CHAPTER 7
Finetuning

Finetuning is the process of adapting a model to a specific task by further training the whole model or part of the model. Chapters 5 and 6 discuss prompt-based methods, which adapt a model by giving it instructions, context, and tools. Finetuning adapts a model by adjusting its weights.

Finetuning can enhance various aspects of a model. It can improve the model's domain-specific capabilities, such as coding or medical question answering, and can also strengthen its safety. However, it is most often used to improve the model's instruction-following ability, particularly to ensure it adheres to specific output styles and formats.

While finetuning can help create models that are more customized to your needs, it also requires more up-front investment. A question I hear very often is when to finetune and when to do RAG. After an overview of finetuning, this chapter will discuss the reasons for finetuning and the reasons for not finetuning, as well as a simple framework for thinking about choosing between finetuning and alternate methods.

Compared to prompt-based methods, finetuning incurs a much higher memory footprint. At the scale of today's foundation models, naive finetuning often requires more memory than what's available on a single GPU. This makes finetuning expensive and challenging to do. As discussed throughout this chapter, reducing memory requirements is a primary motivation for many finetuning techniques. This chapter dedicates one section to outlining factors contributing to a model's memory footprint, which is important for understanding these techniques.

A memory-efficient approach that has become dominant in the finetuning space is PEFT (parameter-efficient finetuning). This chapter explores PEFT and how it differs from traditional finetuning; this chapter also provides an overview of its evolving

techniques. I'll focus particularly on one compelling category: adapter-based techniques.

With prompt-based methods, knowledge about how ML models operate under the hood is recommended but not strictly necessary. However, finetuning brings you to the realm of model training, where ML knowledge is required. ML basics are beyond the scope of this book. If you want a quick refresh, the book's GitHub repository has pointers to helpful resources. In this chapter, I'll cover a few core concepts immediately relevant to the discussion.

This chapter is the most technically challenging one for me to write, not because of the complexity of the concepts, but because of the broad scope these concepts cover. I suspect it might also be technically challenging to read. If, at any point, you feel like you're diving too deep into details that aren't relevant to your work, feel free to skip.

There's a lot to discuss. Let's dive in!

Finetuning Overview

To finetune, you start with a base model that has some, but not all, of the capabilities you need. The goal of finetuning is to get this model to perform well enough for your specific task.

Finetuning is one way to do *transfer learning*, a concept first introduced by Bozinovski and Fulgosi in 1976. Transfer learning focuses on how to transfer the knowledge gained from one task to accelerate learning for a new, related task. This is conceptually similar to how humans transfer skills: for example, knowing how to play the piano can make it easier to learn another musical instrument.

An early large-scale success in transfer learning was Google's multilingual translation system (Johnson et. al, 2016). The model transferred its knowledge of Portuguese–English and English–Spanish translation to directly translate Portuguese to Spanish, even though there were no Portuguese–Spanish examples in the training data.

Since the early days of deep learning, transfer learning has offered a solution for tasks with limited or expensive training data. By training a base model on tasks with abundant data, you can then transfer that knowledge to a target task.

For LLMs, knowledge gained from pre-training on text completion (a task with abundant data) is transferred to more specialized tasks, like legal question answering or text-to-SQL, which often have less available data. This capability for transfer learning makes foundation models particularly valuable.

Transfer learning improves *sample efficiency*, allowing a model to learn the same behavior with fewer examples. A *sample-efficient* model learns effectively from fewer samples. For example, while training a model from scratch for legal question answering may need millions of examples, finetuning a good base model might only require a few hundred.

Ideally, much of what the model needs to learn is already present in the base model, and finetuning just refines the model's behavior. OpenAI's InstructGPT paper (2022) suggested viewing finetuning as unlocking the capabilities a model already has but that are difficult for users to access via prompting alone.

Finetuning isn't the only way to do transfer learning. Another approach is *feature-based transfer*. In this approach, a model is trained to extract features from the data, usually as embedding vectors, which are then used by another model. I mention feature-based transfer briefly in Chapter 2, when discussing how part of a foundation model can be reused for a classification task by *adding a classifier head*.

Feature-based transfer is very common in computer vision. For instance, in the second half of the 2010s, many people used models trained on the ImagetNet dataset to extract features from images and use these features in other computer vision tasks such as object detection or image segmentation.

Finetuning is part of a model's training process. It's an extension of model pre-training. Because any training that happens after pre-training is finetuning, finetuning can take many different forms. Chapter 2 already discussed two types of finetuning: supervised finetuning and preference finetuning. Let's do a quick recap of these methods and how you might leverage them as an application developer.

Recall that a model's training process starts with *pre-training*, which is usually done with self-supervision. Self-supervision allows the model to learn from a large amount of unlabeled data. For language models, self-supervised data is typically just *sequences of text* that don't need annotations.

Before finetuning this pre-trained model with expensive task-specific data, you can finetune it with self-supervision using cheap task-related data. For example, to finetune a model for legal question answering, before finetuning it on expensive annotated (question, answer) data, you can finetune it on raw legal documents. Similarly, to finetune a model to do book summarization in Vietnamese, you can first finetune it on a large collection of Vietnamese text. *Self-supervised finetuning* is also called *continued pre-training*.

As discussed in Chapter 1, language models can be autoregressive or masked. An autoregressive model predicts the next token in a sequence using the previous tokens as the context. A masked model fills in the blank using the tokens both before and after it. Similarly, with supervised finetuning, you can also finetune a model to predict the next token or fill in the blank. The latter, also known as *infilling finetuning*, is especially useful for tasks such as text editing and code debugging. You can finetune a model for infilling even if it was pre-trained autoregressively.

The massive amount of data a model can learn from during self-supervised learning outfits the model with a rich understanding of the world, but it might be hard for users to extract that knowledge for their tasks, or the way the model behaves might be misaligned with human preference. Supervised finetuning uses high-quality annotated data to refine the model to align with human usage and preference.

During *supervised finetuning*, the model is trained using (input, output) pairs: the input can be an instruction and the output can be a response. A response can be open-ended, such as for the task of book summarization. A response can be also close-ended, such as for a classification task. High-quality instruction data can be challenging and expensive to create, especially for instructions that require factual consistency, domain expertise, or political correctness. Chapter 8 discusses how to acquire instruction data.

A model can also be finetuned with reinforcement learning to generate responses that maximize human preference. Preference finetuning requires comparative data that typically follows the format (instruction, winning response, losing response).

It's possible to finetune a model to extend its context length. *Long-context finetuning* typically requires modifying the model's architecture, such as adjusting the positional embeddings. A long sequence means more possible positions for tokens, and positional embeddings should be able to handle them. Compared to other finetuning techniques, long-context finetuning is harder to do. The resulting model might also degrade on shorter sequences.

Figure 7-1 shows the making of different Code Llama models (Rozière et al., 2024), from the base model Llama 2, using different finetuning techniques. Using long-context finetuning, they were able to increase the model's maximum context length from 4,096 tokens to 16,384 tokens to accommodate longer code files. In the image, instruction finetuning refers to supervised finetuning.

Finetuning can be done by both model developers and application developers. Model developers typically post-train a model with different finetuning techniques before releasing it. A model developer might also release different model versions, each finetuned to a different extent, so that application developers can choose the version that works best for them.

