CHAPTER 8

Making Transformers Efficient in Production

In the previous chapters, you've seen how transformers can be fine-tuned to produce great results on a wide range of tasks. However, in many situations accuracy (or whatever metric you're optimizing for) is not enough; your state-of-the-art model is not very useful if it's too slow or large to meet the business requirements of your application. An obvious alternative is to train a faster and more compact model, but the reduction in model capacity is often accompanied by a degradation in performance. So what can you do when you need a fast, compact, yet highly accurate model?

In this chapter we will explore four complementary techniques that can be used to speed up the predictions and reduce the memory footprint of your transformer models: *knowledge distillation*, *quantization*, *pruning*, and *graph optimization* with the Open Neural Network Exchange (ONNX) format and ONNX Runtime (ORT). We'll also see how some of these techniques can be combined to produce significant performance gains. For example, this was the approach taken by the Roblox engineering team in their article "How We Scaled Bert to Serve 1+ Billion Daily Requests on CPUs", who as shown in Figure 8-1 found that combining knowledge distillation and quantization enabled them to improve the latency and throughput of their BERT classifier by over a factor of 30!

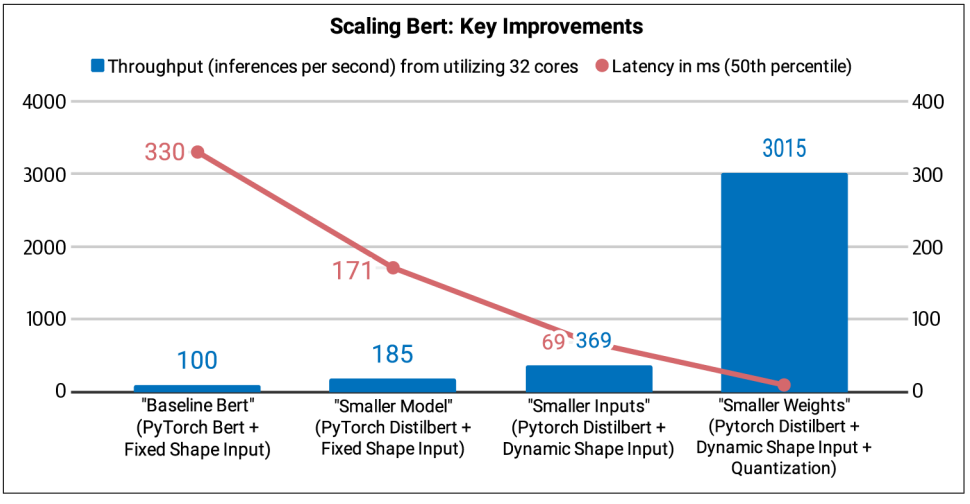


Figure 8-1. How Roblox scaled BERT with knowledge distillation, dynamic padding, and weight quantization (photo courtesy of Roblox employees Quoc N. Le and Kip Kaehler)

To illustrate the benefits and trade-offs associated with each technique, we'll use intent detection as a case study; this is an important component of text-based assistants, where low latencies are critical for maintaining a conversation in real time. Along the way you'll learn how to create custom trainers, perform efficient hyperparameter search, and gain a sense of what it takes to implement cutting-edge research with 🤗 Transformers. Let's dive in!

Intent Detection as a Case Study

Let's suppose that we're trying to build a text-based assistant for our company's call center so that customers can request their account balance or make bookings without needing to speak with a human agent. In order to understand the goals of a customer, our assistant will need to be able to classify a wide variety of natural language text into a set of predefined actions or *intents*. For example, a customer might send a message like the following about an upcoming trip:

> Hey, I'd like to rent a vehicle from Nov 1st to Nov 15th in Paris and I need a 15 passenger van

and our intent classifier could automatically categorize this as a *Car Rental* intent, which then triggers an action and response. To be robust in a production environment, our classifier will also need to be able to handle *out-of-scope* queries, where a customer makes a query that doesn't belong to any of the predefined intents and the system should yield a fallback response. For example, in the second case shown in Figure 8-2, a customer asks a question about sports (which is out of scope), and the text assistant mistakenly classifies it as one of the known in-scope intents and returns

the payday response. In the third case, the text assistant has been trained to detect out-of-scope queries (usually labeled as a separate class) and informs the customer about which topics it can answer questions about.

Figure 8-2. Three exchanges between a human (right) and a text-based assistant (left) for personal finance (courtesy of Stefan Larson et al.)

As a baseline, we've fine-tuned a BERT-base model that achieves around 94% accuracy on the CLINC150 dataset.[1] This dataset includes 22,500 in-scope queries across 150 intents and 10 domains like banking and travel, and also includes 1,200 out-of-scope queries that belong to an `oos` intent class. In practice we would also gather our own in-house dataset, but using public data is a great way to iterate quickly and generate preliminary results.

To get started, let's download our fine-tuned model from the Hugging Face Hub and wrap it in a pipeline for text classification:

```
from transformers import pipeline

bert_ckpt = "transformersbook/bert-base-uncased-finetuned-clinc"
pipe = pipeline("text-classification", model=bert_ckpt)
```

Now that we have a pipeline, we can pass a query to get the predicted intent and confidence score from the model:

1 S. Larson et al., "An Evaluation Dataset for Intent Classification and Out-of-Scope Prediction", (2019).

```
query = """Hey, I'd like to rent a vehicle from Nov 1st to Nov 15th in
Paris and I need a 15 passenger van"""
pipe(query)
```

```
[{'label': 'car_rental', 'score': 0.549003541469574}]
```

Great, the `car_rental` intent makes sense. Let's now look at creating a benchmark that we can use to evaluate the performance of our baseline model.

Creating a Performance Benchmark

Like other machine learning models, deploying transformers in production environments involves a trade-off among several constraints, the most common being:[2]

Model performance
: How well does our model perform on a well-crafted test set that reflects production data? This is especially important when the cost of making errors is large (and best mitigated with a human in the loop), or when we need to run inference on millions of examples and small improvements to the model metrics can translate into large gains in aggregate.

Latency
: How fast can our model deliver predictions? We usually care about latency in real-time environments that deal with a lot of traffic, like how Stack Overflow needed a classifier to quickly detect unwelcome comments on the website.

Memory
: How can we deploy billion-parameter models like GPT-2 or T5 that require gigabytes of disk storage and RAM? Memory plays an especially important role in mobile or edge devices, where a model has to generate predictions without access to a powerful cloud server.

Failing to address these constraints can have a negative impact on the user experience of your application. More commonly, it can lead to ballooning costs from running expensive cloud servers that may only need to handle a few requests. To explore how each of these constraints can be optimized with various compression techniques, let's begin by creating a simple benchmark that measures each quantity for a given pipeline and test set. A skeleton of what we'll need is given by the following class:

```
class PerformanceBenchmark:
    def __init__(self, pipeline, dataset, optim_type="BERT baseline"):
        self.pipeline = pipeline
```

2 As described by Emmanuel Ameisen in *Building Machine Learning Powered Applications* (O'Reilly), business or product metrics are the *most* important ones to consider. After all, it doesn't matter how accurate your model is if it doesn't solve a problem your business cares about. In this chapter we'll assume that you have already defined the metrics that matter for your application and focus on optimizing the model metrics.

```
        self.dataset = dataset
        self.optim_type = optim_type

    def compute_accuracy(self):
        # We'll define this later
        pass

    def compute_size(self):
        # We'll define this later
        pass

    def time_pipeline(self):
        # We'll define this later
        pass

    def run_benchmark(self):
        metrics = {}
        metrics[self.optim_type] = self.compute_size()
        metrics[self.optim_type].update(self.time_pipeline())
        metrics[self.optim_type].update(self.compute_accuracy())
        return metrics
```

We've defined an `optim_type` parameter to keep track of the different optimization techniques that we'll cover in this chapter. We'll use the `run_benchmark()` method to collect all the metrics in a dictionary, with keys given by `optim_type`.

Let's now put some flesh on the bones of this class by computing the model accuracy on the test set. First we need some data to test on, so let's download the CLINC150 dataset that was used to fine-tune our baseline model. We can get the dataset from the Hub with 🤗 Datasets as follows:

```
from datasets import load_dataset

clinc = load_dataset("clinc_oos", "plus")
```

Here, the `plus` configuration refers to the subset that contains the out-of-scope training examples. Each example in the CLINC150 dataset consists of a query in the `text` column and its corresponding intent. We'll use the test set to benchmark our models, so let's take a look at one of the dataset's examples:

```
sample = clinc["test"][42]
sample
```

```
{'intent': 133, 'text': 'transfer $100 from my checking to saving account'}
```

The intents are provided as IDs, but we can easily get the mapping to strings (and vice versa) by accessing the `features` attribute of the dataset:

```
intents = clinc["test"].features["intent"]
intents.int2str(sample["intent"])
```

```
'transfer'
```

Now that we have a basic understanding of the contents in the CLINC150 dataset, let's implement the `compute_accuracy()` method of `PerformanceBenchmark`. Since the dataset is balanced across the intent classes, we'll use accuracy as our metric. We can load this metric with 🤗 Datasets as follows:

```
from datasets import load_metric

accuracy_score = load_metric("accuracy")
```

The accuracy metric expects the predictions and references (i.e., the ground truth labels) to be integers. We can use the pipeline to extract the predictions from the `text` field and then use the `str2int()` method of our `intents` object to map each prediction to its corresponding ID. The following code collects all the predictions and labels in lists before returning the accuracy on the dataset. Let's also add it to our `PerformanceBenchmark` class:

```
def compute_accuracy(self):
    """This overrides the PerformanceBenchmark.compute_accuracy() method"""
    preds, labels = [], []
    for example in self.dataset:
        pred = self.pipeline(example["text"])[0]["label"]
        label = example["intent"]
        preds.append(intents.str2int(pred))
        labels.append(label)
    accuracy = accuracy_score.compute(predictions=preds, references=labels)
    print(f"Accuracy on test set - {accuracy['accuracy']:.3f}")
    return accuracy

PerformanceBenchmark.compute_accuracy = compute_accuracy
```

Next, let's compute the size of our model by using the `torch.save()` function from PyTorch to serialize the model to disk. Under the hood, `torch.save()` uses Python's `pickle` module and can be used to save anything from models to tensors to ordinary Python objects. In PyTorch, the recommended way to save a model is by using its `state_dict`, which is a Python dictionary that maps each layer in a model to its learnable parameters (i.e., weights and biases). Let's see what is stored in the `state_dict` of our baseline model:

```
list(pipe.model.state_dict().items())[42]
```

```
('bert.encoder.layer.2.attention.self.value.weight',
 tensor([[-1.0526e-02, -3.2215e-02,  2.2097e-02,  ..., -6.0953e-03,
           4.6521e-03,  2.9844e-02],
         [-1.4964e-02, -1.0915e-02,  5.2396e-04,  ...,  3.2047e-05,
          -2.6890e-02, -2.1943e-02],
         [-2.9640e-02, -3.7842e-03, -1.2582e-02,  ..., -1.0917e-02,
           3.1152e-02, -9.7786e-03],
         ...,
         [-1.5116e-02, -3.3226e-02,  4.2063e-02,  ..., -5.2652e-03,
           1.1093e-02,  2.9703e-03],
```

```
          [-3.6809e-02,  5.6848e-02, -2.6544e-02,  ..., -4.0114e-02,
            6.7487e-03,  1.0511e-03],
          [-2.4961e-02,  1.4747e-03, -5.4271e-02,  ...,  2.0004e-02,
            2.3981e-02, -4.2880e-02]]))
```

