Chapter 10

Efficient Inference

10.1 Introduction

In the past, the typical machine learning workflow in industry was "relatively" straightforward: collect data for the problem at hand, train a model on that data, and then deploy the trained model directly into production to respond to new inputs.

However, with the advent of increasingly large models, this traditional approach has become less feasible. The large size of modern models often makes it impractical to deploy them in production as they are, at least for most companies or institutions. This can be due to constraints such as latency requirements for the specific applications these models are used for, or due to the prohibitive infrastructure costs associated with running these models at scale.

As a result, the typical machine learning workflow in the industry has evolved. Instead of deploying the model directly after training, a compression step is now commonly introduced. This compression step takes the model produced after training and typically produces a smaller version—smaller in terms of the number of parameters or the memory required to store them—while striving to maintain or even improve the performance of the original model. Nowadays, model compression is essential for meeting system requirements for certain applications, such as reducing latency or simply to reduce infrastructure costs.

One example from industry is the eBay case study (see Xue et al. [2023]). They improved their recommender system by fine-tuning BERT on eBay item titles, which often include product-specific words, alongside the Wikipedia corpus, that is more suited

to the needs of eBay language understanding tasks. In offline evaluations, this eBERT model outperforms out-of-the-box BERT models on a collection of eBay-specific tagging tasks. However, the eBERT model was too large to achieve low latency specifications at inference time, making it challenging to deliver recommendations in real time. To address this issue, eBay used model compression techniques to produce a smaller version of BERT that met their inference requirements. They also optimized the model for CPU inference to reduce costs. For more details on their work, see Xue et al. (2023).

In Hamlet's professional experience at Criteo, he has seen that compression techniques are crucial not only for cost reduction, which is a significant concern, but also for meeting system design constraints. While massive parallelization techniques can be employed to deploy models and potentially meet latency requirements, in some cases, the high costs associated with these solutions can diminish the potential benefits of the system.

In this chapter, we introduce the most common techniques used in the industry for efficient inference, such as knowledge distillation, and model quantization. To illustrate these techniques, we use FinBERT—the model applied in our practical application in the last chapter—as a case study, demonstrating how to make it faster at inference. We conclude the chapter by showcasing techniques for customizing LLMs, allowing you to adapt models to your specific needs.

10.2 Scaling Large Language Models: High Performance, High Computational Cost, and Emergent Abilities

As discussed in the previous chapter, where we explored one of the first, most widely used, and successful large language models, BERT, the modern approach to solving a variety of NLP tasks involves a two-step training process: pre-training and fine-tuning. In the first step, a large model is pre-trained on vast amounts of data, then fine-tuned to solve specific NLP tasks.

Nowadays, there is strong evidence that improving the model's performance during the pre-training phase leads to improvements across multiple NLP tasks. For example, in the paper by Devlin et al. (2019), they empirically demonstrated that the larger version of BERT, named BERT-Large, outperformed the smaller version, BERT-Base, during pre-training and consistently outperformed it across all the downstream NLP tasks considered.

Given this evidence, a natural question arises: if strong performance during pre-training is crucial, how should we approach this phase effectively? Among the many variables and strategies available, which are the most critical?

In the paper by Kaplan et al. (2020), the authors found that one path to improving performance lies in scaling up the models—specifically, increasing the number of model parameters, the amount of training data, and the duration of training time. When these factors are increased, they lead to significant gains in performance. The authors empirically found that performance follows a power-law relationship with respect to these factors, as shown in Figure 10.1.

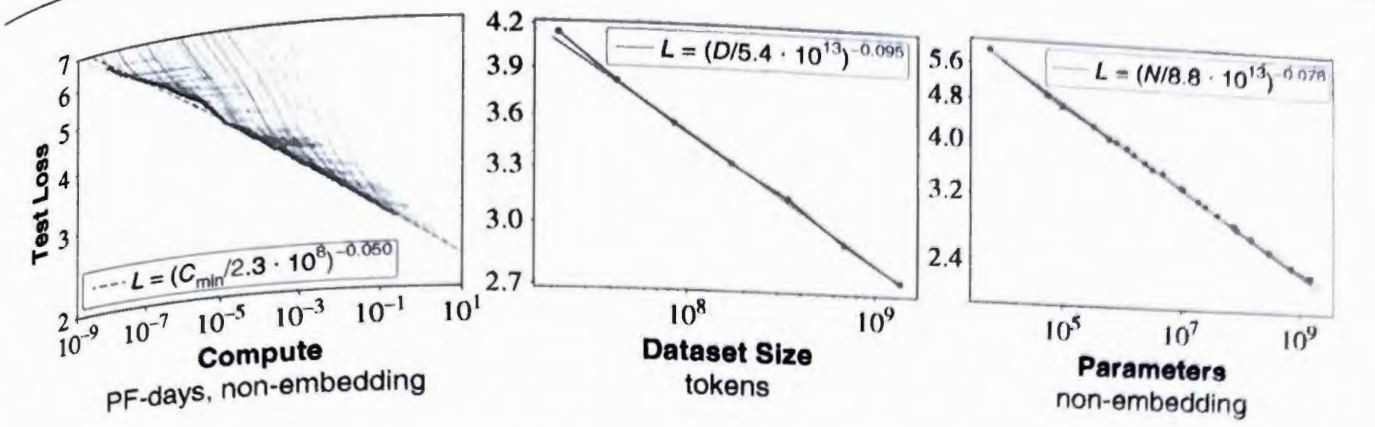

Figure 10.1 Figure 1 from Kaplan et al. (2020): Illustration of how language modeling performance improves with increases in model size, dataset size, and the amount of compute used for training. For optimal performance, all three factors must be scaled up in tandem. Empirical performance exhibits a power-law relationship with each individual factor when the other two are not limiting.

For more details about how other parameters affect performance, refer to the paper by Kaplan et al. (2020).

These empirical laws, which suggest that increasing any of the mentioned factors can lead to almost predictable improvements in model performance, have further motivated the industry to scale up models. Scaling up models is also crucial for unlocking *emergent abilities* of large models, which we briefly discuss on the next section.

While this path of scaling up is highly effective for improving performance, it also comes with some drawbacks, such as increased inference costs and the infrastructure required to serve these large models.

10.2.1 *Emergent Abilities*

While the scaling laws provide a powerful path for improving model performance, recent evidence suggests that they are also key for unlocking the *emergent abilities* of large language models. Emergent abilities is defined by Wei et al. (2022) as, "An ability is emergent if it is not present in smaller models but is present in larger models."

There are two main aspects of emergent abilities that are important to know:

- They cannot be directly predicted by extrapolating from scaling laws (like the one discussed earlier).
- These abilities begin to emerge only after the model reaches a certain scale.

In the following section, we will use the main work by Wei et al. (2023) on emergent abilities of large models. Particularly, we will illustrate how model size and training time contribute to the emergence of these abilities.