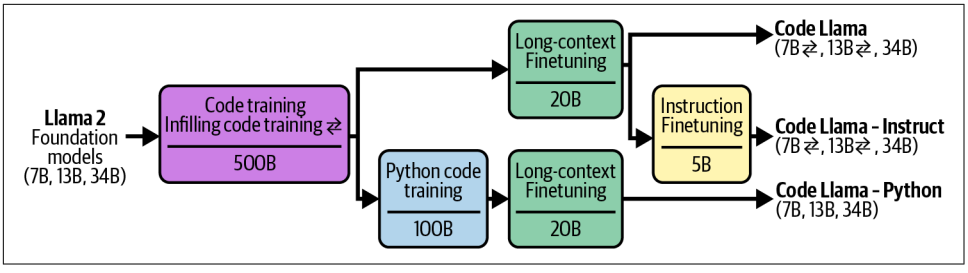

Figure 7-1. Different finetuning techniques used to make different Code Llama models. Image from the Rozière et al. (2024). Adapted from an original image licensed under CC BY 4.0.

As an application developer, you might finetune a pre-trained model, but most likely, you'll finetune a model that has been post-trained. The more refined a model is and the more relevant its knowledge is to your task, the less work you'll have to do to adapt it.

When to Finetune

Before jumping into different finetuning techniques, it's necessary to consider whether finetuning is the right option for you. Compared to prompt-based methods, finetuning requires significantly more resources, not just in data and hardware, but also in ML talent. Therefore, finetuning is generally attempted *after* extensive experiments with prompt-based methods. However, finetuning and prompting aren't mutually exclusive. Real-world problems often require both approaches.

Reasons to Finetune

The primary reason for finetuning is to improve a model's quality, in terms of both general capabilities and task-specific capabilities. Finetuning is commonly used to improve a model's ability to generate outputs following specific structures, such as JSON or YAML formats.

A general-purpose model that performs well on a wide range of benchmarks might not perform well on your specific task. If the model you want to use wasn't sufficiently trained on your task, finetuning it with your data can be especially useful.

For example, an out-of-the-box model might be good at converting from text to the standard SQL dialect but might fail with a less common SQL dialect. In this case, finetuning this model on data containing this SQL dialect will help. Similarly, if the model works well on standard SQL for common queries but often fails for customer-specific queries, finetuning the model on customer-specific queries might help.

One especially interesting use case of finetuning is bias mitigation. The idea is that if the base model perpetuates certain biases from its training data, exposing it to carefully curated data during finetuning can counteract these biases (Wang and Russakovsky, 2023). For example, if a model consistently assigns CEOs male-sounding names, finetuning it on a dataset with many female CEOs can mitigate this bias. Garimella et al. (2022) found that finetuning BERT-like language models on text authored by women can reduce these models' gender biases, while finetuning them on texts by African authors can reduce racial biases.

You can finetune a big model to make it even better, but finetuning smaller models is much more common. Smaller models require less memory, and, therefore, are easier to finetune. They are also cheaper and faster to use in production.

A common approach is to finetune a small model to imitate the behavior of a larger model using data generated by this large model. Because this approach distills the larger model's knowledge into the smaller model, it's called *distillation*. This is discussed in Chapter 8 together with other data synthesis techniques.

A small model, finetuned on a specific task, might outperform a much larger out-of-the-box model on that task. For example, Grammarly found that their finetuned Flan-T5 models (Chung et al., 2022) outperformed a GPT-3 variant specialized in text editing across a wide range of writing assistant tasks despite being 60 times smaller. The finetuning process used only 82,000 (instruction, output) pairs, which is smaller than the data typically needed to train a text-editing model from scratch.

In the early days of foundation models, when the strongest models were commercial with limited finetuning access, there weren't many competitive models available for finetuning. However, as the open source community proliferates with high-quality models of all sizes, tailored for a wide variety of domains, finetuning has become a lot more viable and attractive.

Reasons Not to Finetune

While finetuning can improve a model in many ways, many of these improvements can also be achieved, to a certain extent, without finetuning. Finetuning can improve a model's performance, but so do carefully crafted prompts and context. Finetuning can help with structured outputs, but many other techniques, as discussed in Chapter 2, can also do that.

First, while finetuning a model for a specific task can improve its performance for that task, it can degrade its performance for other tasks.[1] This can be frustrating when you intend this model for an application that expects diverse prompts.

1 Some people call this phenomenon an alignment tax (Bai et al., 2020), but this term can be confused with penalties against human preference alignment.

Imagine you need a model for three types of queries: product recommendations, changing orders, and general feedback. Originally, the model works well for product recommendations and general feedback but poorly for changing orders. To fix this, you finetune the model on a dataset of (query, response) pairs about changing orders. The finetuned model might indeed perform better for this type of query, but worse for the two other tasks.

What do you do in this situation? You can finetune the model on all the queries you care about, not just changing orders. If you can't seem to get a model to perform well on all your tasks, consider using separate models for different tasks. If you wish to combine these separate models into one to make serving them easier, you can also consider merging them together, as discussed later in this chapter.

If you're just starting to experiment with a project, finetuning is rarely the first thing you should attempt. Finetuning requires high up-front investments and continual maintenance. First, you need data. Annotated data can be slow and expensive to acquire manually, especially for tasks that demand critical thinking and domain expertise. Open source data and AI-generated data can mitigate the cost, but their effectiveness is highly variable.

Second, finetuning requires the knowledge of how to train models. You need to evaluate base models to choose one to finetune. Depending on your needs and resources, options might be limited. While finetuning frameworks and APIs can automate many steps in the actual finetuning process, you still need to understand the different training knobs you can tweak, monitor the learning process, and debug when something is wrong. For example, you need to understand how an optimizer works, what learning rate to use, how much training data is needed, how to address overfitting/underfitting, and how to evaluate your models throughout the process.

Third, once you have a finetuned model, you'll need to figure out how to serve it. Will you host it yourself or use an API service? As discussed in Chapter 9, inference optimization for large models, especially LLMs, isn't trivial. Finetuning requires less of a technical leap if you're already hosting your models in-house and familiar with how to operate models.

More importantly, you need to establish a policy and budget for monitoring, maintaining, and updating your model. As you iterate on your finetuned model, new base models are being developed at a rapid pace. These base models may improve faster than you can enhance your finetuned model. If a new base model outperforms your finetuned model on your specific task, how significant does the performance improvement have to be before you switch to the new base model? What if a new base model doesn't immediately outperform your existing model but has the potential to do so after finetuning—would you experiment with it?

In many cases, switching to a better model would provide only a small incremental improvement, and your task might be given a lower priority than projects with larger returns, like enabling new use cases.[2]

AI engineering experiments should start with prompting, following the best practices discussed in Chapter 6. Explore more advanced solutions only if prompting alone proves inadequate. Ensure you have thoroughly tested various prompts, as a model's performance can vary greatly with different prompts.

Many practitioners I've spoken with share a similar story that goes like this. Someone complains that prompting is ineffective and insists on finetuning. Upon investigation, it turns out that prompt experiments were minimal and unsystematic. Instructions were unclear, examples didn't represent actual data, and metrics were poorly defined. After refining the prompt experiment process, the prompt quality improved enough to be sufficient for their application.[3]

Finetuning Domain-Specific Tasks

Beware of the argument that general-purpose models don't work well for domain-specific tasks, and, therefore, you must finetune or train models for your specific tasks. As general-purpose models become more capable, they also become better at domain-specific tasks and can outperform the domain-specific models.