We can clearly see that each key/value pair corresponds to a specific layer and tensor in BERT. So if we save our model with:

```
torch.save(pipe.model.state_dict(), "model.pt")
```

we can then use the `Path.stat()` function from Python's `pathlib` module to get information about the underlying files. In particular, `Path("model.pt").stat().st_size` will give us the model size in bytes. Let's put this all together in the `compute_size()` function and add it to `PerformanceBenchmark`:

```
import torch
from pathlib import Path

def compute_size(self):
    """This overrides the PerformanceBenchmark.compute_size() method"""
    state_dict = self.pipeline.model.state_dict()
    tmp_path = Path("model.pt")
    torch.save(state_dict, tmp_path)
    # Calculate size in megabytes
    size_mb = Path(tmp_path).stat().st_size / (1024 * 1024)
    # Delete temporary file
    tmp_path.unlink()
    print(f"Model size (MB) - {size_mb:.2f}")
    return {"size_mb": size_mb}

PerformanceBenchmark.compute_size = compute_size
```

Finally let's implement the `time_pipeline()` function so that we can time the average latency per query. For this application, latency refers to the time it takes to feed a text query to the pipeline and return the predicted intent from the model. Under the hood the pipeline also tokenizes the text, but this is around one thousand times faster than generating the predictions and thus adds a negligible contribution to the overall latency. A simple way to measure the execution time of a code snippet is to use the `perf_counter()` function from Python's `time` module. This function has a better time resolution than the `time.time()` function and is well suited for getting precise results.

We can use `perf_counter()` to time our pipeline by passing our test query and calculating the time difference in milliseconds between the start and end:

```
from time import perf_counter

for _ in range(3):
    start_time = perf_counter()
    _ = pipe(query)
```

```
        latency = perf_counter() - start_time
        print(f"Latency (ms) - {1000 * latency:.3f}")

    Latency (ms) - 85.367
    Latency (ms) - 85.241
    Latency (ms) - 87.275
```

These results exhibit quite some spread in the latencies and suggest that timing a single pass through the pipeline can give wildly different results each time we run the code. So instead, we'll collect the latencies over many runs and then use the resulting distribution to calculate the mean and standard deviation, which will give us an idea about the spread in values. The following code does what we need and includes a phase to warm up the CPU before performing the actual timed run:

```
    import numpy as np

    def time_pipeline(self, query="What is the pin number for my account?"):
        """This overrides the PerformanceBenchmark.time_pipeline() method"""
        latencies = []
        # Warmup
        for _ in range(10):
            _ = self.pipeline(query)
        # Timed run
        for _ in range(100):
            start_time = perf_counter()
            _ = self.pipeline(query)
            latency = perf_counter() - start_time
            latencies.append(latency)
        # Compute run statistics
        time_avg_ms = 1000 * np.mean(latencies)
        time_std_ms = 1000 * np.std(latencies)
        print(f"Average latency (ms) - {time_avg_ms:.2f} +\- {time_std_ms:.2f}")
        return {"time_avg_ms": time_avg_ms, "time_std_ms": time_std_ms}

    PerformanceBenchmark.time_pipeline = time_pipeline
```

To keeps things simple, we'll use the same `query` value to benchmark all our models. In general, the latency will depend on the query length, and a good practice is to benchmark your models with queries that they're likely to encounter in production environments.

Now that our `PerformanceBenchmark` class is complete, let's give it a spin! Let's start by benchmarking our BERT baseline. For the baseline model, we just need to pass the pipeline and the dataset we wish to perform the benchmark on. We'll collect the results in the `perf_metrics` dictionary to keep track of each model's performance:

```
    pb = PerformanceBenchmark(pipe, clinc["test"])
    perf_metrics = pb.run_benchmark()

    Model size (MB) - 418.16
    Average latency (ms) - 54.20 +\- 1.91
    Accuracy on test set - 0.867
```

Now that we have a reference point, let's look at our first compression technique: knowledge distillation.

The average latency values will differ depending on what type of hardware you are running on. For example, you can usually get better performance by running inference on a GPU since it enables batch processing. For the purposes of this chapter, what's important is the relative difference in latencies between models. Once we have determined the best-performing model, we can then explore different backends to reduce the absolute latency if needed.

Making Models Smaller via Knowledge Distillation

Knowledge distillation is a general-purpose method for training a smaller *student* model to mimic the behavior of a slower, larger, but better-performing *teacher*. Originally introduced in 2006 in the context of ensemble models,[3] it was later popularized in a famous 2015 paper that generalized the method to deep neural networks and applied it to image classification and automatic speech recognition.[4]

Given the trend toward pretraining language models with ever-increasing parameter counts (the largest at the time of writing having over one trillion parameters),[5] knowledge distillation has also become a popular strategy to compress these huge models and make them more suitable for building practical applications.

Knowledge Distillation for Fine-Tuning

So how is knowledge actually "distilled" or transferred from the teacher to the student during training? For supervised tasks like fine-tuning, the main idea is to augment the ground truth labels with a distribution of "soft probabilities" from the teacher which provide complementary information for the student to learn from. For example, if our BERT-base classifier assigns high probabilities to multiple intents, then this could be a sign that these intents lie close to each other in the feature space. By training the student to mimic these probabilities, the goal is to distill some of this "dark knowledge"[6] that the teacher has learned—that is, knowledge that is not available from the labels alone.

3 C. Buciluǎ et al., "Model Compression," *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (August 2006): 535–541, *https://doi.org/10.1145/1150402.1150464*.

4 G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network", (2015).

5 W. Fedus, B. Zoph, and N. Shazeer, "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity", (2021).

6 Geoff Hinton coined this term in a talk to refer to the observation that softened probabilities reveal the hidden knowledge of the teacher.

Mathematically, the way this works is as follows. Suppose we feed an input sequence x to the teacher to generate a vector of logits $\mathbf{z}(x) = [z_1(x), ..., z_N(x)]$. We can convert these logits into probabilities by applying a softmax function:

$$\frac{\exp\left(z_i(x)\right)}{\sum_j \exp\left(z_i(x)\right)}$$

This isn't quite what we want, though, because in many cases the teacher will assign a high probability to one class, with all other class probabilities close to zero. When that happens, the teacher doesn't provide much additional information beyond the ground truth labels, so instead we "soften" the probabilities by scaling the logits with a temperature hyperparameter T before applying the softmax:[7]

$$p_i(x) = \frac{\exp\left(z_i(x)/T\right)}{\sum_j \exp\left(z_i(x)/T\right)}$$

As shown in Figure 8-3, higher values of T produce a softer probability distribution over the classes and reveal much more information about the decision boundary that the teacher has learned for each training example. When $T = 1$ we recover the original softmax distribution.

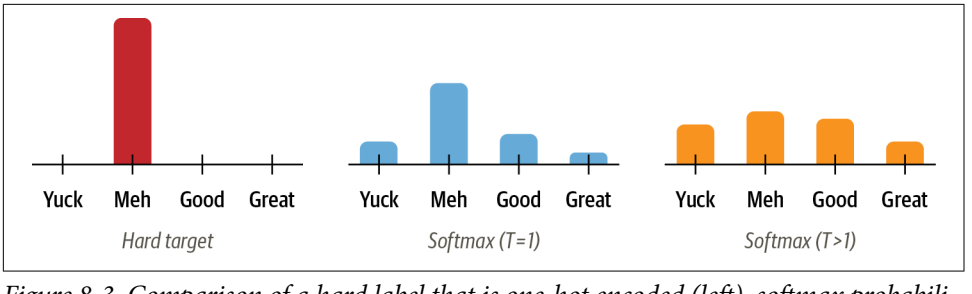


Figure 8-3. Comparison of a hard label that is one-hot encoded (left), softmax probabilities (middle), and softened class probabilities (right)

Since the student also produces softened probabilities $q_i(x)$ of its own, we can use the Kullback–Leibler (KL) divergence to measure the difference between the two probability distributions:

$$D_{KL}(p, q) = \sum_i p_i(x) \log \frac{p_i(x)}{q_i(x)}$$

7 We also encountered temperature in the context of text generation in Chapter 5.

With the KL divergence we can calculate how much is lost when we approximate the probability distribution of the teacher with the student. This allows us to define a knowledge distillation loss:

$$L_{KD} = T^2 D_{KL}$$

where T^2 is a normalization factor to account for the fact that the magnitude of the gradients produced by soft labels scales as $1/T^2$. For classification tasks, the student loss is then a weighted average of the distillation loss with the usual cross-entropy loss L_{CE} of the ground truth labels:

$$L_{\text{student}} = \alpha L_{CE} + (1 - \alpha) L_{KD}$$

where α is a hyperparameter that controls the relative strength of each loss. A diagram of the whole process is shown in Figure 8-4; the temperature is set to 1 at inference time to recover the standard softmax probabilities.

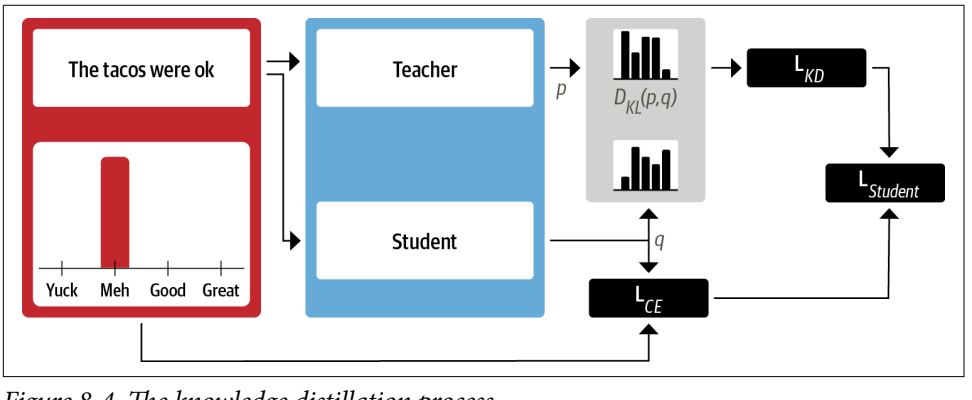


Figure 8-4. The knowledge distillation process

Knowledge Distillation for Pretraining

Knowledge distillation can also be used during pretraining to create a general-purpose student that can be subsequently fine-tuned on downstream tasks. In this case, the teacher is a pretrained language model like BERT, which transfers its knowledge about masked language modeling to the student. For example, in the DistilBERT paper,[8] the masked language modeling loss L_{mlm} is augmented with a term from knowledge distillation and a cosine embedding loss $L_{cos} = 1 - \cos\left(h_s, h_t\right)$ to align the directions of the hidden state vectors between the teacher and student:

$$L_{\text{DistilBERT}} = \alpha L_{mlm} + \beta L_{KD} + \gamma L_{cos}$$

Since we already have a fine-tuned BERT-base model, let's see how we can use knowledge distillation to fine-tune a smaller and faster model. To do that we'll need a way to augment the cross-entropy loss with an L_{KD} term. Fortunately we can do this by creating our own trainer!