10.2.2 *Impact of Model Size*

To illustrate the impact of model size on emergent abilities, let's examine how Large Language Models (LLMs) perform in solving middle school math word problems.

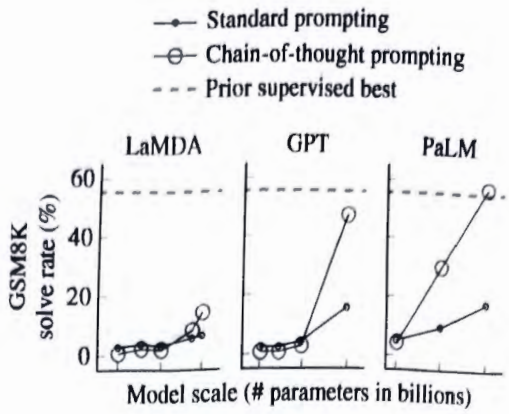

Figure 10.2 Figure 4 from Wei et al. (2023): An illustration of how performance (solve rate) improves as model scale, measured in billions of parameters, increases for standard prompting versus chain-of-thought prompting. Chain-of-thought prompting enables LLMs to solve challenging math problems, with chain-of-thought reasoning notably emerging as an ability with increasing model scale.

In the work by Wei et al. (2023), the GSM8K dataset is used for this evaluation, where the LLMs are asked to solve math problems using both standard prompting and chain-of-thought prompting. According to Wikipedia, "Chain-of-thought (CoT) prompting is a technique that allows large language models (LLMs) to solve a problem as a series of intermediate steps before giving a final answer" (see [https://en.wikipedia.org/wiki/Prompt_engineering]).

In Figure 10.2, you can see the performance measured as solve rate (%) versus model scale, represented by the number of model parameters in billions.

The plot shows the performance of three different models: LaMDA, GPT, and PaLM. A clear pattern emerges for each of these models: as the number of parameters increases, so does the performance. For example, focusing on the results using chain-of-thought prompting, which tends to yield better results, LaMDA, with 137 billion parameters, achieves a performance of less than 20%. In contrast, GPT, with 175 billion parameters, achieves a performance exceeding 40%. PaLM, with 540 billion parameters, reaches a performance of around 50%.

10.2.3 Effect of Training Time

Evidence suggests that emergent abilities only appear after a large number of training iterations. It's not enough to simply have a large model; it must also be trained extensively to unlock these capabilities.

In Figure 10.3, taken from the paper by Wei et al. (2022), we see the performance of various models—LaMDA, GPT-3, Gopher, Chinchilla, PaLM, and a Random Model—on eight different common NLP tasks, like Modular Arithmetic, Multi-task Natural Language Understanding (NLU), Word-in-Context, among others.

Each graph shows model performance (measured in accuracy) as a function of the model scale (training FLOPS, or training floating point operations per second). For example, looking at the Modular Arithmetic task, large models like GPT-3 and PaLM show a significant performance boost only when their training FLOPS exceed 10^{22}.

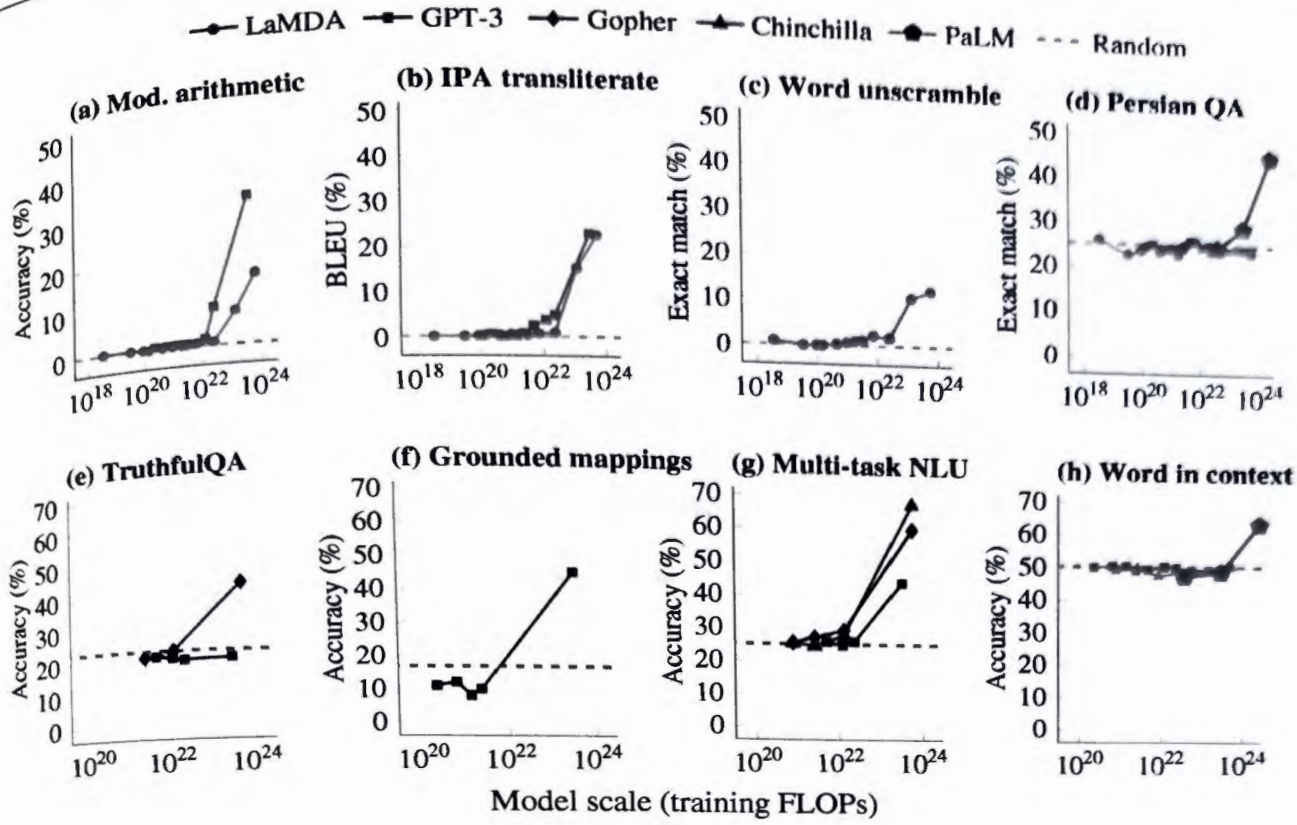

Figure 10.3 Figure 2 from Wei et al. (2022): Illustration of performance, measured by accuracy, versus model scale (measured in training FLOPs) for eight different tasks. The figure shows how performance transitions from random levels to significantly above random after reaching a certain scale threshold. These tasks demonstrate examples of emergent abilities in the few-shot prompting setting, which only appear after sufficient training.