An interesting early specialized model is BloombergGPT, which was introduced by Bloomberg in March 2023. The strongest models on the market then were all proprietary, and Bloomberg wanted a mid-size model that performed well on financial tasks and could be hosted in-house for use cases with sensitive data. The model, with 50 billion parameters, required 1.3 million A100 GPU hours for training. The estimated cost of the compute was between $1.3 million and $2.6 million, excluding data costs (Wu et al., 2023).

In the same month, OpenAI released GPT-4-0314.[4] Research by Li et al. (2023) demonstrated that GPT-4-0314 significantly outperformed BloombergGPT across various financial benchmarks. Table 7-1 provides details of two such benchmarks.

2 Many businesses resist changing technologies they consider "good enough." If all companies were quick to adopt more optimal solutions, fax machines would have become obsolete by now.

3 I've also noticed a few cases when engineers know that finetuning isn't strictly necessary but still insist on doing it because they want to learn how to finetune. As an engineer who likes learning new skills, I appreciate this mindset. However, if you're in a leadership position, it can be hard to differentiate whether finetuning is needed or wanted.

4 0314 denotes the date this GPT-4 version came out, March 14, 2024. The specific date stamp matters because different versions vary significantly in performance.

Table 7-1. General-purpose models like GPT-4 can outperform financial models in financial domains.

Model	FiQA sentiment analysis (weighted F1)	ConvFinQA (accuracy)
GPT-4-0314 (zero-shot)	87.15	76.48
BloombergGPT	75.07	43.41

Since then, several mid-size models with performance comparable to GPT-4 have been released, including Claude 3.5 Sonnet (70B parameters), Llama 3-70B-Instruct, and Qwen2-72B-Instruct. The latter two are open weight and can be self-hosted.

Because benchmarks are insufficient to capture real-world performance, it's possible that BloombergGPT works well for Bloomberg for their specific use cases. The Bloomberg team certainly gained invaluable experience through training this model, which might enable them to better develop and operate future models.

Both finetuning and prompting experiments require systematic processes. Doing prompt experiments enables developers to build an evaluation pipeline, data annotation guideline, and experiment tracking practices that will be stepping stones for finetuning.

One benefit of finetuning, before prompt caching was introduced, was that it can help optimize token usage. The more examples you add to a prompt, the more input tokens the model will use, which increases both latency and cost. Instead of including your examples in each prompt, you can finetune a model on these examples. This allows you to use shorter prompts with the finetuned model, as shown in Figure 7-2.

With prompt caching, where repetitive prompt segments can be cached for reuse, this is no longer a strong benefit. Prompt caching is discussed further in Chapter 9. However, the number of examples you can use with a prompt is still limited by the maximum context length. With finetuning, there's no limit to how many examples you can use.

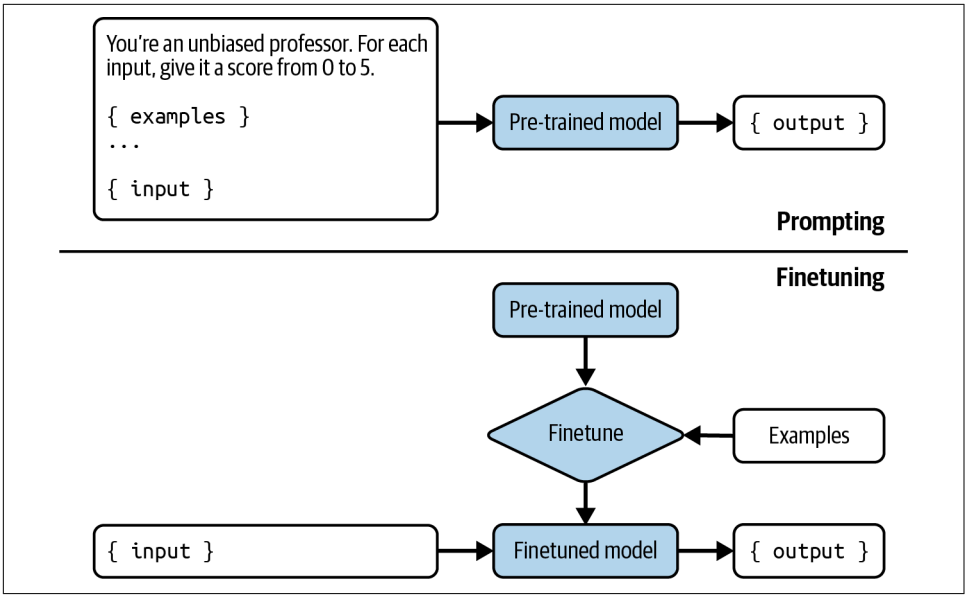

Figure 7-2. Instead of including examples in each prompt, which increases cost and latency, you finetune a model on these examples.

Finetuning and RAG

Once you've maximized the performance gains from prompting, you might wonder whether to do RAG or finetuning next. The answer depends on whether your model's failures are information-based or behavior-based.

If the model fails because it lacks information, a RAG system that gives the model access to the relevant sources of information can help. Information-based failures happen when the outputs are factually wrong or outdated. Here are two example scenarios in which information-based failures happen:

The model doesn't have the information.
: Public models are unlikely to have information private to you or your organization. When a model doesn't have the information, it either tells you so or hallucinates an answer.

The model has outdated information.
: If you ask: "How many studio albums has Taylor Swift released?" and the correct answer is 11, but the model answers 10, it can be because the model's cut-off date was before the release of the latest album.

The paper "Fine-Tuning or Retrieval?" by Ovadia et al. (2024) demonstrated that for tasks that require up-to-date information, such as questions about current events, RAG outperformed finetuned models. Not only that, RAG with the base model

outperformed RAG with finetuned models, as shown in Table 7-2. This finding indicates that *while finetuning can enhance a model's performance on a specific task, it may also lead to a decline in performance in other areas.*

Table 7-2. RAG outperforms finetuning on a question-answering task about current events, curated by Ovadia et al. (2024). FT-reg and FT-par refer to two different finetuning approaches the author used.

	Base model	Base model + RAG	FT-reg	FT-par	FT-reg + RAG	FT-par + RAG
Mistral-7B	0.481	0.875	0.504	0.588	0.810	0.830
Llama 2-7B	0.353	0.585	0.219	0.392	0.326	0.520
Orca 2-7B	0.456	0.876	0.511	0.566	0.820	0.826

On the other hand, *if the model has behavioral issues, finetuning might help*. One behavioral issue is when the model's outputs are factually correct but irrelevant to the task. For example, you ask the model to generate technical specifications for a software project to provide to your engineering teams. While accurate, the generated specs lack the details your teams need. Finetuning the model with well-defined technical specifications can make the outputs more relevant.

Another issue is when it fails to follow the expected output format. For example, if you asked the model to write HTML code, but the generated code didn't compile, it might be because the model wasn't sufficiently exposed to HTML in its training data. You can correct this by exposing the model to more HTML code during finetuning.

Semantic parsing is a category of tasks whose success hinges on the model's ability to generate outputs in the expected format and, therefore, often requires finetuning. Semantic parsing is discussed briefly in Chapters 2 and 6. As a reminder, semantic parsing means converting natural language into a structured format like JSON. Strong off-the-shelf models are generally good for common, less complex syntaxes like JSON, YAML, and regex. However, they might not be as good for syntaxes with fewer available examples on the internet, such as a domain-specific language for a less popular tool or a complex syntax.

