
tokenizer.push_to_hub("my-fine-tuned-model")
```

This script authenticates your Hugging Face credentials, saves your fine-tuned model and tokenizer to a directory, and then pushes them to the Hub. The `push_to_hub` method takes the name of the model's repository as a parameter.

You can also log in separately using the Hugging Face CLI and the command `huggingface-cli login`, or you can use the `huggingface_hub` package to interact with the hub programmatically to save your credentials locally (although the code provided in the listing should prompt you to log in without doing this). Note that this example assumes that you've already created a repository on the Hugging Face Model Hub with the name "my-fine-tuned-model." If the repository does not exist, you'll need to create it first or use the `repository_name` argument when calling `push_to_hub`.

Using Hugging Face Inference Endpoints to Deploy Models

After we push our model to the Hugging Face repository, we can use its **inference endpoint** product for easy deployment on a dedicated, fully managed infrastructure. This service enables the creation of production-ready APIs without requiring users to deal with containers, GPUs, or really any MLOps. It operates on a pay-as-you-go basis for the raw computing power used, helping to keep production costs down.

Figure 11.9 shows a screenshot of an inference endpoint I made for the DistilBERT-based sequence classifier we created in **Chapter 5**. It's based on the `app_reviews` dataset, which costs only about $23 per month ($0.032 per hour × 24 hours per day × roughly 30 days per month).

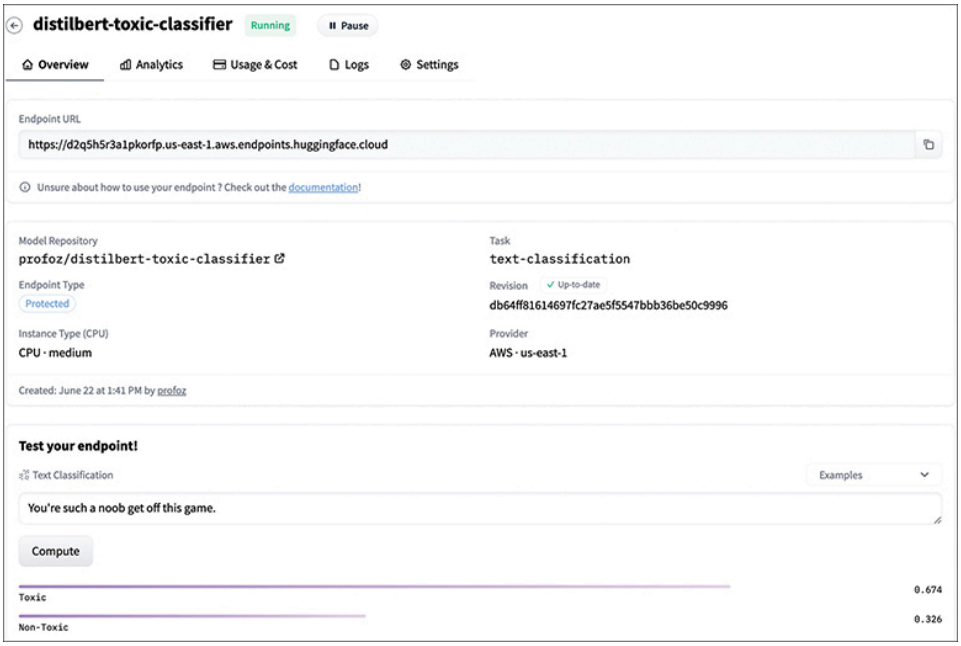


Figure 11.9 An inference endpoint for the model we fine-tuned in **Chapter 5** to predict the number of stars given to an app based on the written review.

Listing 11.8 shows an example of using this endpoint to handle requests.

Listing 11.8 **Using a Hugging Face inference endpoint to classify text**

Click here to view code image

```
import requests, json

# The URL of a Hugging Face inference endpoint. Replace with your own.
API_URL = "https://t7gvgsj77yrypla7.us-east-1.aws.endpoints.huggingface.cloud"

# This would need a 'HF_API_KEY' if the API were not public as I made mine to b
headers = {
         "Accept" : "application/json",
         "Content-Type": "application/json"
}
# The data we want to send in our HTTP request.
data = {
         "inputs": "I hate this app",
         "parameters": {
                  "top_k": 5 # the number of classes we have
          }
}
# Make a POST request to the Hugging Face API with our headers and data.
response = requests.post(API_URL, headers=headers, data=json.dumps(data))
```

```
# Print the response from the server.
print(response.json())
[
  {'label': 'LABEL_0', 'score': 0.8901807069778442},
  {'label': 'LABEL_4', 'score': 0.056254707276821136},
  {'label': 'LABEL_1', 'score': 0.03358633071184158},
  {'label': 'LABEL_2', 'score': 0.012845375575125217},
  {'label': 'LABEL_3', 'score': 0.007132874336093664}
]
```

Deploying ML models to the cloud is its own behemoth of a topic. Obviously, the discussion here omits a ton of work on MLOps processes, monitoring dashboards, and continuous training pipelines. Even so, it should be enough to get you started with your deployed models.

Summary

Techniques like quantization and distillation can yield smaller, more memory-efficient models that retain or even exceed the performance of the original LLMs. Deploying LLMs is itself a large task. Depending on which cloud provider you're most comfortable with and what kinds of features providers offer, you can choose whichever provider you like.

In our final chapter, we will look closely at a topic we've been using all along in this book but will finally pick apart and interrogate—LLM evaluation.