Similarly, for the Multi-task NLU task, models such as Chinchilla, Gopher, and GPT-3 only start improving once their training FLOPS pass the same threshold. We observe the same behavior for the Word-in-Context task as well.

The key takeaway is that not only do we need large models, but we also need to train them for many iterations to improve their performance and unlock emergent abilities.

However, it is important to note that even after extensive training, large models can be slow during inference due to their huge number of parameters. This creates the need for methods to speed up inference, as the ones described in this chapter.

Lastly, although not explicitly shown in the figures, the amount of training data is also a critical factor in the model's performance. Large models benefit from vast amounts of data to learn from, further enhancing their capabilities.

10.2.4 *Efficient Inference for Deep Models*

To develop efficient inference for deep models, recent work has focused on model compression and acceleration techniques, such as the following:

1. Knowledge distillation
2. Quantization

3. Parameter pruning
4. Neural Architecture Search

In this chapter, we will focus only on techniques 1 and 2.

10.3 Making FinBERT Faster

10.3.1 Knowledge Distillation

Knowledge distillation is a technique used to create a faster model (the *student*)—meaning one with lower inference time—that approximates the performance of a slower but more powerful model (the *teacher*). In this process, the student model is by design smaller than the teacher model, with fewer parameters, making it more efficient at inference time. Knowledge distillation involves a learning process for the student model where it not only learns from data, as in typical training, but also has knowledge transferred from the teacher model.

In our case study, the teacher model we want to approximate is FinBERT; see Araci (2019). As a reminder, FinBERT is a fine-tuned version of BERT that has been specifically trained for sentiment analysis prediction in the financial domain.

If our only concern were inference speed, we could simply train the student model using the given data (which, for example, could be a combination of pre-training on the BERT corpus followed by fine-tuning with the financial data used by FinBERT). However, because the student model is smaller than the teacher, it would likely underperform compared to the teacher. Knowledge distillation offers a different approach to this situation: it involves not only learning from the data but also aligning the student model with certain aspects of the teacher model. In its simplest form, this alignment could involve matching the probability distribution that the teacher model assigns to a given input, as described by Hinton et al. (2015) in their paper on knowledge distillation. (Yes, the same Hinton who shared the 2024 Nobel Prize in Physics with John Hopfield and is considered one of the Godfathers of Deep Learning. See https://en.wikipedia.org/wiki/Geoffrey_Hinton.)

An analogy can help clarify this process. Consider how we learn a subject in school: imagine we have the option to study directly from a textbook or to attend lectures given by a teacher who is an expert in the subject. Most of the time, a combination of both—attending lectures and reading the textbook—leads to a better understanding than either approach alone. In this analogy, the textbook represents the data used in standard training, while the teacher represents the model that also guides the learning process. Similarly, in knowledge distillation, the student model benefits from learning both from the data and from the guidance provided by the teacher model.

To enable the student model to learn both from the data and from the teacher model, the cost function used during training must account not only for how well the

student model fits the data but also for how well it matches the specific aspects of the teacher model that we want to capture.

We will explore how to apply knowledge distillation to FinBERT, demonstrating how this technique can be used to create a faster model without sacrificing too much performance.

10.3.1.1 Which Aspect of the Teacher Model to Match. In knowledge distillation, there are various aspects of the teacher model that the student model can try to align with. These different aspects are known in the knowledge distillation literature as different types of knowledge. For example, in the most basic form of knowledge distillation, often called vanilla knowledge distillation, the focus is on aligning the probability distribution that the teacher model assigns to different classes given the input, as in the case of a classification task. This approach was popularized by Hinton et al. (2015).

However, there are other forms of knowledge that can be transferred from the teacher to the student model. For instance, the intermediate representations computed by the network are a potential source of knowledge that can be transferred. Some methods use the activation features of intermediate layers in the teacher model to guide the learning of the student model, as described by Romero et al. (2015) in their work on FitNets. Also, the parameters or weights of the teacher model can also be a different source of knowledge.

These different types of knowledge to transfer fall into three categories: response-based knowledge, feature-based knowledge, and relation-based knowledge. A very good overview of these classification can be found in the excellent survey on knowledge distillation by Gou et al. (2021).

For the purpose of this book and in our case study, we will focus on the response-based knowledge, which is the simplest yet highly effective method for model compression.

10.3.1.2 Response-based Knowledge. Response-based knowledge refers to a scenario where the student model attempts to mimic or match the output of the teacher model's last layer, meaning that we try to match its final prediction.

In typical classification tasks, the most popular form of response-based knowledge is known as *soft targets*, as described in Hinton et al. (2015). In this context, soft targets refer to the probability distribution allocated by the teacher model over the different classes for a given input. This contrast with the "ground truth" or *hard targets*, where the probability mass is fully allocated to the "correct class."

In neural networks for classification tasks, the class probabilities are usually generated by applying a "softmax" function to the logits, which are the outputs of the last fully connected layer of the network. Following the notation introduced in Hinton et al. (2015), we denote the outputs logits as the vector $\mathbf{z}$, whose dimension depends on the number of classes specified by the classification problem. (Logits are the outputs of a neural network before the softmax activation function is applied. They are scores of the input belonging to a certain class. Recall the discussion on Logistic Regression

in Chapter 3). The probability distribution is then computed by applying the softmax function to **z**:

$$p(z_i, T) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

where z_i is the output corresponding to the i-th class, and T is a parameter known as the *temperature*. Typically, in classification problems, T is set to 1, but in knowledge distillation, using larger values of T tends to produce a "softer" probability distribution over the classes.

For example, in Figure 10.4, we show how the output of the softmax function applied to the vector of logits $\mathbf{z} = [30, 20, 10]^T$ changes as we increase the temperature. At a temperature of 1, the output is close to a hard target, with almost all of the probability mass allocated to a single class. As the temperature increases, the probability distribution becomes progressively softer. For very large temperatures—such as in our case temperature = 1,000—the distribution approaches a uniform distribution across all classes. (Physicists will recognize this as the Boltzman distribution that gives the probability that a system will be in a certain state as a function of that state's energy and the temperature of the system.)