In short, finetuning is for form, and RAG is for facts. A RAG system gives your model external knowledge to construct more accurate and informative answers. A RAG system can help mitigate your model's hallucinations. Finetuning, on the other hand, helps your model understand and follow syntaxes and styles.[5] While finetuning can potentially reduce hallucinations if done with enough high-quality data, it can also worsen hallucinations if the data quality is low.

5 Some people, such as the authors of the Llama 3.1 paper (Dubey et al., 2024), adhere to "the principle that post-training should align the model to 'know what it knows' rather than add knowledge."

If your model has both information and behavior issues, start with RAG. RAG is typically easier since you won't have to worry about curating training data or hosting the finetuned models. When doing RAG, start with simple term-based solutions such as BM25 instead of jumping straight into something that requires vector databases.

RAG can also introduce a more significant performance boost than finetuning. Ovadia et al. (2024) showed that for almost all question categories in the MMLU benchmark, RAG outperforms finetuning for three different models: Mistral 7B, Llama 2-7B, and Orca 2-7B.

However, RAG and finetuning aren't mutually exclusive. They can sometimes be used together to maximize your application's performance. In the same experiment, Ovadia et al. (2024) showed that incorporating RAG on top of a finetuned model can boost its performance on the MMLU benchmark 43% of the time. It's important to note that in this experiment, using RAG with finetuned models doesn't improve the performance 57% of the time, compared to using RAG alone.

There's no universal workflow for all applications. Figure 7-3 shows some paths an application development process might follow over time. The arrow indicates what next step you might try. This figure is inspired by an example workflow shown by OpenAI (2023).

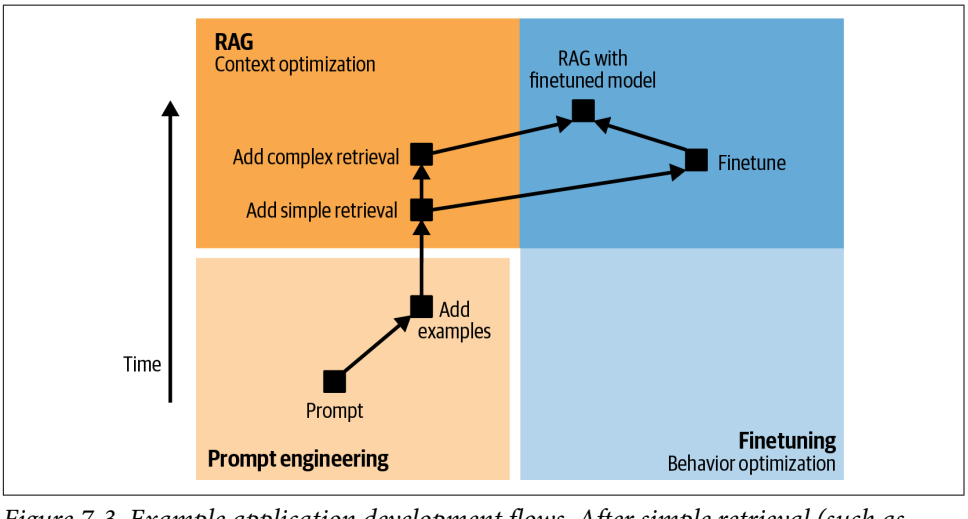

Figure 7-3. Example application development flows. After simple retrieval (such as term-based retrieval), whether to experiment with more complex retrieval (such as hybrid search) or finetuning depends on each application and its failure modes.

So the workflow to adapt a model to a task might work as follows. Note that before any of the adaptation steps, you should define your evaluation criteria and design your evaluation pipeline, as discussed in Chapter 4. This evaluation pipeline is what you'll use to benchmark your progress as you develop your application. Evaluation

doesn't happen only in the beginning. It should be present during every step of the process:

1. Try to get a model to perform your task with prompting alone. Use the prompt engineering best practices covered in Chapter 5, including systematically versioning your prompts.
2. Add more examples to the prompt. Depending on the use case, the number of examples needed might be between 1 and 50.
3. If your model frequently fails due to missing information, connect it to data sources that can supply relevant information. When starting with RAG, begin by using basic retrieval methods like term-based search. Even with simple retrieval, adding relevant and accurate knowledge should lead to some improvement in your model's performance.
4. Depending on your model's failure modes, you might explore one of these next steps:
    a. If the model continues having information-based failures, you might want to try even more advanced RAG methods, such as embedding-based retrieval.
    b. If the model continues having behavioral issues, such as it keeps generating irrelevant, malformatted, or unsafe responses, you can opt for finetuning. Embedding-based retrieval increases inference complexity by introducing additional components into the pipeline, while finetuning increases the complexity of model development but leaves inference unchanged.
5. Combine both RAG and finetuning for even more performance boost.

If, after considering all the pros and cons of finetuning and other alternate techniques, you decide to finetune your model, the rest of the chapter is for you. First, let's look into the number one challenge of finetuning: its memory bottleneck.

Memory Bottlenecks

Because finetuning is memory-intensive, many finetuning techniques aim to minimize their memory footprint. Understanding what causes this memory bottleneck is necessary to understand why and how these techniques work. This understanding, in turn, can help you select a finetuning method that works best for you.

Besides explaining finetuning's memory bottleneck, this section also introduces formulas for back-of-the-napkin calculation of the memory usage of each model. This calculation is useful in estimating what hardware you'd need to serve or finetune a model.

Because memory calculation requires a breakdown of low-level ML and computing concepts, this section is technically dense. If you're already familiar with these concepts, feel free to skip them.

Key Takeaways for Understanding Memory Bottlenecks

If you decide to skip this section, here are a few key takeaways. If you find any of these takeaways unfamiliar, the concepts in this section should help explain it:

1. Because of the scale of foundation models, memory is a bottleneck for working with them, both for inference and for finetuning. The memory needed for finetuning is typically much higher than the memory needed for inference because of the way neural networks are trained.
2. The key contributors to a model's memory footprint during finetuning are its number of parameters, its number of trainable parameters, and its numerical representations.
3. The more trainable parameters, the higher the memory footprint. You can reduce memory requirement for finetuning by reducing the number of trainable parameters. Reducing the number of trainable parameters is the motivation for PEFT, parameter-efficient finetuning.
4. Quantization refers to the practice of converting a model from a format with more bits to a format with fewer bits. Quantization is a straightforward and efficient way to reduce a model's memory footprint. For a model of 13 billion parameters, using FP32 means 4 bytes per weight or 52 GB for the whole weights. If you can reduce each value to only 2 bytes, the memory needed for the model's weights decreases to 26 GB.
5. Inference is typically done using as few bits as possible, such as 16 bits, 8 bits, and even 4 bits.
6. Training is more sensitive to numerical precision, so it's harder to train a model in lower precision. Training is typically done in mixed precision, with some operations done in higher precision (e.g., 32-bit) and some in lower precision (e.g., 16-bit or 8-bit).

Backpropagation and Trainable Parameters

A key factor that determines a model's memory footprint during finetuning is its number of *trainable parameters*. A trainable parameter is a parameter that can be updated during finetuning. During pre-training, all model parameters are updated. During inference, no model parameters are updated. During finetuning, some or all model parameters may be updated. The parameters that are kept unchanged are *frozen parameters*.