Creating a Knowledge Distillation Trainer

To implement knowledge distillation we need to add a few things to the `Trainer` base class:

- The new hyperparameters α and T, which control the relative weight of the distillation loss and how much the probability distribution of the labels should be smoothed
- The fine-tuned teacher model, which in our case is BERT-base
- A new loss function that combines the cross-entropy loss with the knowledge distillation loss

Adding the new hyperparameters is quite simple, since we just need to subclass `TrainingArguments` and include them as new attributes:

```
from transformers import TrainingArguments

class DistillationTrainingArguments(TrainingArguments):
    def __init__(self, *args, alpha=0.5, temperature=2.0, **kwargs):
        super().__init__(*args, **kwargs)
        self.alpha = alpha
        self.temperature = temperature
```

8 V. Sanh et al., "DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter", (2019).

For the trainer itself, we need a new loss function. The way to implement this is by subclassing `Trainer` and overriding the `compute_loss()` method to include the knowledge distillation loss term L_{KD}:

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

class DistillationTrainer(Trainer):
    def __init__(self, *args, teacher_model=None, **kwargs):
        super().__init__(*args, **kwargs)
        self.teacher_model = teacher_model

    def compute_loss(self, model, inputs, return_outputs=False):
        outputs_stu = model(**inputs)
        # Extract cross-entropy loss and logits from student
        loss_ce = outputs_stu.loss
        logits_stu = outputs_stu.logits
        # Extract logits from teacher
        with torch.no_grad():
            outputs_tea = self.teacher_model(**inputs)
            logits_tea = outputs_tea.logits
        # Soften probabilities and compute distillation loss
        loss_fct = nn.KLDivLoss(reduction="batchmean")
        loss_kd = self.args.temperature ** 2 * loss_fct(
            F.log_softmax(logits_stu / self.args.temperature, dim=-1),
            F.softmax(logits_tea / self.args.temperature, dim=-1))
        # Return weighted student loss
        loss = self.args.alpha * loss_ce + (1. - self.args.alpha) * loss_kd
        return (loss, outputs_stu) if return_outputs else loss
```

Let's unpack this code a bit. When we instantiate `DistillationTrainer` we pass a `teacher_model` argument with a teacher that has already been fine-tuned on our task. Next, in the `compute_loss()` method we extract the logits from the student and teacher, scale them by the temperature, and then normalize them with a softmax before passing them to PyTorch's `nn.KLDivLoss()` function for computing the KL divergence. One quirk with `nn.KLDivLoss()` is that it expects the inputs in the form of log probabilities and the labels as normal probabilities. That's why we've used the `F.log_softmax()` function to normalize the student's logits, while the teacher's logits are converted to probabilities with a standard softmax. The `reduction=batchmean` argument in `nn.KLDivLoss()` specifies that we average the losses over the batch dimension.

You can also perform knowledge distillation with the Keras API of the 🤗 Transformers library. To do this, you'll need to implement a custom `Distiller` class that overrides the `train_step()`, `test_step()`, and `compile()` methods of `tf.keras.Model()`. See the Keras documentation for an example of how to do this.

Choosing a Good Student Initialization

Now that we have our custom trainer, the first question you might have is which pretrained language model should we pick for the student? In general we should pick a smaller model for the student to reduce the latency and memory footprint. A good rule of thumb from the literature is that knowledge distillation works best when the teacher and student are of the same *model type*.[9] One possible reason for this is that different model types, say BERT and RoBERTa, can have different output embedding spaces, which hinders the ability of the student to mimic the teacher. In our case study the teacher is BERT, so DistilBERT is a natural candidate to initialize the student with since it has 40% fewer parameters and has been shown to achieve strong results on downstream tasks.

First we'll need to tokenize and encode our queries, so let's instantiate the tokenizer from DistilBERT and create a simple `tokenize_text()` function to take care of the preprocessing:

```
from transformers import AutoTokenizer

student_ckpt = "distilbert-base-uncased"
student_tokenizer = AutoTokenizer.from_pretrained(student_ckpt)

def tokenize_text(batch):
    return student_tokenizer(batch["text"], truncation=True)

clinc_enc = clinc.map(tokenize_text, batched=True, remove_columns=["text"])
clinc_enc = clinc_enc.rename_column("intent", "labels")
```

Here we've removed the `text` column since we no longer need it, and we've also renamed the `intent` column to `labels` so it can be automatically detected by the trainer.[10]

Now that we've processed our texts, the next thing we need to do is define the hyperparameters and `compute_metrics()` function for our `DistillationTrainer`. We'll also push all of our models to the Hugging Face Hub, so let's start by logging in to our account:

```
from huggingface_hub import notebook_login

notebook_login()
```

9 Y. Kim and H. Awadalla, "FastFormers: Highly Efficient Transformer Models for Natural Language Understanding", (2020).

10 By default, the `Trainer` looks for a column called `labels` when fine-tuning on classification tasks. You can also override this behavior by specifying the `label_names` argument of `TrainingArguments`.

Next, we'll define the metrics to track during training. As we did in the performance benchmark, we'll use accuracy as the main metric. This means we can reuse our `accuracy_score()` function in the `compute_metrics()` function that we'll include in `DistillationTrainer`:

```
def compute_metrics(pred):
    predictions, labels = pred
    predictions = np.argmax(predictions, axis=1)
    return accuracy_score.compute(predictions=predictions, references=labels)
```

In this function, the predictions from the sequence modeling head come in the form of logits, so we use the `np.argmax()` function to find the most confident class prediction and compare that against the ground truth label.

Next we need to define the training arguments. To warm up, we'll set $\alpha = 1$ to see how well DistilBERT performs without any signal from the teacher.[11] Then we will push our fine-tuned model to a new repository called `distilbert-base-uncased-finetuned-clinc`, so we just need to specify that in the `output_dir` argument of `DistillationTrainingArguments`:

```
batch_size = 48

finetuned_ckpt = "distilbert-base-uncased-finetuned-clinc"
student_training_args = DistillationTrainingArguments(
    output_dir=finetuned_ckpt, evaluation_strategy = "epoch",
    num_train_epochs=5, learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size, alpha=1, weight_decay=0.01,
    push_to_hub=True)
```

We've also tweaked a few of the default hyperparameter values, like the number of epochs, the weight decay, and the learning rate. The next thing to do is initialize a student model. Since we will be doing multiple runs with the trainer, we'll create a `student_init()` function to initialize the model with each new run. When we pass this function to the `DistillationTrainer`, this will ensure we initialize a new model each time we call the `train()` method.

One other thing we need to do is provide the student model with the mappings between each intent and label ID. These mappings can be obtained from our BERT-base model that we downloaded in the pipeline:

```
id2label = pipe.model.config.id2label
label2id = pipe.model.config.label2id
```

11 This approach of fine-tuning a general-purpose, distilled language model is sometimes referred to as "task-agnostic" distillation.

With these mappings, we can now create a custom model configuration with the `AutoConfig` class hat we encountered in Chapters 3 and 4. Let's use this to create a configuration for our student with the information about the label mappings:

```
from transformers import AutoConfig

num_labels = intents.num_classes
student_config = (AutoConfig
                  .from_pretrained(student_ckpt, num_labels=num_labels,
                                   id2label=id2label, label2id=label2id))
```

Here we've also specified the number of classes our model should expect. We can then provide this configuration to the `from_pretrained()` function of the `AutoModelForSequenceClassification` class as follows:

```
import torch
from transformers import AutoModelForSequenceClassification

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def student_init():
    return (AutoModelForSequenceClassification
            .from_pretrained(student_ckpt, config=student_config).to(device))
```

We now have all the ingredients needed for our distillation trainer, so let's load the teacher and fine-tune:

```
teacher_ckpt = "transformersbook/bert-base-uncased-finetuned-clinc"
teacher_model = (AutoModelForSequenceClassification
                 .from_pretrained(teacher_ckpt, num_labels=num_labels)
                 .to(device))

distilbert_trainer = DistillationTrainer(model_init=student_init,
    teacher_model=teacher_model, args=student_training_args,
    train_dataset=clinc_enc['train'], eval_dataset=clinc_enc['validation'],
    compute_metrics=compute_metrics, tokenizer=student_tokenizer)

distilbert_trainer.train()
```

Epoch	Training Loss	Validation Loss	Accuracy
1	4.2923	3.289337	0.742258
2	2.6307	1.883680	0.828065
3	1.5483	1.158315	0.896774
4	1.0153	0.861815	0.909355
5	0.7958	0.777289	0.917419

The 92% accuracy on the validation set looks quite good compared to the 94% that the BERT-base teacher achieves. Now that we've fine-tuned DistilBERT, let's push the model to the Hub so we can reuse it later:

```
distilbert_trainer.push_to_hub("Training completed!")
```

With our model now safely stored on the Hub, we can immediately use it in a pipeline for our performance benchmark:

```
finetuned_ckpt = "transformersbook/distilbert-base-uncased-finetuned-clinc"
pipe = pipeline("text-classification", model=finetuned_ckpt)
```

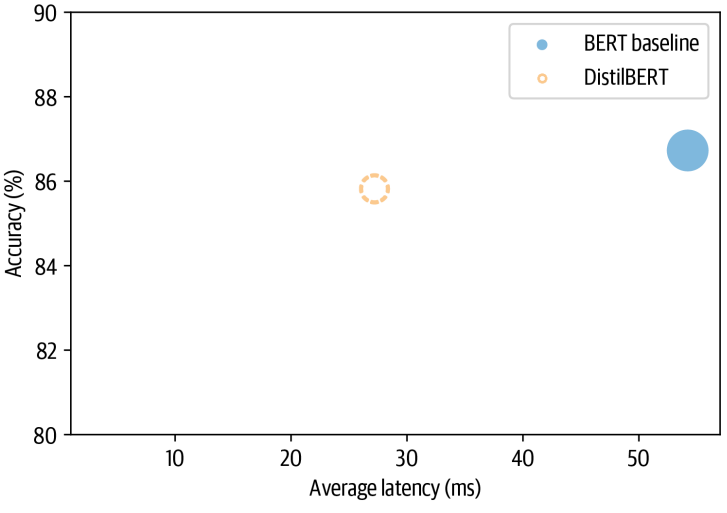
We can then pass this pipeline to our `PerformanceBenchmark` class to compute the metrics associated with this model:

```
optim_type = "DistilBERT"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
```

```
Model size (MB) - 255.89
Average latency (ms) - 27.53 +\- 0.60
Accuracy on test set - 0.858
```

To compare these results against our baseline, let's create a scatter plot of the accuracy against the latency, with the radius of each point corresponding to the size of the model on disk. The following function does what we need and marks the current optimization type as a dashed circle to aid the comparison to previous results:

```
import pandas as pd

def plot_metrics(perf_metrics, current_optim_type):
    df = pd.DataFrame.from_dict(perf_metrics, orient='index')

    for idx in df.index:
        df_opt = df.loc[idx]
        # Add a dashed circle around the current optimization type
        if idx == current_optim_type:
            plt.scatter(df_opt["time_avg_ms"], df_opt["accuracy"] * 100,
                        alpha=0.5, s=df_opt["size_mb"], label=idx,
                        marker='$\u25CC$')
        else:
            plt.scatter(df_opt["time_avg_ms"], df_opt["accuracy"] * 100,
                        s=df_opt["size_mb"], label=idx, alpha=0.5)

    legend = plt.legend(bbox_to_anchor=(1,1))
    for handle in legend.legendHandles:
        handle.set_sizes([20])

    plt.ylim(80,90)
    # Use the slowest model to define the x-axis range
    xlim = int(perf_metrics["BERT baseline"]["time_avg_ms"] + 3)
    plt.xlim(1, xlim)
    plt.ylabel("Accuracy (%)")
    plt.xlabel("Average latency (ms)")
    plt.show()

plot_metrics(perf_metrics, optim_type)
```



From the plot we can see that by using a smaller model we've managed to significantly decrease the average latency. And all this at the price of just over a 1% reduction in accuracy! Let's see if we can close that last gap by including the distillation loss of the teacher and finding good values for α and T.

Finding Good Hyperparameters with Optuna

To find good values for α and T, we could do a grid search over the 2D parameter space. But a much better alternative is to use *Optuna*,[12] which is an optimization framework designed for just this type of task. Optuna formulates the search problem in terms of an objective function that is optimized through multiple *trials*. For example, suppose we wished to minimize Rosenbrock's "banana function":

$$f(x, y) = (1 - x)^2 + 100\left(y - x^2\right)^2$$

which is a famous test case for optimization frameworks. As shown in Figure 8-5, the function gets its name from the curved contours and has a global minimum at $(x, y) = (1, 1)$. Finding the valley is an easy optimization problem, but converging to the global minimum is not.

12 T. Akiba et al., "Optuna: A Next-Generation Hyperparameter Optimization Framework", (2019).

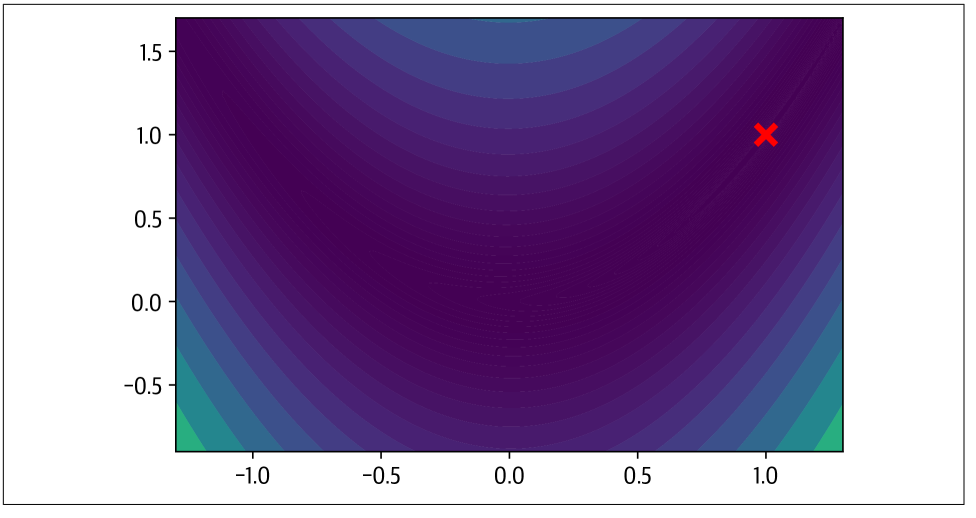


Figure 8-5. Plot of the Rosenbrock function of two variables

In Optuna, we can find the minimum of $f(x, y)$ by defining an `objective()` function that returns the value of $f(x, y)$:

```
def objective(trial):
    x = trial.suggest_float("x", -2, 2)
    y = trial.suggest_float("y", -2, 2)
    return (1 - x) ** 2 + 100 * (y - x ** 2) ** 2
```

The `trial.suggest_float` object specifies the parameter ranges to sample uniformly from; Optuna also provides `suggest_int` and `suggest_categorical` for integer and categorical parameters, respectively. Optuna collects multiple trials as a *study*, so to create one we just pass the `objective()` function to `study.optimize()` as follows:

```
import optuna

study = optuna.create_study()
study.optimize(objective, n_trials=1000)
```

Once the study is completed, we can then find the best parameters as follows:

```
study.best_params
```

```
{'x': 1.003024865971437, 'y': 1.00315167589307}
```

We see that with one thousand trials, Optuna has managed to find values for x and y that are reasonably close to the global minimum. To use Optuna in 🤗 Transformers, we use similar logic by first defining the hyperparameter space that we wish to optimize over. In addition to α and T, we'll include the number of training epochs as follows:

```
def hp_space(trial):
    return {"num_train_epochs": trial.suggest_int("num_train_epochs", 5, 10),
        "alpha": trial.suggest_float("alpha", 0, 1),
        "temperature": trial.suggest_int("temperature", 2, 20)}
```

Running the hyperparameter search with the `Trainer` is then quite simple; we just need to specify the number of trials to run and a direction to optimize for. Because we want the best possible accuracy, we specify `direction="maximize"` in the `hyperparameter_search()` method of the trainer and pass the hyperparameter search space as follows:

```
best_run = distilbert_trainer.hyperparameter_search(
    n_trials=20, direction="maximize", hp_space=hp_space)
```

The `hyperparameter_search()` method returns a `BestRun` object, which contains the value of the objective that was maximized (by default, the sum of all metrics) and the hyperparameters it used for that run:

```
print(best_run)
```

```
BestRun(run_id='1', objective=0.927741935483871,
hyperparameters={'num_train_epochs': 10, 'alpha': 0.12468168730193585,
'temperature': 7})
```

This value of α tells us that most of the training signal is coming from the knowledge distillation term. Let's update our training arguments with these values and run the final training run:

```
for k,v in best_run.hyperparameters.items():
    setattr(student_training_args, k, v)

# Define a new repository to store our distilled model
distilled_ckpt = "distilbert-base-uncased-distilled-clinc"
student_training_args.output_dir = distilled_ckpt

# Create a new Trainer with optimal parameters
distil_trainer = DistillationTrainer(model_init=student_init,
    teacher_model=teacher_model, args=student_training_args,
    train_dataset=clinc_enc['train'], eval_dataset=clinc_enc['validation'],
    compute_metrics=compute_metrics, tokenizer=student_tokenizer)

distil_trainer.train();
```

Epoch	Training Loss	Validation Loss	Accuracy
1	0.9031	0.574540	0.736452
2	0.4481	0.285621	0.874839
3	0.2528	0.179766	0.918710
4	0.1760	0.139828	0.929355
5	0.1416	0.121053	0.934839
6	0.1243	0.111640	0.934839
7	0.1133	0.106174	0.937742
8	0.1075	0.103526	0.938710
9	0.1039	0.101432	0.938065
10	0.1018	0.100493	0.939355

Remarkably, we've been able to train the student to match the accuracy of the teacher, despite it having almost half the number of parameters! Let's push the model to the Hub for future use:

```
distil_trainer.push_to_hub("Training complete")
```

Benchmarking Our Distilled Model

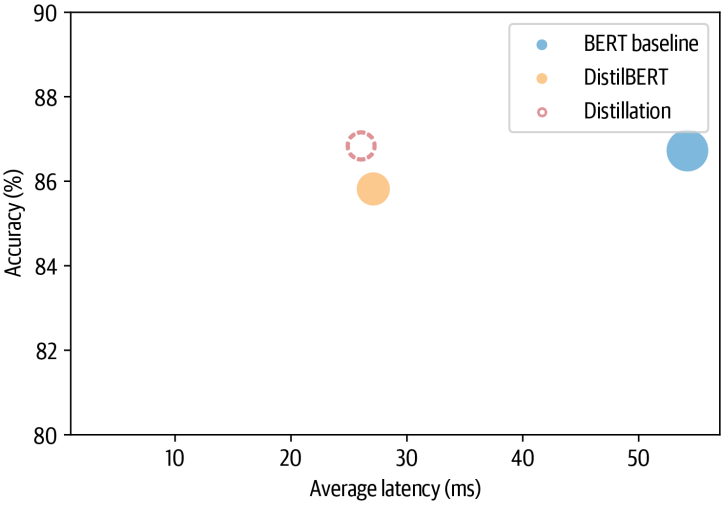
Now that we have an accurate student, let's create a pipeline and redo our benchmark to see how we perform on the test set:

```
distilled_ckpt = "transformersbook/distilbert-base-uncased-distilled-clinc"
pipe = pipeline("text-classification", model=distilled_ckpt)
optim_type = "Distillation"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
```

```
Model size (MB) - 255.89
Average latency (ms) - 25.96 +\- 1.63
Accuracy on test set - 0.868
```

To put these results in context, let's also visualize them with our `plot_metrics()` function:

```
plot_metrics(perf_metrics, optim_type)
```



As expected, the model size and latency remain essentially unchanged compared to the DistilBERT benchmark, but the accuracy has improved and even surpassed the performance of the teacher! One way to interpret this surprising result is that the teacher has likely not been fine-tuned as systematically as the student. This is great, but we can actually compress our distilled model even further using a technique known as quantization. That's the topic of the next section.

Making Models Faster with Quantization

We've now seen that with knowledge distillation we can reduce the computational and memory cost of running inference by transferring the information from a teacher into a smaller student. Quantization takes a different approach; instead of reducing the number of computations, it makes them much more efficient by representing the weights and activations with low-precision data types like 8-bit integer (INT8) instead of the usual 32-bit floating point (FP32). Reducing the number of bits means the resulting model requires less memory storage, and operations like matrix multiplication can be performed much faster with integer arithmetic. Remarkably, these performance gains can be realized with little to no loss in accuracy!

A Primer on Floating-Point and Fixed-Point Numbers

Most transformers today are pretrained and fine-tuned with floating-point numbers (usually FP32 or a mix of FP16 and FP32), since they provide the precision needed to accommodate the very different ranges of weights, activations, and gradients. A floating-point number like FP32 represents a sequence of 32 bits that are grouped in terms of a *sign*, *exponent*, and *significand*. The sign determines whether the number is positive or negative, while the significand corresponds to the number of significant digits, which are scaled using the exponent in some fixed base (usually 2 for binary or 10 for decimal).

For example, the number 137.035 can be expressed as a decimal floating-point number through the following arithmetic:

$$137.035 = (-1)^0 \times 1.37035 \times 10^2$$

where the 1.37035 is the significand and 2 is the exponent of the base 10. Through the exponent we can represent a wide range of real numbers, and the decimal or binary point can be placed anywhere relative to the significant digits (hence the name "floating-point").

However, once a model is trained, we only need the forward pass to run inference, so we can reduce the precision of the data types without impacting the accuracy too much. For neural networks it is common to use a *fixed-point format* for the low-precision data types, where real numbers are represented as B-bit integers that are scaled by a common factor for all variables of the same type. For example, 137.035 can be represented as the integer 137,035 that is scaled by 1/1,000. We can control the range and precision of a fixed-point number by adjusting the scaling factor.

The basic idea behind quantization is that we can "discretize" the floating-point values f in each tensor by mapping their range $[f_{\text{max}}, f_{\text{min}}]$ into a smaller one $[q_{\text{max}}, q_{\text{min}}]$ of fixed-point numbers q, and linearly distributing all values in between. Mathematically, this mapping is described by the following equation:

$$f = \left(\frac{f_{\text{max}} - f_{\text{min}}}{q_{\text{max}} - q_{\text{min}}}\right)(q - Z) = S(q - Z)$$

where the scale factor S is a positive floating-point number and the constant Z has the same type as q and is called the *zero point* because it corresponds to the quantized value of the floating-point value $f = 0$. Note that the map needs to be *affine* so that we

get back floating-point numbers when we dequantize the fixed-point ones.[13] An illustration of the conversion is shown in Figure 8-6.

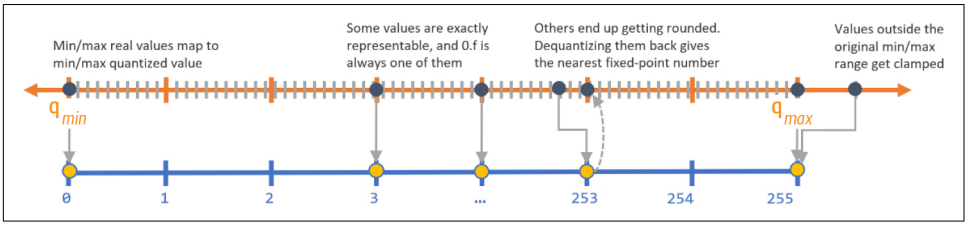
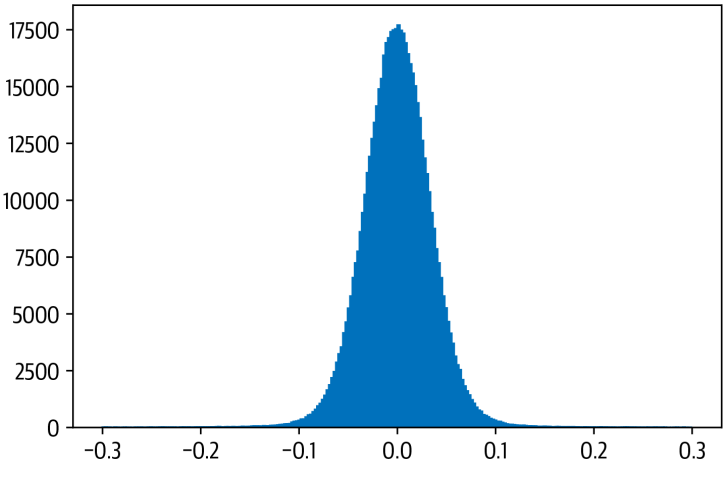


Figure 8-6. Quantizing floating-point numbers as unsigned 8-bit integers (courtesy of Manas Sahni)

Now, one of the main reasons why transformers (and deep neural networks more generally) are prime candidates for quantization is that the weights and activations tend to take values in relatively small ranges. This means we don't have to squeeze the whole range of possible FP32 numbers into, say, the $2^8 = 256$ numbers represented by INT8. To see this, let's pick out one of the attention weight matrices from our distilled model and plot the frequency distribution of the values:

```
import matplotlib.pyplot as plt

state_dict = pipe.model.state_dict()
weights = state_dict["distilbert.transformer.layer.0.attention.out_lin.weight"]
plt.hist(weights.flatten().numpy(), bins=250, range=(-0.3,0.3), edgecolor="C0")
plt.show()
```



13 An affine map is just a fancy name for the $y = Ax + b$ map that you're familiar with in the linear layers of a neural network.

As we can see, the values of the weights are distributed in the small range $[-0.1, 0.1]$ around zero. Now, suppose we want to quantize this tensor as a signed 8-bit integer. In that case, the range of possible values for our integers is $[q_{\text{max}}, q_{\text{min}}] = [-128, 127]$. The zero point coincides with the zero of FP32 and the scale factor is calculated according to the previous equation:

```
zero_point = 0
scale = (weights.max() - weights.min()) / (127 - (-128))
```

To obtain the quantized tensor, we just need to invert the mapping $q = f/S + Z$, clamp the values, round them to the nearest integer, and represent the result in the `torch.int8` data type using the `Tensor.char()` function:

```
(weights / scale + zero_point).clamp(-128, 127).round().char()
```

```
tensor([[ -5,  -8,   0,  ...,  -6,  -4,   8],
        [  8,   3,   1,  ...,  -4,   7,   0],
        [ -9,  -6,   5,  ...,   1,   5,  -3],
        ...,
        [  6,   0,  12,  ...,   0,   6,  -1],
        [  0,  -2, -12,  ...,  12,  -7, -13],
        [-13,  -1, -10,  ...,   8,   2,  -2]], dtype=torch.int8)
```

Great, we've just quantized our first tensor! In PyTorch we can simplify the conversion by using the `quantize_per_tensor()` function together with a quantized data type, `torch.qint`, that is optimized for integer arithmetic operations:

```
from torch import quantize_per_tensor

dtype = torch.qint8
quantized_weights = quantize_per_tensor(weights, scale, zero_point, dtype)
quantized_weights.int_repr()
```

```
tensor([[ -5,  -8,   0,  ...,  -6,  -4,   8],
        [  8,   3,   1,  ...,  -4,   7,   0],
        [ -9,  -6,   5,  ...,   1,   5,  -3],
        ...,
        [  6,   0,  12,  ...,   0,   6,  -1],
        [  0,  -2, -12,  ...,  12,  -7, -13],
        [-13,  -1, -10,  ...,   8,   2,  -2]], dtype=torch.int8)
```

The plot in Figure 8-7 shows very clearly the discretization that's induced by only mapping some of the weight values precisely and rounding the rest.

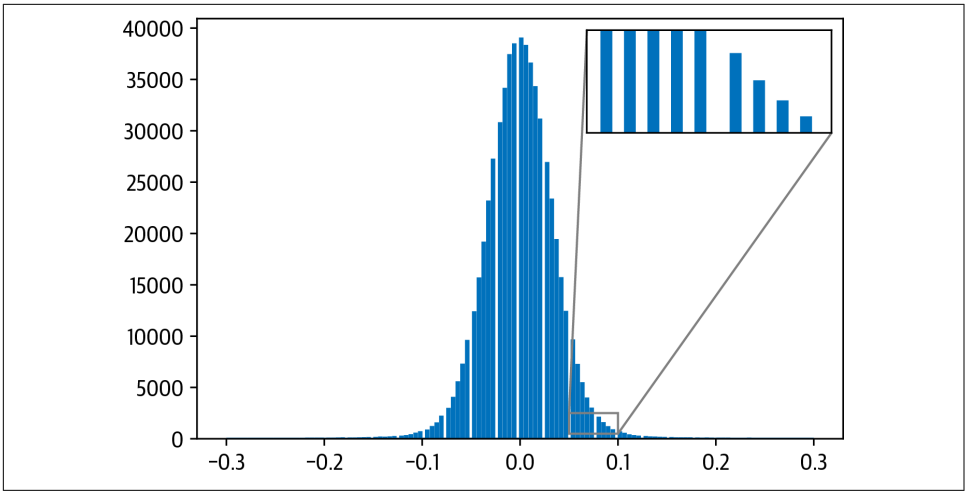


Figure 8-7. Effect of quantization on a transformer's weights

To round out our little analysis, let's compare how long it takes to compute the multiplication of two weight tensors with FP32 and INT8 values. For the FP32 tensors, we can multiply them using PyTorch's nifty `@` operator:

```
%%timeit
weights @ weights
```

```
393 µs ± 3.84 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

For the quantized tensors we need the `QFunctional` wrapper class so that we can perform operations with the special `torch.qint8` data type:

```
from torch.nn.quantized import QFunctional

q_fn = QFunctional()
```

This class supports various elementary operations, like addition, and in our case we can time the multiplication of our quantized tensors as follows:

```
%%timeit
q_fn.mul(quantized_weights, quantized_weights)
```

```
23.3 µs ± 298 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Compared to our FP32 computation, using the INT8 tensors is almost 100 times faster! Even larger gains can be obtained by using dedicated backends for running quantized operators efficiently. As of this book's writing, PyTorch supports:

- x86 CPUs with AVX2 support or higher
- ARM CPUs (typically found in mobile/embedded devices)

Since INT8 numbers have four times fewer bits than FP32 numbers, quantization also reduces the memory storage requirements by up to a factor of four. In our simple example we can verify this by comparing the underlying storage size of our weight tensor and its quantized cousin by using the `Tensor.storage()` function and the `getsizeof()` function from Python's `sys` module:

```
import sys

sys.getsizeof(weights.storage()) / sys.getsizeof(quantized_weights.storage())
```

```
3.999633833760527
```

For a full-scale transformer, the actual compression rate depends on which layers are quantized (as we'll see in the next section it is only the linear layers that typically get quantized).

So what's the catch with quantization? Changing the precision for all computations in our model introduces small disturbances at each point in the model's computational graph, which can compound and affect the model's performance. There are several ways to quantize a model, which all have pros and cons. For deep neural networks, there are typically three main approaches to quantization:

Dynamic quantization
: When using dynamic quantization nothing is changed during training and the adaptations are only performed during inference. Like with all the quantization methods we will discuss, the weights of the model are converted to INT8 ahead of inference time. In addition to the weights, the model's activations are also quantized. This approach is dynamic because the quantization happens on the fly. This means that all the matrix multiplications can be calculated with highly optimized INT8 functions. Of all the quantization methods discussed here, dynamic quantization is the simplest one. However, with dynamic quantization the activations are written and read to memory in floating-point format. This conversion between integer and floating point can be a performance bottleneck.

Static quantization
: Instead of computing the quantization of the activations on the fly, we can avoid the conversion to floating point by precomputing the quantization scheme. Static quantization achieves this by observing the activation patterns on a representative sample of the data ahead of inference time. The ideal quantization scheme is calculated and then saved. This enables us to skip the conversion between INT8 and FP32 values and speeds up the computations. However, it requires access to a good data sample and introduces an additional step in the pipeline, since we now need to train and determine the quantization scheme before we can perform inference. There is also one aspect that static quantization does not address: the discrepancy between the precision during training and inference, which leads to a performance drop in the model's metrics (e.g., accuracy).

Quantization-aware training
: The effect of quantization can be effectively simulated during training by "fake" quantization of the FP32 values. Instead of using INT8 values during training, the FP32 values are rounded to mimic the effect of quantization. This is done during both the forward and the backward pass and improves performance in terms of model metrics over static and dynamic quantization.

The main bottleneck for running inference with transformers is the compute and memory bandwidth associated with the enormous numbers of weights in these models. For this reason, dynamic quantization is currently the best approach for transformer-based models in NLP. In smaller computer vision models the limiting factor is the memory bandwidth of the activations, which is why static quantization is generally used (or quantization-aware training in cases where the performance drops are too significant).

Implementing dynamic quantization in PyTorch is quite simple and can be done with a single line of code:

```
from torch.quantization import quantize_dynamic

model_ckpt = "transformersbook/distilbert-base-uncased-distilled-clinc"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = (AutoModelForSequenceClassification
         .from_pretrained(model_ckpt).to("cpu"))

model_quantized = quantize_dynamic(model, {nn.Linear}, dtype=torch.qint8)
```

Here we pass to `quantize_dynamic()` the full-precision model and specify the set of PyTorch layer classes in that model that we want to quantize. The `dtype` argument specifies the target precision and can be `fp16` or `qint8`. A good practice is to pick the lowest precision that you can tolerate with respect to your evaluation metrics. In this chapter we'll use INT8, which as we'll soon see has little impact on our model's accuracy.

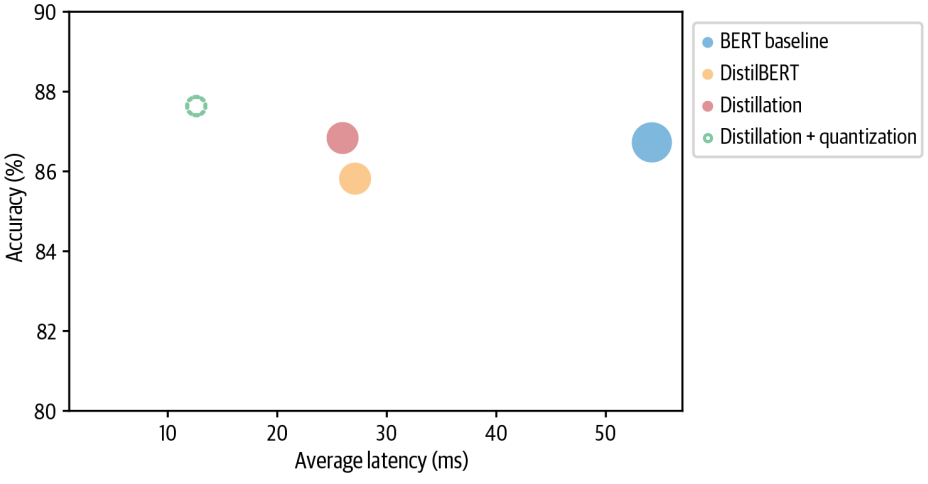
Benchmarking Our Quantized Model

With our model now quantized, let's pass it through the benchmark and visualize the results:

```
pipe = pipeline("text-classification", model=model_quantized,
                tokenizer=tokenizer)
optim_type = "Distillation + quantization"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
```

```
Model size (MB) - 132.40
Average latency (ms) - 12.54 +\- 0.73
Accuracy on test set - 0.876
```

```
plot_metrics(perf_metrics, optim_type)
```



Nice, the quantized model is almost half the size of our distilled one and has even gained a slight accuracy boost! Let's see if we can push our optimization to the limit with a powerful framework called the ONNX Runtime.

Optimizing Inference with ONNX and the ONNX Runtime

ONNX is an open standard that defines a common set of operators and a common file format to represent deep learning models in a wide variety of frameworks, including PyTorch and TensorFlow.[14] When a model is exported to the ONNX format, these operators are used to construct a computational graph (often called an *intermediate representation*) that represents the flow of data through the neural network. An example of such a graph for BERT-base is shown in Figure 8-8, where each node receives some input, applies an operation like `Add` or `Squeeze`, and then feeds the output to the next set of nodes.

14 There is a separate standard called ONNX-ML that is designed for traditional machine learning models like random forests and frameworks like Scikit-learn.

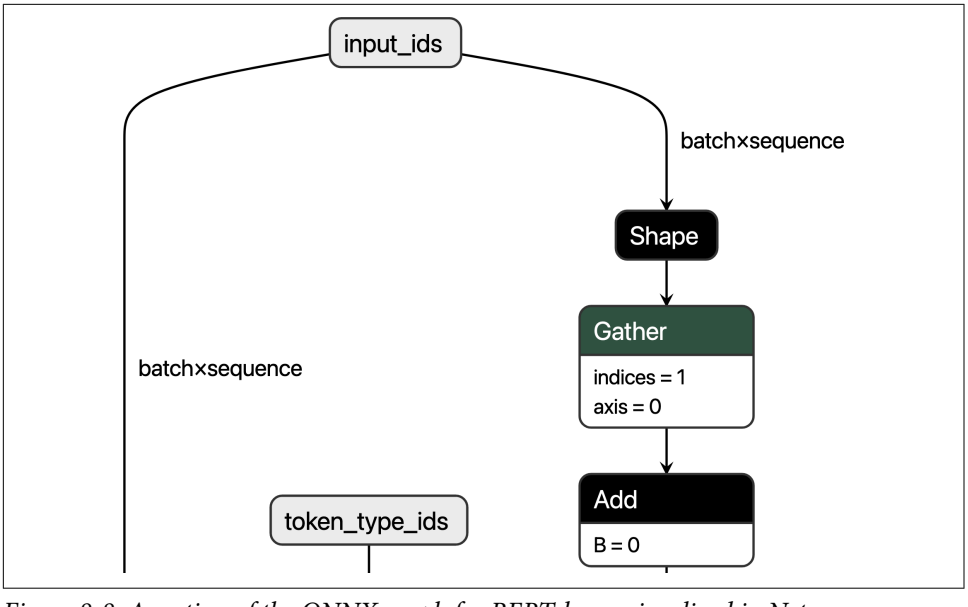


Figure 8-8. A section of the ONNX graph for BERT-base, visualized in Netron

By exposing a graph with standardized operators and data types, ONNX makes it easy to switch between frameworks. For example, a model trained in PyTorch can be exported to ONNX format and then imported in TensorFlow (and vice versa).

Where ONNX really shines is when it is coupled with a dedicated accelerator like ONNX Runtime, or ORT for short.[15] ORT provides tools to optimize the ONNX graph through techniques like operator fusion and constant folding,[16] and defines an interface to *execution providers* that allow you to run the model on different types of hardware. This is a powerful abstraction. Figure 8-9 shows the high-level architecture of the ONNX and ORT ecosystem.

15 Other popular accelerators include NVIDIA's TensorRT and Apache TVM.

16 A fused operation involves merging one operator (usually an activation function) into another so that they can be executed together. For example, suppose we want to apply an activation f to a matrix product $A \times B$. Normally the result of the product needs to be written back to the GPU memory before the activation is computed. Operator fusion allows as to compute $f(A \times B)$ in a single step. Constant folding refers to the process of evaluating constant expressions at compile time instead of runtime.

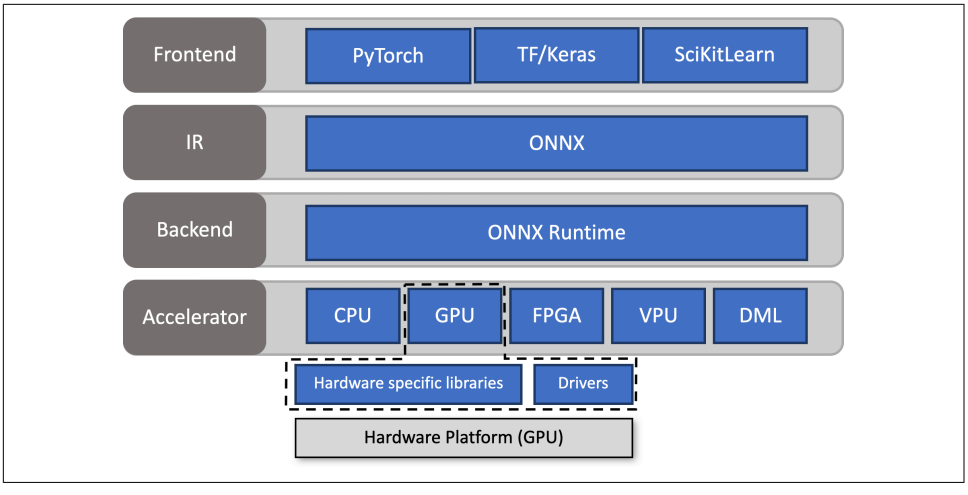


Figure 8-9. Architecture of the ONNX and ONNX Runtime ecosystem (courtesy of the ONNX Runtime team)

To see ORT in action, the first thing we need to do is convert our distilled model into the ONNX format. The 🤗 Transformers library has a built-in function called `convert_graph_to_onnx.convert()` that simplifies the process by taking the following steps:

1. Initialize the model as a `Pipeline`.
2. Run placeholder inputs through the pipeline so that ONNX can record the computational graph.
3. Define dynamic axes to handle dynamic sequence lengths.
4. Save the graph with network parameters.

To use this function, we first need to set some OpenMP environment variables for ONNX:

```
import os
from psutil import cpu_count

os.environ["OMP_NUM_THREADS"] = f"{cpu_count()}"
os.environ["OMP_WAIT_POLICY"] = "ACTIVE"
```

OpenMP is an API designed for developing highly parallelized applications. The `OMP_NUM_THREADS` environment variable sets the number of threads to use for parallel computations in the ONNX Runtime, while `OMP_WAIT_POLICY=ACTIVE` specifies that waiting threads should be active (i.e., using CPU processor cycles).

Next, let's convert our distilled model to the ONNX format. Here we need to specify the argument `pipeline_name="text-classification"` since `convert()` wraps the

model in a 🤗 Transformers `pipeline()` function during the conversion. In addition to the `model_ckpt`, we also pass the tokenizer to initialize the pipeline:

```
from transformers.convert_graph_to_onnx import convert

model_ckpt = "transformersbook/distilbert-base-uncased-distilled-clinc"
onnx_model_path = Path("onnx/model.onnx")
convert(framework="pt", model=model_ckpt, tokenizer=tokenizer,
        output=onnx_model_path, opset=12, pipeline_name="text-classification")
```

ONNX uses *operator sets* to group together immutable operator specifications, so `opset=12` corresponds to a specific version of the ONNX library.

Now that we have our model saved, we need to create an `InferenceSession` instance to feed inputs to the model:

```
from onnxruntime import (GraphOptimizationLevel, InferenceSession,
                         SessionOptions)

def create_model_for_provider(model_path, provider="CPUExecutionProvider"):
    options = SessionOptions()
    options.intra_op_num_threads = 1
    options.graph_optimization_level = GraphOptimizationLevel.ORT_ENABLE_ALL
    session = InferenceSession(str(model_path), options, providers=[provider])
    session.disable_fallback()
    return session

onnx_model = create_model_for_provider(onnx_model_path)
```

Now when we call `onnx_model.run()`, we can get the class logits from the ONNX model. Let's test this out with an example from the test set. Since the output from `convert()` tells us that ONNX expects just the `input_ids` and `attention_mask` as inputs, we need to drop the `label` column from our sample:

```
inputs = clinc_enc["test"][:1]
del inputs["labels"]
logits_onnx = onnx_model.run(None, inputs)[0]
logits_onnx.shape
```

```
(1, 151)
```

Once we have the logits, we can easily get the predicted label by taking the argmax:

```
np.argmax(logits_onnx)
```

```
61
```

which indeed agrees with the ground truth label:

```
clinc_enc["test"][0]["labels"]
```

```
61
```

The ONNX model is not compatible with the `text-classification` pipeline, so we'll create our own class that mimics the core behavior:

```
from scipy.special import softmax

class OnnxPipeline:
    def __init__(self, model, tokenizer):
        self.model = model
        self.tokenizer = tokenizer

    def __call__(self, query):
        model_inputs = self.tokenizer(query, return_tensors="pt")
        inputs_onnx = {k: v.cpu().detach().numpy()
                       for k, v in model_inputs.items()}
        logits = self.model.run(None, inputs_onnx)[0][0, :]
        probs = softmax(logits)
        pred_idx = np.argmax(probs).item()
        return [{"label": intents.int2str(pred_idx), "score": probs[pred_idx]}]
```

We can then test this on our simple query to see if we recover the `car_rental` intent:

```
pipe = OnnxPipeline(onnx_model, tokenizer)
pipe(query)
```

```
[{'label': 'car_rental', 'score': 0.7848334}]
```

Great, our pipeline works as expected. The next step is to create a performance benchmark for ONNX models. Here we can build on the work we did with the `PerformanceBenchmark` class by simply overriding the `compute_size()` method and leaving the `compute_accuracy()` and `time_pipeline()` methods intact. The reason we need to override the `compute_size()` method is that we cannot rely on the `state_dict` and `torch.save()` to measure a model's size, since `onnx_model` is technically an ONNX `InferenceSession` object that doesn't have access to the attributes of PyTorch's `nn.Module`. In any case, the resulting logic is simple and can be implemented as follows:

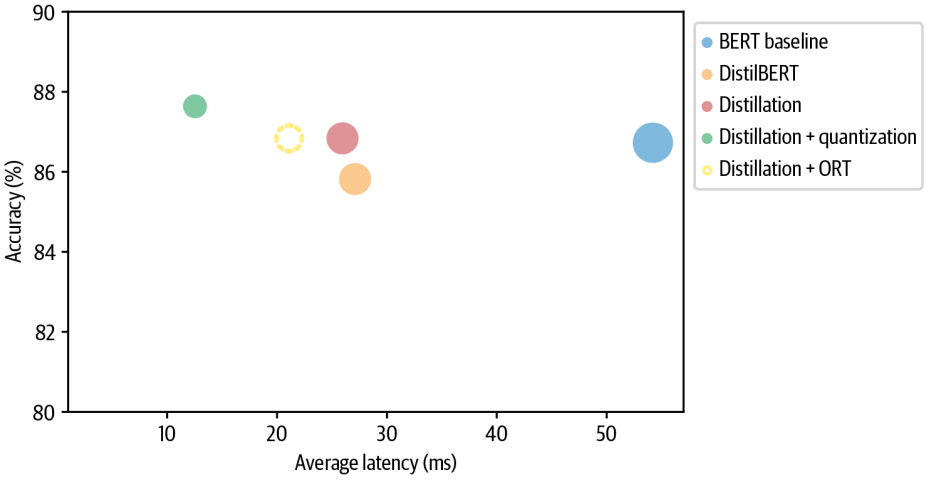
```
class OnnxPerformanceBenchmark(PerformanceBenchmark):
    def __init__(self, *args, model_path, **kwargs):
        super().__init__(*args, **kwargs)
        self.model_path = model_path

    def compute_size(self):
        size_mb = Path(self.model_path).stat().st_size / (1024 * 1024)
        print(f"Model size (MB) - {size_mb:.2f}")
        return {"size_mb": size_mb}
```

With our new benchmark, let's see how our distilled model performs when converted to ONNX format:

```
optim_type = "Distillation + ORT"
pb = OnnxPerformanceBenchmark(pipe, clinc["test"], optim_type,
                              model_path="onnx/model.onnx")
perf_metrics.update(pb.run_benchmark())
```

```
Model size (MB) - 255.88
Average latency (ms) - 21.02 +\- 0.55
Accuracy on test set - 0.868
```

```
plot_metrics(perf_metrics, optim_type)
```



Remarkably, converting to the ONNX format and using the ONNX Runtime has given our distilled model (i.e. the "Distillation" circle in the plot) a boost in latency! Let's see if we can squeeze out a bit more performance by adding quantization to the mix.

Similar to PyTorch, ORT offers three ways to quantize a model: dynamic, static, and quantization-aware training. As we did with PyTorch, we'll apply dynamic quantization to our distilled model. In ORT, the quantization is applied through the `quantize_dynamic()` function, which requires a path to the ONNX model to quantize, a target path to save the quantized model to, and the data type to reduce the weights to:

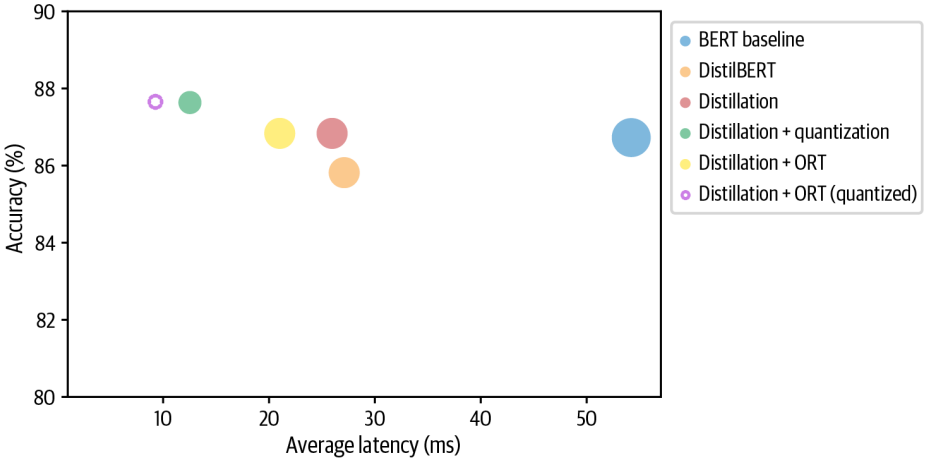
```
from onnxruntime.quantization import quantize_dynamic, QuantType

model_input = "onnx/model.onnx"
model_output = "onnx/model.quant.onnx"
quantize_dynamic(model_input, model_output, weight_type=QuantType.QInt8)
```

Now that the model is quantized, let's run it through our benchmark:

```
onnx_quantized_model = create_model_for_provider(model_output)
pipe = OnnxPipeline(onnx_quantized_model, tokenizer)
optim_type = "Distillation + ORT (quantized)"
pb = OnnxPerformanceBenchmark(pipe, clinc["test"], optim_type,
                              model_path=model_output)
perf_metrics.update(pb.run_benchmark())
```

```
Model size (MB) - 64.20
Average latency (ms) - 9.24 +\- 0.29
Accuracy on test set - 0.877
```