As we have discussed, the goal in knowledge distillation is for the student model to learn both from the data and from the teacher model, and specifically in soft targets response-based knowledge, from the soft targets produced by the teacher. To achieve this, we need an objective function for the student that accounts for how well it fits the

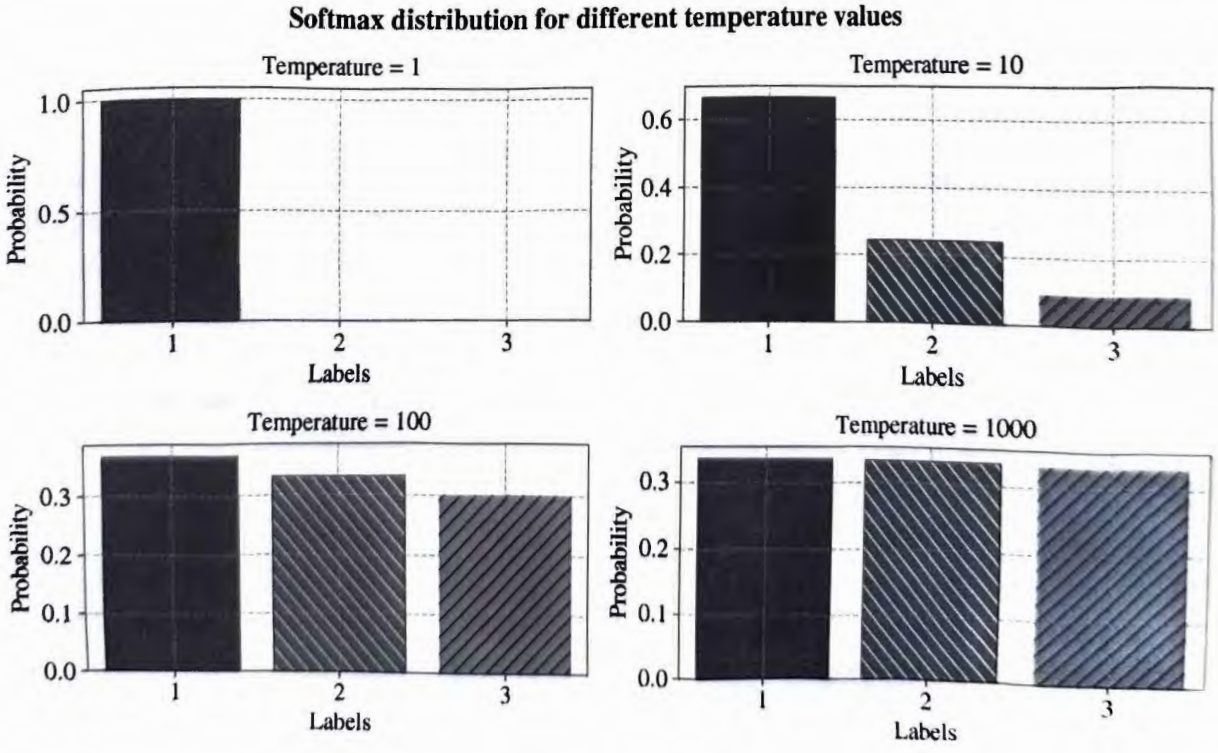

Figure 10.4 Softmax distribution.

empirical data distribution (typically measured by the cross-entropy loss in classification problems) and how well it matches the teacher's soft targets. One approach is to use a weighted average of both objectives:

$$\mathcal{L}_{\text{student}} = \alpha \mathcal{L}_{\text{CE}}(y, p(\mathbf{z}_s, T = 1)) + (1 - \alpha)\mathcal{L}_{\text{KD}}(p(\mathbf{z}_t, T), p(\mathbf{z}_s, T))$$

where $\mathcal{L}_{\text{student}}$ represents the total loss for the student, $\mathcal{L}_{\text{CE}}$ represents the cross-entropy loss between the student model and the ground truth label y, α is the weight hyperparameter between 0 and 1, and $\mathcal{L}_{\text{KD}}$ is the distillation loss that measures how well the student's distribution $p(\mathbf{z}_s, T)$ aligns with the teacher's distribution $p(\mathbf{z}_t, T)$.

The knowledge distillation loss is typically proportional to the Kullback-Leibler (KL) divergence,

$$\mathcal{L}_{\text{KD}}(p(\mathbf{z}_t, T), p(\mathbf{z}_s, T)) = T^2 D_{\text{KL}}(p(\mathbf{z}_t, T), p(\mathbf{z}_s, T))$$

where T is the temperature and D_{KL} is the KL divergence, which measures the difference between the teacher and student distributions. The factor T^2 ensures that the relative contributions of the hard and soft targets remain balanced. For more detailed explanations about the proportionally factor, we encourage you to refer to Hinton et al. (2015)

10.3.1.3 Implementation Details. We will use the Hugging Face (https://huggingface.co) `transformers` python library for performing knowledge distillation due to its powerful API and utilities, which greatly simplifies the training and inference with transformer models. In the repository accompanying this book, you will find a short tutorial covering the basics of this library, making sure you have the have the knowledge needed to understand the relevant components discussed in the book.

In the code snippets provided later, we focus on the most important steps for implementing knowledge distillation. Many steps, such as dataset creation, tokenization, train-test splitting, among others, are common across machine learning and NLP workflows and not specific to this project, so they are not shown here. However, you can find the complete step-by-step process for building this use case in the accompanying notebook for this chapter.

Our goal is to find a balance between theory and specific implementation, allowing you to understand the core concepts of the topic and apply them using different libraries, such as PyTorch Lightning (https://lightning.ai/docs/pytorch/stable/), or even implementing everything in plain Pytorch or TensorFlow. We hope this approach give you a solid understanding of the theory while providing detailed implementation examples in the notebook, which can be updated as new powerful tools become available.

Now that we understand the student loss described earlier—a linear combination of cross-entropy loss (commonly used for classification tasks) and knowledge distillation loss (proportional to the KL divergence between the teacher and student distributions)—we can proceed to implement it.

For standard training procedures, we typically use the `Trainer` class of the transformers library, which provides and API for training models in PyTorch, offering many features out of the box, such as distributed training on multiple GPUs/TPUs, etc. More information can be found in the library documentation https://huggingface.co/docs/transformers/v4.15.0/main_classes/trainer.

Since our objective is to train the model using the distillation loss $\mathcal{L}_{student}$, and the `Trainer` class does not support this loss out of the box, we need to extend its capabilities. Fortunately, this is straightforward.

We extend the standard Trainer class to create a `DistillationTrainer`, as suggested in Tunstall et al. (2022), which supports the student loss function necessary for knowledge distillation. The `DistillationTrainer` can be instantiated as shown in the following code:

```
distilbert_trainer = DistillationTrainer(
    model_init=student_init,
    teacher_model=teacher_model,
    args=student_training_args
    train_dataset=YOUR_TRAINING_SET,
    eval_dataset=YOUR_EVALUATION_SET,
    compute_metrics=compute_metrics,
    tokenizer=student_tokenizer)
```

The `DistillationTrainer` object takes the following inputs:

- `student_init`: a function used to provide the initialization of the student model
- `teacher_model`: the teacher model used for distillation
- `student_training_args`: an instance of the `TrainingArguments` class
- `YOUR_TRAINING_SET`: a `torch.utils.data.Dataset`, `torch.utils.data.IterableDataset`, or `datasets.Dataset`, the dataset used to train the model
- `YOUR_EVALUATION_SET`: a `torch.utils.data.Dataset`, `torch.utils.data.IterableDataset`, or `datasets.Dataset`, the dataset used to evaluate the model
- `compute_metrics`: a function used to report metrics on the evaluation set, such as precision, recall, accuracy, etc.
- `student_tokenizer`: The tokenizer used to preprocess the data

A key difference, from the input arguments point of view, between the `DistillationTrainer` and the standard `Trainer` class is the inclusion of the teacher_model argument, which is essential for knowledge distillation.