The memory needed for each trainable parameter results from the way a model is trained. As of this writing, neural networks are typically trained using a mechanism called *backpropagation*.[6] With backpropagation, each training step consists of two phases:

1. Forward pass: the process of computing the output from the input.
2. Backward pass: the process of updating the model's weights using the aggregated signals from the forward pass.

During inference, only the forward pass is executed. During training, both passes are executed. At a high level, the backward pass works as follows:

1. Compare the computed output from the forward pass against the expected output (ground truth). If they are different, the model made a mistake, and the parameters need to be adjusted. The difference between the computed output and the expected output is called the *loss*.
2. Compute how much each trainable parameter contributes to the mistake. This value is called the *gradient*. Mathematically, gradients are computed by taking the derivative of the loss with respect to each trainable parameter. There's one gradient value per trainable parameter.[7] If a parameter has a high gradient, it significantly contributes to the loss and should be adjusted more.
3. Adjust trainable parameter values using their corresponding gradient. How much each parameter should be readjusted, given its gradient value, is determined by the *optimizer*. Common optimizers include SGD (stochastic gradient descent) and Adam. For transformer-based models, Adam is, by far, the most widely used optimizer.

The forward and backward pass for a hypothetical neural network with three parameters and one nonlinear activation function is visualized in Figure 7-4. I use this dummy neural network to simplify the visualization.

6 Other than backpropagation, a promising approach to training neural networks is evolutionary strategy. One example, described by Maheswaranathan et al., combines random search with surrogate gradients, instead of using real gradients, to update model weights. Another interesting approach is direct feedback alignment (Arild Nøkland, 2016).

7 If a parameter is not trainable, it doesn't need to be updated and, therefore, there's no need to compute its gradient.

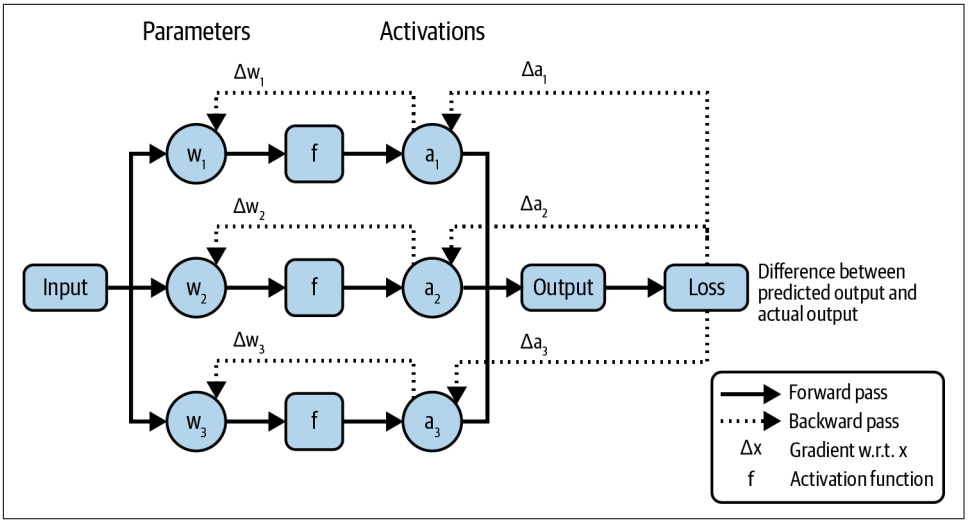

Figure 7-4. The forward and backward pass of a simple neural network.

During the backward pass, each trainable parameter comes with additional values, its gradient, and its optimizer states. Therefore, the more trainable parameters there are, the more memory is needed to store these additional values.

Memory Math

It's useful to know how much memory a model needs so that you can use the right hardware for it. Often, you might already have the hardware and need to calculate whether you can afford to run a certain model. If a model requires 30 GB of memory to do inference, a chip with 24 GB of memory won't be sufficient.

A model's memory footprint depends on the model as well as the workload and the different optimization techniques used to reduce its memory usage. Because it's impossible to account for all optimization techniques and workloads, in this section, I'll outline only the formulas for approximate calculations, which should give you a rough idea of how much memory you need to operate a model, both during inference and training.

Inference and training having distinct memory profiles is one of the reasons for the divergence in chips for training and inference, as discussed in Chapter 9.

Memory needed for inference

During inference, only the forward pass is executed. The forward pass requires memory for the model's weights. Let N be the model's parameter count and M be the memory needed for each parameter; the memory needed to load the model's parameters is:

```
N × M
```

The forward pass also requires memory for activation values. Transformer models need memory for key-value vectors for the attention mechanism. The memory for both activation values and key-value vectors grows linearly with sequence length and batch size.

For many applications, the memory for activation and key-value vectors can be assumed to be 20% of the memory for the model's weights. If your application uses a longer context or larger batch size, the actual memory needed will be higher. This assumption brings the model's memory footprint to:

```
N × M × 1.2
```

Consider a 13B-parameter model. If each parameter requires 2 bytes, the model's weights will require 13B × 2 bytes = 26 GB. The total memory for inference will be 26 GB × 1.2 = 31.2 GB.

A model's memory footprint grows rapidly with its size. As models become bigger, memory becomes a bottleneck for operating them.[8] A 70B-parameter model with 2 bytes per parameter will require a whooping 140 GB of memory just for its weights.[9]

Memory needed for training

To train a model, you need memory for the model's weights and activations, which has already been discussed. Additionally, you need memory for gradients and optimizer states, which scales with the number of trainable parameters.

Overall, the memory needed for training is calculated as:

Training memory = model weights + activations + gradients + optimizer states

8 Some might say that you're not doing AI until you've seen a "RuntimeError: CUDA out of memory" error.

9 To learn more about inference memory calculation, check out Carol Chen's "Transformer Inference Arithmetic", kipply's blog (March 2022).

During the backward pass, each trainable parameter requires one value for gradient plus zero to two values for optimizer states, depending on the optimizer:

- A vanilla SGD optimizer has no state.
- A momentum optimizer stores one value per trainable parameter.
- An Adam optimizer stores two values per trainable parameter.

Imagine you're updating all parameters in a 13B-parameter model using the Adam optimizer. Because each trainable parameter has three values for its gradient and optimizer states, if it takes two bytes to store each value, the memory needed for gradients and optimizer states will be:

```
13 billion × 3 × 2 bytes = 78 GB
```

However, if you only have 1B trainable parameters, the memory needed for gradients and optimizer states will be only:

```
1 billion × 3 × 2 bytes = 6 GB
```

One important thing to note is that in the previous formula, I assumed that the memory needed for activations is less than the memory needed for the model's weights. However, in reality, the activation memory can be much larger. If activations are stored for gradient computation, the memory needed for activations can dwarf the memory needed for the model's weights. Figure 7-5 shows the memory needed for activations compared to the memory needed for the model's weights for different Megatron models at different scales, according to the paper "Reducing Activation Recomputation in Large Transformer Models", by Korthikanti et al. (2022).

One way to reduce the memory needed for activations is not to store them. Instead of storing activations for reuse, you recompute activations when necessary. This technique is called *gradient checkpointing* or *activation recomputation*. While this reduces the memory requirements, it increases the time needed for training due to the recomputation.[10]

10 To learn more about training memory calculation, check out EleutherAI's "Transformer Math 101" (Anthony et al., April 2023).