```
plot_metrics(perf_metrics, optim_type)
```



ORT quantization has reduced the model size and latency by around 30% compared to the model obtained from PyTorch quantization (the distillation + quantization blob). One reason for this is that PyTorch only optimizes the `nn.Linear` modules, while ONNX quantizes the embedding layer as well. From the plot we can also see that applying ORT quantization to our distilled model has provided an almost three-fold gain compared to our BERT baseline!

This concludes our analysis of techniques to speed up transformers for inference. We have seen that methods such as quantization reduce the model size by reducing the precision of the representation. Another strategy to reduce the size is to remove some weights altogether. This technique is called *weight pruning*, and it's the focus of the next section.

Making Models Sparser with Weight Pruning

So far we've seen that knowledge distillation and weight quantization are quite effective at producing faster models for inference, but in some cases you might also have strong constraints on the memory footprint of your model. For example, if our product manager suddenly decides that our text assistant needs to be deployed on a mobile device, then we'll need our intent classifier to take up as little storage space as possible. To round out our survey of compression methods, let's take a look at how we can shrink the number of parameters in our model by identifying and removing the least important weights in the network.

Sparsity in Deep Neural Networks

As shown in Figure 8-10, the main idea behind pruning is to gradually remove weight connections (and potentially neurons) during training such that the model becomes progressively sparser. The resulting pruned model has a smaller number of nonzero parameters, which can then be stored in a compact sparse matrix format. Pruning can be also combined with quantization to obtain further compression.

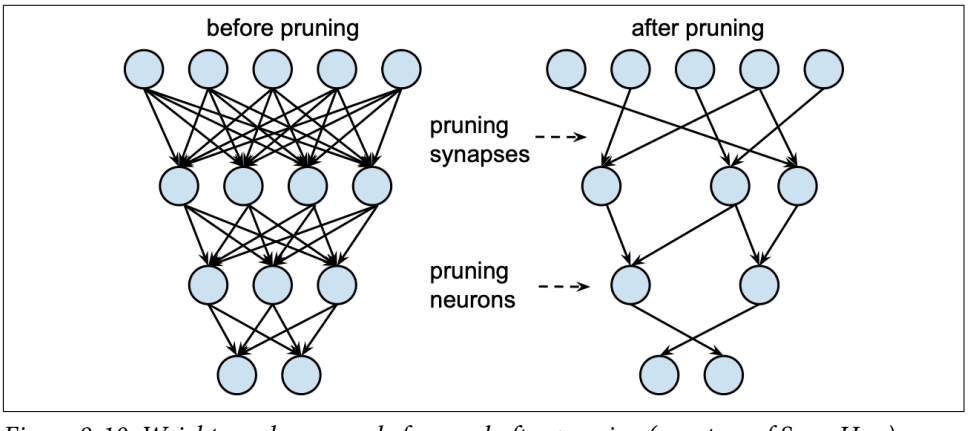


Figure 8-10. Weights and neurons before and after pruning (courtesy of Song Han)

Weight Pruning Methods

Mathematically, the way most weight pruning methods work is to calculate a matrix $\mathbf{S}$ of *importance scores* and then select the top k percent of weights by importance:

$$\text{Top}_k(\mathbf{S})_{ij} = \begin{cases} 1 & \text{if } S_{ij} \text{ in top } k\,\% \\ 0 & \text{otherwise} \end{cases}$$

In effect, k acts as a new hyperparameter to control the amount of sparsity in the model—that is, the proportion of weights that are zero-valued. Lower values of k correspond to sparser matrices. From these scores we can then define a *mask matrix* $\mathbf{M}$ that masks the weights W_{ij} during the forward pass with some input x_i and effectively creates a sparse network of activations a_i:

$$a_i = \sum_k W_{ik} M_{ik} x_k$$

As discussed in the tongue-in-cheek "Optimal Brain Surgeon" paper,[17] at the heart of each pruning method are a set of questions that need to be considered:

- Which weights should be eliminated?
- How should the remaining weights be adjusted for best performance?
- How can such network pruning be done in a computationally efficient way?

The answers to these questions inform how the score matrix $\mathbf{S}$ is computed, so let's begin by looking at one of the earliest and most popular pruning methods: magnitude pruning.

Magnitude pruning

As the name suggests, magnitude pruning calculates the scores according to the magnitude of the weights $\mathbf{S} = \left(\mid W_{ij} \mid \right)_{1 \le j, j \le n}$ and then derives the masks from $\mathbf{M} = \text{Top}_k(\mathbf{S})$. In the literature it is common to apply magnitude pruning in an iterative fashion by first training the model to learn which connections are important and pruning the weights of least importance.[18] The sparse model is then retrained and the process repeated until the desired sparsity is reached.

One drawback with this approach is that it is computationally demanding: at every step of pruning we need to train the model to convergence. For this reason it is generally better to gradually increase the initial sparsity s_i (which is usually zero) to a final value s_f after some number of steps N:[19]

$$s_t = s_f + \left(s_i - s_f\right)\left(1 - \frac{t - t_0}{N\Delta t}\right)^3 \qquad \text{for } t \in \left\{t_0, t_0 + \Delta t, \ldots, t_0 + N\Delta t\right\}$$

Here the idea is to update the binary masks $\mathbf{M}$ every Δt steps to allow masked weights to reactivate during training and recover from any potential loss in accuracy that is induced by the pruning process. As shown in Figure 8-11, the cubic factor implies that the rate of weight pruning is highest in the early phases (when the number of redundant weights is large) and gradually tapers off.

17 B. Hassibi and D. Stork, "Second Order Derivatives for Network Pruning: Optimal Brain Surgeon," *Proceedings of the 5th International Conference on Neural Information Processing Systems* (November 1992): 164–171, *https://papers.nips.cc/paper/1992/hash/303ed4c69846ab36c2904d3ba8573050-Abstract.html*.

18 S. Han et al., "Learning Both Weights and Connections for Efficient Neural Networks", (2015).

19 M. Zhu and S. Gupta, "To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression", (2017).

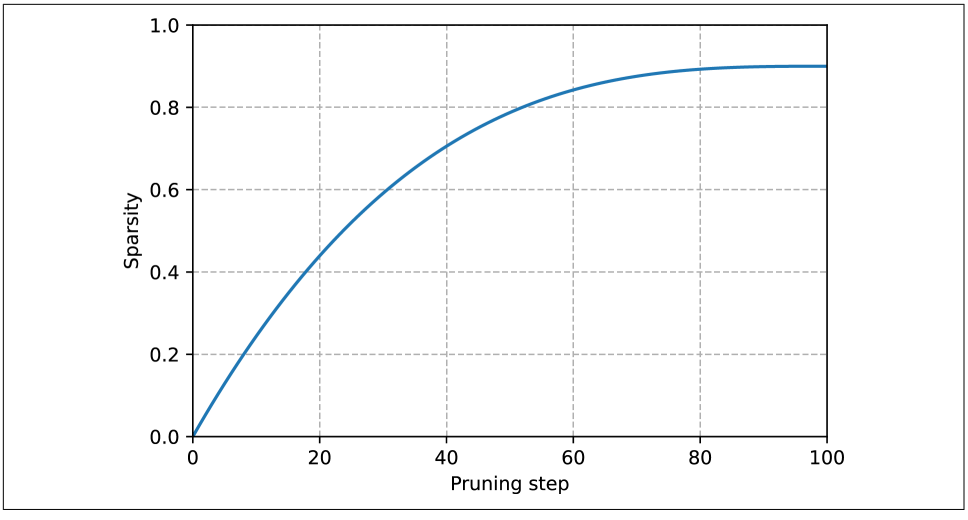


Figure 8-11. The cubic sparsity scheduler used for pruning

One problem with magnitude pruning is that it is really designed for pure supervised learning, where the importance of each weight is directly related to the task at hand. By contrast, in transfer learning the importance of the weights is primarily determined by the pretraining phase, so magnitude pruning can remove connections that are important for the fine-tuning task. Recently, an adaptive approach called movement pruning has been proposed by Hugging Face researchers—let's take a look.[20]

Movement pruning

The basic idea behind movement pruning is to *gradually* remove weights during fine-tuning such that the model becomes progressively *sparser*. The key novelty is that both the weights and the scores are learned during fine-tuning. So, instead of being derived directly from the weights (like with magnitude pruning), the scores in movement pruning are arbitrary and are learned through gradient descent like any other neural network parameter. This implies that in the backward pass, we also track the gradient of the loss L with respect to the scores S_{ij}.

Once the scores are learned, it is then straightforward to generate the binary mask using $\mathbf{M} = \text{Top}_k(\mathbf{S})$.[21]

The intuition behind movement pruning is that the weights that are "moving" the most from zero are the most important ones to keep. In other words, the positive

20 V. Sanh, T. Wolf, and A.M. Rush, "Movement Pruning: Adaptive Sparsity by Fine-Tuning", (2020).

21 There is also a "soft" version of movement pruning where instead of picking the top k% of weights, one uses a global threshold τ to define the binary mask: $\mathbf{M} = (\mathbf{S} > \tau)$.

weights increase during fine-tuning (and vice versa for the negative weights), which is equivalent to saying that the scores increase as the weights move away from zero. As shown in Figure 8-12, this behavior differs from magnitude pruning, which selects as the most important weights those that are *furthest* from zero.

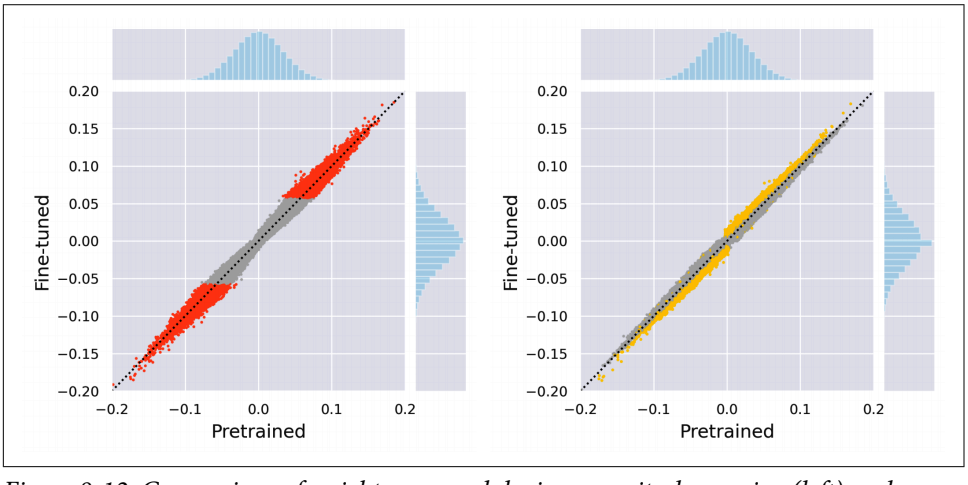


Figure 8-12. Comparison of weights removed during magnitude pruning (left) and movement pruning (right)

These differences between the two pruning methods are also evident in the distribution of the remaining weights. As shown in Figure 8-13, magnitude pruning produces two clusters of weights, while movement pruning produces a smoother distribution.

As of this book's writing, 🤗 Transformers does not support pruning methods out of the box. Fortunately, there is a nifty library called *Neural Networks Block Movement Pruning* that implements many of these ideas, and we recommend checking it out if memory constraints are a concern.

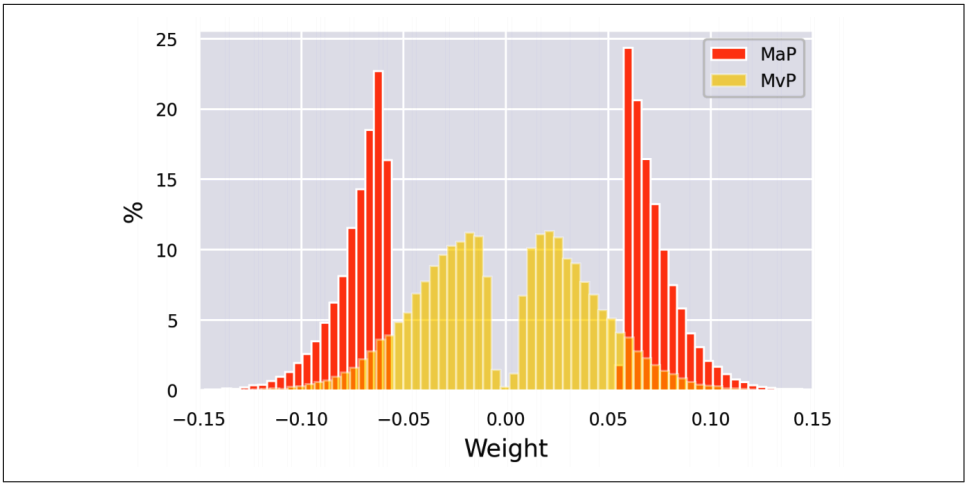


Figure 8-13. Distribution of remaining weights for magnitude pruning (MaP) and movement pruning (MvP)

Conclusion

We've seen that optimizing transformers for deployment in production environments involves compression along two dimensions: latency and memory footprint. Starting from a fine-tuned model, we applied distillation, quantization, and optimizations through ORT to significantly reduce both of these. In particular, we found that quantization and conversion in ORT gave the largest gains with minimal effort.

Although pruning is an effective strategy for reducing the storage size of transformer models, current hardware is not optimized for sparse matrix operations, which limits the usefulness of this technique. However, this is an active area of research, and by the time this book hits the shelves many of these limitations may have been resolved.

So where to from here? All of the techniques in this chapter can be adapted to other tasks, such as question answering, named entity recognition, or language modeling. If you find yourself struggling to meet the latency requirements or your model is eating up all your compute budget, we suggest giving one of them a try.

In the next chapter, we'll switch gears away from performance optimization and explore every data scientist's worst nightmare: dealing with few to no labels.