From a functional perspective, DistillationTrainer must compute the correct loss function. To do this, it needs access to the α and temperature T hyperparameters. The implementation of the `compute_loss` method is shown here:

```
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):
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        inputs = inputs.to(device)
        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
```

In this customized `Trainer` class, we:

- Save the teacher model in the `__init__` method.
- Compute the student loss in the `compute_loss` method, where:
  - `loss_ce` corresponds to the cross-entropy loss.
  - `loss_fct` is an instance of the `nn.KLDivLoss` object, which computes the KL divergence loss.
  - `loss_kd` corresponds to the knowledge distillation loss between the teacher and student distributions.
  - `loss` is the linear combination of both losses.

As shown in the code, the class accesses α and T as `self.args.alpha` and `self.args.temperature` through the `student_training_args` parameters object. To make these available during training, we extend the TrainingArguments class as shown here:

```
from transformers import TrainingArguments

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

After creating our custom trainer and custom training arguments, we can train our student model using the convenient features provided by the `transformers` library by simply calling `distilbert_trainer.train()`. As previously mentioned, the detailed

step-by-step process for distilling FinBERT is provided in the accompanying notebook, while this section focuses on the key concepts and implementation details.

10.3.2 Case study results. Distilling FinBERT

In the original paper by Araci (2019) on FinBERT, the authors used two different datasets to fine-tune the model. One of these datasets is TRC2-financial, which was employed to improve the relevance of corpus to financial keywords. You can obtain these dataset for research purposes by applying here: https://trec.nist.gov/data/reuters/reuters.html.

The second dataset, Financial PhraseBank created by Malo et al. (2014), is the main dataset used the FinBERT paper for the specific task of sentiment analysis. It can be obtained for free here: https://www.researchgate.net/publication/251231364_FinancialPhraseBank-v10. In our experiments, this is the dataset we used to perform knowledge distillation on FinBERT.

To ensure a fair comparison, we used the same train-test split specified by the original authors, as provided in their GitHub repository, FinBERT: Financial Sentiment Analysis with BERT (https://github.com/ProsusAI/finBERT). This allowed us to report evaluation metrics that are comparable to those in the original paper. The evaluation metrics we used to assess performance are the same as those in the original study, namely accuracy, and macro F1 average.

Our first step was to reproduce the results reported in the original paper, which we successfully did on the test portion of the TRC2-financial sentiment dataset.

For the student model, we selected `distilbert/distilbert-base-uncased`, provided by Hugging Face at https://huggingface.co/distilbert/distilbert-base-uncased. The Transformers library makes it easy to download models from HuggingFace. More information about this process can be found in the accompanying notebook for this chapter and the HuggingFace tutorial in the book repository.

In terms of performance, our DistilledFinBERT slightly outperforms the teacher model, FinBERT, which is an excellent result. Detailed results can be found in Table 10.1.

Since we managed to maintain or even improve performance, how does it compare in terms of inference speed? The results are shown in Table 10.2.

DistilledFinBERT achieves a 2.125 times faster inference speed compared to the original FinBERT. In conclusion, our DistilledFinBERT not only matches the performance of the original model on the same test set, but it also offers significantly faster inference, making it a more efficient option for deployment.

Table 10.1 Performance metrics of teacher vs. student models.

Model	Precision	Recall	F1-Score
FinBERT	0.85	0.84	0.84
DistilledFinBERT	**0.86**	**0.85**	**0.85**

Table 10.2 Inference speed of teacher vs. student models.

Model	Average Latency (ms)
FinBERT	0.034
DistilledFinBERT	**0.016**

10.4 Model Quantization

Quantization refers to the process of reducing the set of values a variable can take so that the variable can be represented using fewer bits, thereby reducing its precision. The key question behind quantization is: Can deep neural networks operate effectively with lower precision?

In deep learning, we are particularly interested in applying quantization to the weights, biases, and activations of our models. Reducing the bit precision of these elements leads to improvements in memory storage, computational speed, and energy consumption. Modern hardware is equipped with specialized instructions for integer arithmetic, which further enhance computational speed when operating in lower precision. However, this reduction in precision can come at the cost of accuracy, as lower bit representation might lose information.

Training and inference in deep neural networks have distinct precision requirements. During training—a highly dynamic process involving a wide range of weight values—a large dynamic range is important. On the other hand, inference often prioritizes precision over a small dynamic range, as the model makes predictions based on pre-trained weights that often concentrates around a particular value. By understanding the different requirements of training and inference, we can design data types optimized for each stage.

As of the time of writing, there are two main quantization techniques: K-Means quantization and linear quantization. In this section, we will focus on linear quantization and explore how it can be applied to achieve efficient model inference.

10.4.1 *Linear Quantization*

Linear quantization applies an affine mapping from the space of integers to the space of real numbers (or floating-point space) using the following formula: $r = S(q - Z)$, as described in Jacob et al. (2017). Here:

- S is the scale, a floating-point value.
- Z is the zero point, an integer that represents the offset or bias.
- q is the quantized number, an integer.
- r is the real value, a floating-point number.

The formula is a reconstruction formula, mapping quantized values q back to real values r. In practice, though, we usually start with the real value r and need to find its quantized counterpart, q.

Table 10.3 Integer range for different bit widths.

Bit width	q_{min}	q_{max}
2	-1	1
3	-4	3
4	-8	7
N	-2^{N-1}	$2^{N-1}-1$

We can view this equation as a mapping from the integer range $[q_{min}, q_{max}]$ to the floating-point range $[r_{min}, r_{max}]$. The range of the integer representation depends on the number of bits used. For example, in signed integer representations, the range of values for different bit widths is shown in Table 10.3:

The floating-point range $[r_{min}, r_{max}]$ is determined by the statistics of the real values input to be quantized. To determine the scale S, we impose that q_{min} maps to r_{min} and q_{max} maps to r_{max}, solving the following system of equations:

$$r_{max} = S(q_{max} - Z)$$
$$r_{min} = S(q_{min} - Z)$$

Subtracting these two equations gives us the value of S:

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$

The zero point Z is found by solving for Z using $r_{min} = S(q_{min} - Z)$, which results in:

$$Z = \text{round}\left(q_{min} - \frac{r_{min}}{S}\right)$$

Since Z is an integer, we round it. To compute the quantized value q, we use:

$$q = r/S + Z$$

Finally, q is clamped to the range $[q_{min}, q_{max}]$ and rounded to the nearest integer. Although there are different methods for rounding, this example uses the simplest case, but adaptive rounding methods also exist.

10.4.1.1 Example of Linear Quantization. Let's illustrate linear quantization with a toy example to give a better understanding of the operations involved. Consider a weight matrix that we want to quantize using 2 bits:

```
weights = np.array([
 [-1.0856306   0.99734545  0.2829785 ]
 [-1.50629471 -0.57860025  1.65143654]
 [-2.42667924 -0.42891263  1.26593626]
 ])
```

First, we compute the scale S. The dynamic range of the weight matrix, defined by its minimum and maximum values, is $r_{min} = -2.42667924$ and $r_{max} = 1.65143654$.

Using a 2-bit representation, the integer range is $[-2, 1]$. From this, we compute the scale as $S = 1.35937$. We can then find the zero point $Z = 0$. Finally, we compute the quantized weights using the equation provided:

```
quantized_weights = (weights / scale + zero_point).clamp(qmin, qmax).round()

>>> quantized_weights
np.array([
[[-1.  1.  0.]
 [-1. -0.  1.]
 [-2. -0.  1.]]
])
```

If we attempt to reconstruct the original matrix from the quantized version, we obtain:

```
>>> reconstructed_weights
np.array([
 [-1.3594  1.3594  0.    ],
 [-1.3594 -0.      1.3594],
 [-2.7187 -0.      1.3594],
 ])
```

The reconstruction error is the difference between the original and reconstructed weights:

```
>>> reconstruction_error
np.array([
 [ 0.2737 -0.362   0.283 ],
 [-0.1469 -0.5786  0.2921],
 [ 0.2921 -0.4289 -0.0934]
 ])
```

10.4.2 *Quantizing an Attention Layer in Distilled FinBERT*

Now, let's apply this process to one of the attention layers in our Distilled FinBERT model using 8 bits. Figure 10.5 is a distribution of the parameters of the layer, following a similar setup to that in Tunstall et al. (2022).

As shown in Figure 10.5, the weights are concentrated within a narrow range of values, exhibiting a small dynamic range. Since we are using signed 8-bit integers, the range is $[-127, 128]$. The scale factor and zero point are calculated similarly to the previous example.

Next, we compute the quantized weights using the formula $q = r/S + Z$, clamp the values, round them to the nearest integer, and store them in the `torch.int8` data type:

```
manually_quantized_weights = (weights / scale + zero_point).clamp(qmin,
qmax).round().char()
>>> manually_quantized_weights
tensor([[ -5,  -8,   0, ...,  -6,  -3,   8],
        [  8,   3,   1, ...,  -4,   7,   1],
        [ -9,  -5,   5, ...,   0,   6,  -3],
```

```
        ...,
        [  5,   0,  13,  ...,   0,   6,  -1],
        [  0,  -2, -12,  ...,  12,  -8, -13],
        [-13,  -1,  -9,  ...,   8,   2,  -2]], dtype=torch.int8)
```

Torch also provides a built-in function, `torch.quantize_per_tensor`, for this process. Here's how the result compares:

```
from torch import quantize_per_tensor
# torch as a function which performs this operation for us
dtype = torch.qint8
quantized_weights = quantize_per_tensor(weights, scale, zero_point, dtype)

quantized_weights.int_repr()

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

Finally, let's compare the results:

```
# make sure they are equivalent

assert abs(quantized_weights.int_repr() -
manually_quantized_weights).sum().item() <= 1e-5
```

Figure 10.6 is the distribution of the quantized weights.

To verify the gain in memory efficiency from moving from FP32 to INT8, we can measure it directly in code:

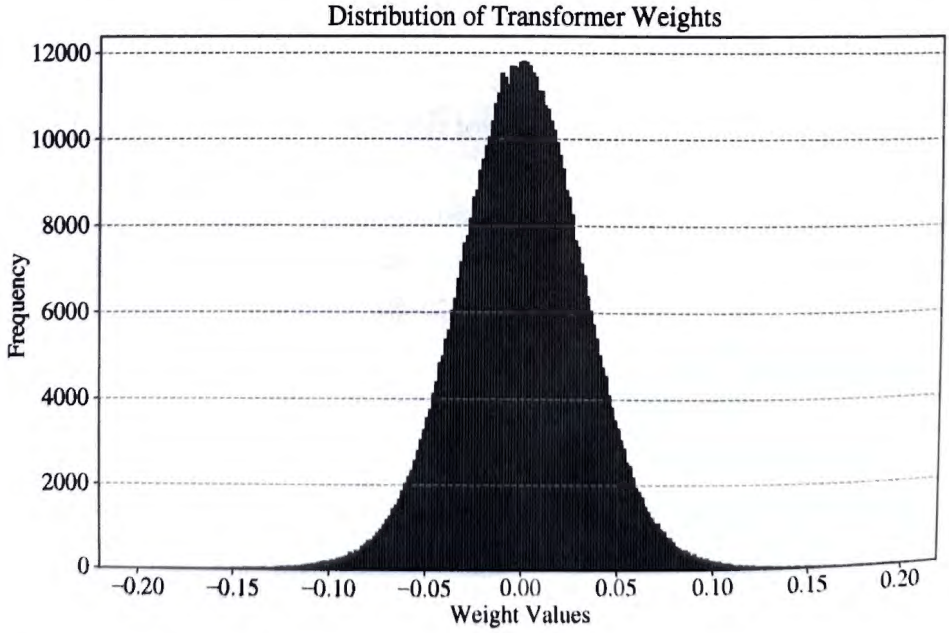

Figure 10.5 Weights distribution.