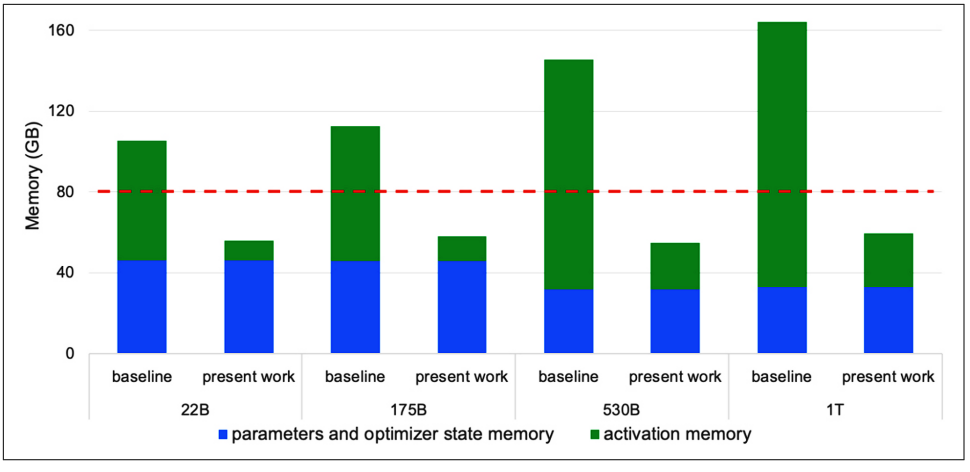

Figure 7-5. The memory needed for activations can dwarf the memory needed for the model's weights. Image from Korthikanti et al., 2022.

Numerical Representations

In the memory calculation so far, I've assumed that each value takes up two bytes of memory. The memory required to represent each value in a model contributes directly to the model's overall memory footprint. If you reduce the memory needed for each value by half, the memory needed for the model's weights is also reduced by half.

Before discussing how to reduce the memory needed for each value, it's useful to understand numerical representations. Numerical values in neural networks are traditionally represented as float numbers. The most common family of floating point formats is the FP family, which adheres to the Institute of Electrical and Electronics Engineers (IEEE) standard for Floating-Point Arithmetic (IEEE 754):

- FP32 uses 32 bits (4 bytes) to represent a float. This format is called single precision.
- FP64 uses 64 bits (8 bytes) and is called double precision.
- FP16 uses 16 bits (2 bytes) and is called half precision.

While FP64 is still used in many computations—as of this writing, FP64 is the default format for NumPy and pandas—it's rarely used in neural networks because of its memory footprint. FP32 and FP16 are more common. Other popular floating point formats in AI workloads include *BF16* (BFloat16) and *TF32* (TensorFloat-32). BF16 was designed by Google to optimize AI performance on TPUs and TF32 was designed by NVIDIA for GPUs.[11]

Numbers can also be represented as integers. Even though not yet as common as floating formats, integer representations are becoming increasingly popular. Common integer formats are INT8 (8-bit integers) and INT4 (4-bit integers).[12]

Each float format usually has 1 bit to represent the number's sign, i.e., negative or positive. The rest of the bits are split between *range* and *precision*:[13]

Range
: The number of range bits determines the range of values the format can represent. More bits means a wider range. This is similar to how having more digits lets you represent a wider range of numbers.

Precision
: The number of precision bits determines how precisely a number can be represented. Reducing the number of precision bits makes a number less precise. For example, if you convert 10.1234 to a format that can support only two decimal digits, this value becomes 10.12, which is less precise than the original value.

Figure 7-6 shows different floating point formats along with their range and precision bits.[14]

11 Google introduced BFloat16 as "the secret to high performance on Cloud TPUs".

12 Integer formats are also called *fixed point* formats.

13 Range bits are called *exponents*. Precision bits are called significands.

14 Note that usually the number at the end of a format's name signifies how many bits it occupies, but TF32 actually has 19 bits, not 32 bits. I believe it was named so to suggest its functional compatibility with FP32. But honestly, why it's called TF32 and not TF19 keeps me up at night. An ex-coworker at NVIDIA volunteered his conjecture that people might be skeptical of weird formats (19-bit), so naming this format TF32 makes it look more friendly.

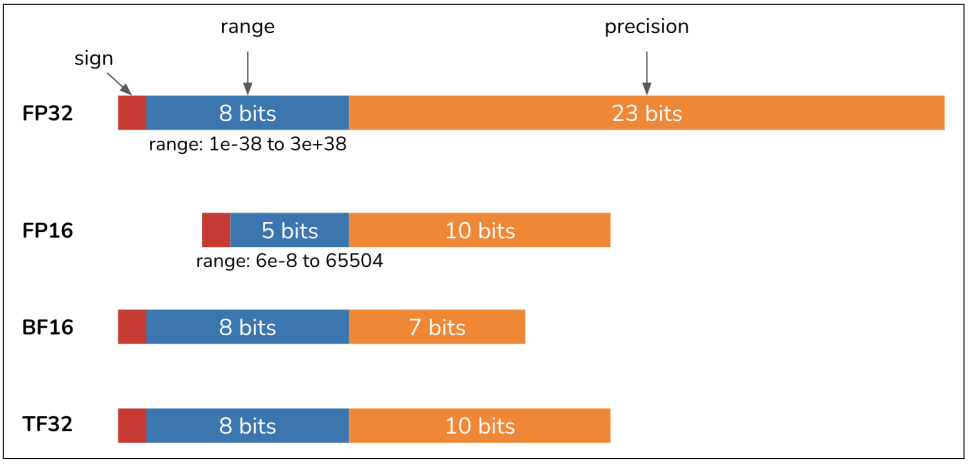

Figure 7-6. Different numerical formats with their range and precision.

Formats with more bits are considered *higher precision*. Converting a number with a high-precision format into a low-precision format (e.g., from FP32 to FP16) means *reducing its precision*. Reducing precision can cause a value to change or result in errors. Table 7-3 shows how FP32 values can be converted into FP16, BF16, and TF32.

Table 7-3. Convert from FP32 values to lower-precision formats. Resultant inaccuracies are in italics.

FP32	FP16	BF16	TF32
0.0123456789	*0.0123443603515625*	*0.0123291*	*0.0123443603515625*
0.123456789	*0.12347412109375*	*0.123535*	*0.1234130859375*
1.23456789	*1.234375*	*1.23438*	*1.234375*
12.3456789	*12.34375*	*12.375*	*12.34375*
123.456789	*123.4375*	*123.5*	*123.4375*
1234.56789	*1235.0*	*1232.0*	*1234.0*
12345.6789	*12344.0*	*12352.0*	*12344.0*
123456.789	*INF*[a]	*123392.0*	*123456.0*
1234567.89	*INF*	*1236990.0*	*1233920.0*

[a] Values out of bound in FP16 are rounded to infinity.

Note in Table 7-3 that even though BF16 and FP16 have the same number of bits, BF16 has more bits for range and fewer bits for precision. This allows BF16 to represent large values that are out-of-bound for FP16. However, this also makes BF16 less precise than FP16. For example, 1234.56789 is 1235.0 in FP16 (0.035% value change) but 1232.0 in BF16 (0.208% value change).

When using a model, make sure to load the model in the format it's intended for. Loading a model into the wrong numerical format can cause the model to change significantly. For example, Llama 2 had its weights set to BF16 when it came out. However, many teams loaded the model in FP16 and were subsequently frustrated to find the model's quality much worse than advertised.[15] While this misunderstanding wasted a lot of people's time, the upside is that it forced many people to learn about numerical representations.

The right format for you depends on the distribution of numerical values of your workload (such as the range of values you need), how sensitive your workload is to small numerical changes, and the underlying hardware.[16]

Quantization

The fewer bits needed to represent a model's values, the lower the model's memory footprint will be. A 10B-parameter model in a 32-bit format requires 40 GB for its weights, but the same model in a 16-bit format will require only 20 GB. Reducing precision, also known as quantization, is a cheap and extremely effective way to reduce a model's memory footprint. It's straightforward to do and generalizes over tasks and architectures. In the context of ML, low precision generally refers to any format with fewer bits than the standard FP32.

Quantization Versus Reduced Precision

Strictly speaking, it's quantization only if the target format is integer. However, in practice, quantization is used to refer to all techniques that convert values to a lower-precision format. In this book, I use quantization to refer to precision reduction, to keep it consistent with the literature.

15 The FP16 and BF16 confusion continued with Llama 3.1. See X and Threads discussions: 1; 2, 3, 4; and llama.cpp's benchmark between BF16 and FP16, Bloke's writeup, and Raschka's writeup.

16 Designing numerical formats is a fascinating discipline. Being able to create a lower-precision format that doesn't compromise a system's quality can make that system much cheaper and faster, enabling new use cases.

To do quantization, you need to decide what to quantize and when:

What to quantize
: Ideally, you want to quantize whatever is consuming most of your memory, but it also depends on what you can quantize without hurting performance too much. As discussed in "Memory Math" on page 322, major contributors to a model's memory footprint during inference are the model's weights and activations.[17] Weight quantization is more common than activation quantization, since weight activation tends to have a more stable impact on performance with less accuracy loss.

When to quantize
: Quantization can happen during training or post-training. Post-training quantization (PTQ) means quantizing a model after it's been fully trained. PTQ is by far the most common. It's also more relevant to AI application developers who don't usually train models.

Inference quantization

In the early days of deep learning, it was standard to train and serve models using 32 bits with FP32. Since the late 2010s, it has become increasingly common to serve models in 16 bits and in even lower precision. For example, Dettmers et al. (2022) have done excellent work quantizing LLMs into 8 bits with LLM.int8() and 4 bits with QLoRA (Dettmers et al., 2023).

A model can also be served in *mixed precision*, where values are reduced in precision when possible and maintained in higher precision when necessary. To serve models on the devices, Apple (2024) leveraged a quantization scheme that uses a mixture of 2-bit and 4-bit formats, averaging 3.5 bits-per-weight. Also in 2024, in anticipation of 4-bit neural networks, NVIDIA announced their new GPU architecture, Blackwell, that supports model inference in 4-bit float.

Once you get to 8 bits and under, numerical representations get more tricky. You can keep parameter values as floats using one of the minifloat formats, such as FP8 (8 bits) and FP4 (4 bits).[18] More commonly, however, parameter values are converted into an integer format, such as INT8 or INT4.

17 Another major contributor to the memory footprint of transformer-based models is the KV cache, which is discussed in Chapter 9.

18 The smallest possible float size that follows all IEEE principles is 4-bit.

Quantization is effective, but there's a limit to how far it can go. You can't have fewer than 1 bit per value, and some have attempted the 1-bit representation, e.g., BinaryConnect (Courbariaux et al., 2015), Xnor-Net (Rastegari et al., 2016), and BitNet (Wang et al., 2023).[19]

In 2024, Microsoft researchers (Ma et al.) declared that we're entering the era of 1-bit LLMs by introducing BitNet b1.58, a transformer-based language model that requires only 1.58 bits per parameter and whose performance is comparable to 16-bit Llama 2 (Touvron et al., 2023) up to 3.9B parameters, as shown in Table 7-4.

Table 7-4. BitNet b1.58's performance compared to that of Llama 2 16-bit on different benchmarks and at different model sizes, up to 3.9B parameters. Results from Ma et al. (2024).

Model	Size	ARCe	ARCc	HS	BQ	OQ	PQ	WGe	Avg.
Llama LLM	700M	54.7	23.0	37.0	60.0	20.2	68.9	54.8	45.5
BitNet b1.58	700M	51.8	21.4	35.1	58.2	20.0	68.1	55.2	44.3
Llama LLM	1.3B	56.9	23.5	38.5	59.1	21.6	70.0	53.9	46.2
BitNet b1.58	1.3B	54.9	24.2	37.7	56.7	19.6	68.8	55.8	45.4
Llama LLM	3B	62.1	25.6	43.3	61.8	24.6	72.1	58.2	49.7
BitNet b1.58	3B	61.4	28.3	42.9	61.5	26.6	71.5	59.3	50.2
BitNet b1.58	3.9B	64.2	28.7	44.2	63.5	24.2	73.2	60.5	51.2

Reduced precision not only reduces the memory footprint but also often improves computation speed. First, it allows a larger batch size, enabling the model to process more inputs in parallel. Second, reduced precision speeds up computation, which further reduces inference latency and training time. To illustrate this, consider the addition of two numbers. If we perform the addition bit by bit, and each takes *t* nanoseconds, it'll take *32t* nanoseconds for 32 bits but only *16t* nanoseconds for 16 bits. However, reducing precision doesn't always reduce latency due to the added computation needed for format conversion.

There are downsides to reduced precision. Each conversion often causes a small value change, and many small changes can cause a big performance change. If a value is outside the range the reduced precision format can represent, it might be converted to infinity or an arbitrary value, causing the model's quality to further degrade. How to reduce precision with minimal impact on model performance is an active area of research, pursued by model developers as well as by hardware makers and application developers.

19 The authors of the Xnor-Net paper spun off Xnor.ai, a startup that focused on model compression. In early 2020, it was acquired by Apple for a reported $200M.

Inference in lower precision has become a standard. A model is trained using a higher-precision format to maximize performance, then its precision is reduced for inference. Major ML frameworks, including PyTorch, TensorFlow, and Hugging Face's transformers, offer PTQ for free with a few lines of code.

Some edge devices only support quantized inference. Therefore, frameworks for on-device inference, such as TensorFlow Lite and PyTorch Mobile, also offer PTQ.

Training quantization

Quantization during training is not yet as common as PTQ, but it's gaining traction. There are two distinct goals for training quantization:

1. To produce a model that can perform well in low precision during inference. This is to address the challenge that a model's quality might degrade during post-training quantization.
2. To reduce training time and cost. Quantization reduces a model's memory footprint, allowing a model to be trained on cheaper hardware or allowing the training of a larger model on the same hardware. Quantization also speeds up computation, which further reduces costs.

A quantization technique might help achieve one or both of these goals.

Quantization-aware training (QAT) aims to create a model with high quality in low precision for inference. With QAT, the model simulates low-precision (e.g., 8-bit) behavior during training, which allows the model to learn to produce high-quality outputs in low precision. However, QAT doesn't reduce a model's training time since its computations are still performed in high precision. QAT can even increase training time due to the extra work of simulating low-precision behavior.