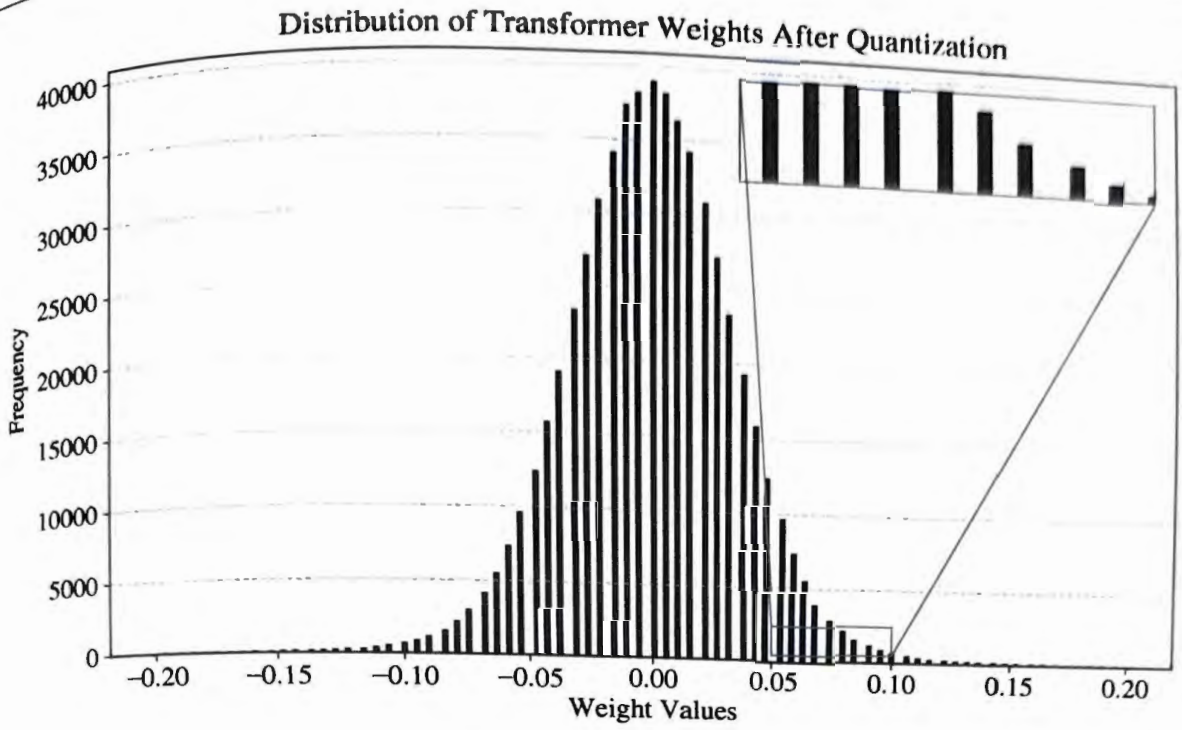

Figure 10.6 Distribution of quantized weights.

Table 10.4 Performance table for speedups because of quantization.

Data Type	Time (s)
float32	0.006926
int8	0.000266
Speedup	26.06x

```
import sys

# 4x time compression!

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

>>> 3.999755879241598
```

We also measure the speedup of performing the operation `weights*weights`, as shown in Table 10.4.

As demonstrated, quantization provides significant improvements in both memory footprint and computational speed. For more details, refer to the associated notebook for this chapter.

10.4.3 *Experiment Results with Linear Quantization on Distilled FinBERT*

In this section, we apply linear quantization to the entire Distilled FinBERT model and evaluate the improvements in memory storage and computational speed, along with any potential loss in accuracy due to representing the model's weights with lower precision. (As a side note, in these experiments, quantization is applied only during the inference process.) Since at the time of writing PyTorch currently supports linear quantization

Table 10.5 Inference speed after quantization.

Model	Accuracy (%)	Average Latency per Sentence (s)
FinBERT	83.9175	0.1533
DistilledFinBERT	85.3608	0.1060
QuantizedDistilledFinBERT	83.7113	0.0458

only on CPU devices, the results presented here are for CPU-based inference. The results are summarized in Table 10.5.

As shown, the quantization process introduces some loss in accuracy, with the model's performance dropping from 85.36% in the distilled version to 83.71% in the quantized version. However, the quantized Distilled FinBERT still matches the original FinBERT's accuracy (83.92%), with only a slight difference of 0.2%. In contrast, the quantized model achieves a significant speedup, performing inference 3.35 times faster than the original model.

This result demonstrates the power of efficient inference: maintaining nearly identical performance while delivering significantly faster predictions. Details on how to apply quantization to the model are provided in the accompanying chapter notebook.

10.5 Customizing Your LLM: Adapting Models to Your Needs

LLMs are incredibly powerful tools that can speed up many kinds of tasks. In earlier chapters, we showed how they can help generate code for trading strategies or assist in developing trading ideas. However, as we have seen, getting satisfactory answers often requires extensive prompt engineering and adding extra context to guide the model to produce relevant outputs.

Sometimes, though, you might already have a lot of useful information, like a codebase or internal documentation, that you would like the LLM to access directly. By customizing the LLM to work with this knowledge, you can reduce the amount of prompt engineering and get answers that are much more tailored to your specific needs. For example, this would allow software engineers, quant researchers, or discretionary traders to interact with a knowledge base more effectively, getting responses that are more relevant and aligned with their work.

As of the time of writing, there are two main ways to customize an LLM to work with your own data. One involves fine-tuning, also known as post-training, which as we have seen involves retraining the model with your specific data to make it better for your tasks. The other approach does not require retraining the LLM but instead enriches the user's prompt by appending extra information as part of the input, allowing the LLM to use that data to answer specific questions.

A popular method in this second category is Retrieval Augmented Generation (RAG). In RAG, contextual information is automatically retrieved from an external

database and appended to the user's prompt, helping the LLM generate more relevant or "grounded" responses.

The contextual information is dynamically retrieved from the database by measuring the similarity between the user's input prompt and the files in the database. The main idea is to represent both the files in the database and the user's prompt as mathematical objects, such as vectors (embeddings). Once we have these representations, a similarity measure can be applied to identify and retrieve the most relevant files for the input prompt. For more information about RAG, please refer to the OpenAI blog post Retrieval-augmented Generation (RAG) and Semantic Search for GPTs at https://help.openai.com/en/articles/8868588-retrieval-augmented-generation-rag-and-semantic-search-for-gpts.

In the next section, we will provide a brief overview of techniques for fine-tuning (or post-training) to customize your LLMs. While we will not delve into the technical details—that is beyond the scope of this book—we will cover the key concepts and focus on one specific technique to build a practical example you can try yourself.

10.5.1 Fine-tuning Techniques

10.5.1.1 Traditional Fine-tuning (FT). FT involves retraining all the parameters of the model, which can be computationally expensive and time-consuming as models scale. Additionally, it comes with the risk of catastrophic forgetting, where the model loses a considerable part of the knowledge gained during pre-training, leading to degraded generalization performance.

To address these challenges, academia and industry have developed techniques that involve adapting only a subset of a model's parameters during training. These techniques fall under the category of parameter-efficient fine-tuning (PEFT), making models faster to adapt, more memory-efficient, and capable of achieving comparable or even better performance than traditional fine-tuning.

10.5.1.2 Parameter-efficient Fine-tuning (PEFT). PEFT focuses on adapting a subset of model's parameters during fine-tuning, which reduces computational requirements, storage needs, and training time compare with FT. Following are some key PEFT methods.

10.5.1.2.1 BitFit. BitFit updates only the bias terms of the network rather than all the model's parameters; see Zaken et al. (2021). The authors have shown that applying BitFit to BERT-like architectures—models we've explored in this and the previous chapter—achieves performance comparable to full fine-tuning on small-to-medium datasets. Remarkably, for models like BERTBase and BERTLarge, the bias terms represent less than 0.1% of the model's parameters, yet modifying only this small percentage appears to be enough for such datasets. For more details, refer to Zaken et al. (2021).

10.5.1.2.2 Adapters. Adapters (Houlsby et al., 2019) introduce trainable layers into the network. During adaptation, only these newly added layers are updated, while the original model parameters remain frozen. Authors have shown that adapters achieve near state-of-the-art performance at the time of publication. However, adding new layers to the network increases both storage requirements and inference latency. More details can be found in Houlsby et al. (2019).

10.5.1.2.3 Prompt-tuning. Prompt-tuning involves enriching the input prompt with a learnable prompt to help the LLM perform well on a downstream task, such as computing the sentiment of a given text, see the paper *The Power of Scale for Parameter-efficient Prompt Tuning* by Lester et al. (2021). The authors of this paper show that prompt-tuning achieves accuracy comparable to traditional fine-tuning as the model size increases. An extension of prompt-tuning is prefix-tuning, introduced in the paper *Prefix-tuning: Optimizing Continuous Prompts for Generation* by Li and Liang (2021), which adds learnable prompts to each layer of the transformer. However, this method introduces additional inference latency due to the learnable prompts and reduces the available input prompt length.

10.5.1.3 LoRA (Low-rank Adaptation of Large Language Models). LoRA, see Hu et al. (2021), is similar to adapters in the sense that it introduces additional learnable layers, but with the difference that it does so as a parallel branch (see Figure 10.7), which provides the advantage of not increasing inference latency. Like adapters, LoRA keeps the original network parameters fixed and trains additional parameters, which are then combined with the original network via addition.

For example, following the terminology of the paper, consider a dense layer in the network with parameters W. If the input to this layer is x, the output is denoted as $h = Wx$. In LoRA, an additional layer, ΔW, is introduced as a parallel branch. The output of LoRA then becomes:

$$h = Wx + \Delta Wx$$

During training, the parameters of W remain fixed (frozen), while the parameters of ΔW are adapted.

The key aspect of LoRA is that the matrix ΔW is constrained to be a low-rank matrix, defined as $\Delta W = BA$. For example, if W has dimensions $d \times k$, ΔW must also have dimensions $d \times k$. However, B and A have dimensions $d \times r$ and $r \times k$, respectively, where $r \ll \min(d, k)$. This, in LoRA, the forward pass is modified as follows:

$$h = Wx + \Delta Wx = Wx + BAx = (W + BA)x = W'x$$

So, once the weights B and A are learned and combined with the original network to form W', no extra latency is introduced at inference time because W and W' have

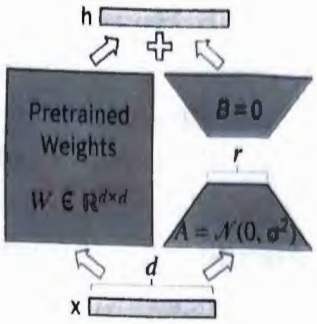

Figure 10.7 Figure 1 from Hu et al. (2021): This figure illustrates the reparameterization of new layers added as a parallel branch using LoRA. Only the matrices A and B are trained, while pretrained weights W are not updated. The figure also shows the initialization of matrices A and B.

the same dimensions. Authors in Hu et al. (2021), propose to initialize A with random Gaussian values and B with zeros, so making sure that $\Delta W = BA$ starts as zero at the beginning of training (see Figure 10.7).

10.5.1.3.1 Efficiency of LoRA. Authors have shown that for large models like GPT-3 (175B parameters), a very low rank r, such as 1 or 2, suffices even when the full rank d is as high as 12,288. This makes LoRA highly storage and compute efficient.

In their experiments, the authors demonstrated that, compared to fine-tuning GPT-3 175B with Adam optimizer, LoRA reduces the number of trainable parameters by 10,000 times and the GPU memory requirements by three times. See Hu et al. (2021) for more details on the results and experiments.

10.5.1.3.2 QLoRA. QLoRA, see Dettmers et al. (2024), extends LoRA by incorporating quantization, making it even more storage and compute efficient. More details about QLoRA can be found at Dettmers et al. (2024).

Fortunately for us, most of these PEFT techniques have already been implemented by the Hugging Face team. You can learn more in their blog post announcing the release of the PEFT library: Parameter-Efficient Fine-Tuning of Billion-Scale Models on Low-Resource Hardware at https://huggingface.co/blog/peft. For a more in-depth exploration, visit the PEFT GitHub repository at https://github.com/huggingface/peft, which provides implementations of state-of-the-art PEFT methods fully compatible with the Hugging Face Library. Also, if you want to learn more about efficient deep learning, check out the MIT course *TinyML and Efficient Deep Learning Computing* at https://hanlab.mit.edu/courses/2024-fall-65940. This excellent resource covers more topics than we can address in a single chapter.

As an example, in the accompanying notebook for this chapter, we will illustrate how to fine-tune the Llama 3 model using LoRA. This approach enables can be extended by you to develop various applications, such as customized Q&A systems for internal use or customer-facing applications.

10.5.1.4 Aligning Your LLM with Human Preferences. Another key aspect of fine-tuning is aligning LLM responses with human preferences. After the initial pre-training phase, model outputs can sometimes be unhelpful or even toxic. Techniques like **Reinforcement Learning from Human Feedback** (RLHF), introduced by Ouyang et al. (2022), help language models generate responses that are more human-like, creative, truthful, and helpful. RLHF also helps to reduce issues like biased or inaccurate content and was one of the major advancements used by OpenAI that transformed GPT-3, into the remarkable ChatGPT that we all love.

A newer approach, direct preference optimization (DPO), simplifies the RLHF process while improving stability, performance, and computational efficiency. For more details about DPO see Rafailov et al. (2024).

Interestingly, PEFT methods—like the ones we talked about earlier—can also be used with RLHF and DPO, making fine-tuning more efficient.

These techniques are important for customizing LLMs to specific use cases. However, since they are beyond the scope of this book—at least in this edition—we will not go into the details. That being said, they are worth keeping an eye on as this field continues to evolve.

For a practical example of applying PEFT techniques with RLHF, we encourage readers to check out the Hugging Face blog post, Fine-tuning 20B LLMs with RLHF on a 24GB consumer GPU at https://huggingface.co/blog/trl-peft, which can be especially useful if you have limited computational resources–basically, if you don't have a cluster of GPUs just available to you to fine-tune your models.

10.6 Conclusions

In this chapter, we highlighted the importance of model scaling for improving performance, not only in a wide range of downstream tasks but also in unlocking emergent abilities. However, as demonstrated, scaling models come with trade-offs, such as increased latency and higher deployment costs. These challenges necessitate modern solutions for efficient production deployment, such as knowledge distillation and model quantization.

We showcased the effectiveness of these techniques using a real-world case study with FinBERT, a BERT-based model specialized for sentiment analysis in financial data. Through distillation and quantization, we achieved a nearly threefold increase in inference speed while maintaining the same level of accuracy.

Additionally, we explored techniques for customizing your own LLM to make it more tailored to your specific data. Among these, we delved into LoRA and QLoRA, which, at the time of writing, stand out as some of the most efficient methods for fine-tuning LLMs. We also briefly mentioned the two major techniques currently used to align LLM outputs with human preferences, which have been crucial in the development of widely adopted LLM solutions like OpenAI's ChatGPT. These techniques are versatile and can be applied across various models, offering powerful solutions for optimizing performance and inference costs in real-world applications.