On the other hand, training a model directly in lower precision can help with both goals. People attempted to train models in reduced precision as early as 2016; see Hubara et al. (2016) and Jacob et al. (2017). Character.AI (2024) shared that they were able to train their models entirely in INT8, which helped eliminate the training/serving precision mismatch while also significantly improving training efficiency. However, training in lower precision is harder to do, as backpropgation is more sensitive to lower precision.[20]

Lower-precision training is often done in *mixed precision*, where a copy of the weights is kept in higher precision but other values, such as gradients and activations,

20 During training, the model's weights are updated via multiple steps. Small rounding changes can compound during the training process, making it difficult for the model to achieve the desirable performance. On top of that, loss values require precise computation. Small changes in the loss value can point parameter updates in the wrong direction.

are kept in lower precision.[21] You can also have less-sensitive weight values computed in lower precision and more-sensitive weight values computed in higher precision. For example, LLM-QAT (Liu et al., 2023) quantizes weights and activations into 4 bits but keeps embeddings in 16 bits.

The portions of the model that should be in lower precision can be set automatically using the *automatic mixed precision* (AMP) functionality offered by many ML frameworks.

It's also possible to have different phases of training in different precision levels. For example, a model can be trained in higher precision but finetuned in lower precision. This is especially common with foundation models, where the team training a model from scratch might be an organization with sufficient compute for higher precision training. Once the model is published, developers with less compute access can finetune that model in lower precision.

Finetuning Techniques

I hope that the previous section has made clear why finetuning large-scale models is so memory-intensive. The more memory finetuning requires, the fewer people who can afford to do it. Techniques that reduce a model's memory footprint make finetuning more accessible, allowing more people to adapt models to their applications. This section focuses on memory-efficient finetuning techniques, which centers around parameter-efficient finetuning.

I'll also cover model merging, an exciting but more experimental approach to creating custom models. While model merging is generally not considered finetuning, I include it in this section because it's complementary to finetuning. Finetuning tailors one model to specific needs. Model merging combines multiple models, often finetuned models, for the same purpose.

While combining multiple models isn't a new concept, new types of models and finetuning techniques have inspired many creative model-merging techniques, making this section especially fun to write about.

Parameter-Efficient Finetuning

In the early days of finetuning, models were small enough that people could finetune entire models. This approach is called *full finetuning*. In full finetuning, the number of trainable parameters is exactly the same as the number of parameters.

21 Personal anecdote: much of my team's work at NVIDIA was on mixed precision training. See "Mixed Precision Training for NLP and Speech Recognition with OpenSeq2Seq" (Huyen et al., NVIDIA Developer Technical Blog, October 2018).

Full finetuning can look similar to training. The main difference is that training starts with randomized model weights, whereas finetuning starts with model weights that have been previously trained.

As discussed in "Memory Math" on page 322, the more trainable parameters there are, the more memory is needed. Consider a 7B-parameter model:

- If you use a 16-bit format like FP16, loading the model's weights alone requires 14 GB for memory.
- Full finetuning this model with the Adam optimizer, also in a 16-bit format, requires an additional 7B × 3 × 2 bytes = 42 GB of memory.
- The total memory needed for the model's weights, gradients, and optimizer states is then 14 GB + 42 GB = 56 GB.

56 GB exceeds the memory capacity of most consumer GPUs, which typically come with 12–24 GB of memory, with higher-end GPUs offering up to 48 GB. And this memory estimation doesn't yet take into account the memory required for activations.

To fit a model on a given hardware, you can either reduce the model's memory footprint or find ways to use the hardware's memory more efficiently. Techniques like quantization and PEFT help minimize the total memory footprint. Techniques that focus on making better use of hardware memory include *CPU offloading*. Instead of trying to fit the whole model on GPUs, you can offload the excess memory onto CPUs, as demonstrated by DeepSpeed (Rasley et al., 2020).

We also haven't touched on the fact that full finetuning, especially supervised finetuning and preference finetuning, typically requires a lot of high-quality annotated data that most people can't afford. Due to the high memory and data requirements of full finetuning, people started doing *partial finetuning*. In partial finetuning, only some of the model's parameters are updated. For example, if a model has ten layers, you might freeze the first nine layers and finetune only the last layer,[22] reducing the number of trainable parameters to 10% of full finetuning.

While partial finetuning can reduce the memory footprint, it's *parameter-inefficient*. Partial finetuning requires many trainable parameters to achieve performance close to that of full finetuning. A study by Houlsby et al. (2019) shows that with BERT large (Devlin et al., 2018), you'd need to update approximately 25% of the parameters

22 In partial finetuning, it's common to finetune the layers closest to the output layer because those layers are usually more task-specific, whereas earlier layers tend to capture more general features.

to achieve performance comparable to that of full finetuning on the GLUE benchmark (Wang et al., 2018). Figure 7-7 shows the performance curve of partial finetuning with different numbers of trainable parameters.

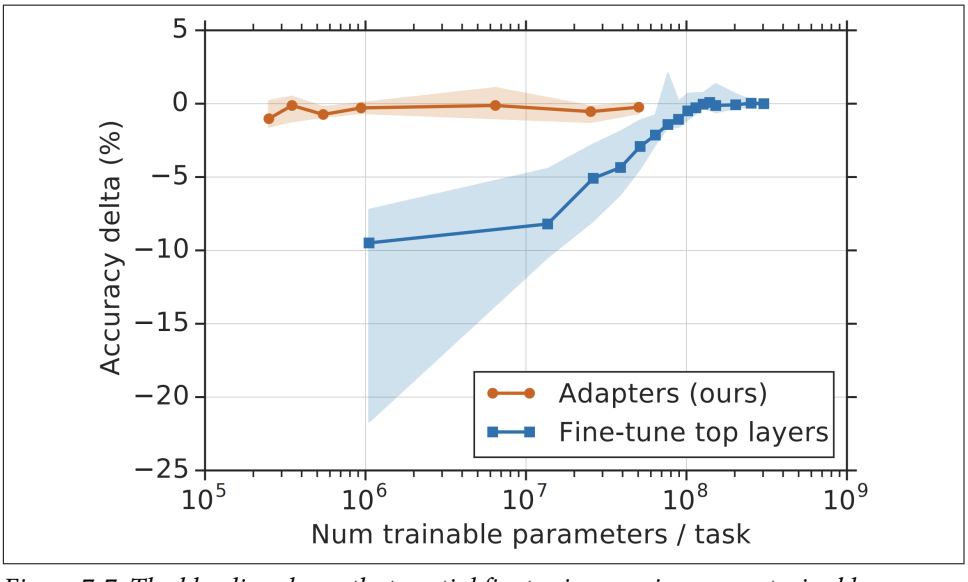

Figure 7-7. The blue line shows that partial finetuning requires many trainable parameters to achieve a performance comparable to full finetuning. Image from Houlsby et al. (2019).

This brings up the question: How to achieve performance close to that of full finetuning while using significantly fewer trainable parameters? Finetuning techniques resulting from this quest are parameter-efficient. There's no clear threshold that a finetuning method has to pass to be considered parameter-efficient. However, in general, a technique is considered parameter-efficient if it can achieve performance close to that of full finetuning while using several orders of magnitude fewer trainable parameters.

The idea of PEFT (parameter-efficient finetuning) was introduced by Houlsby et al. (2019). The authors showed that by inserting additional parameters into the model in the right places, you can achieve strong finetuning performance using a small number of trainable parameters. They inserted two adapter modules into each transformer block of a BERT model, as shown in Figure 7-8.

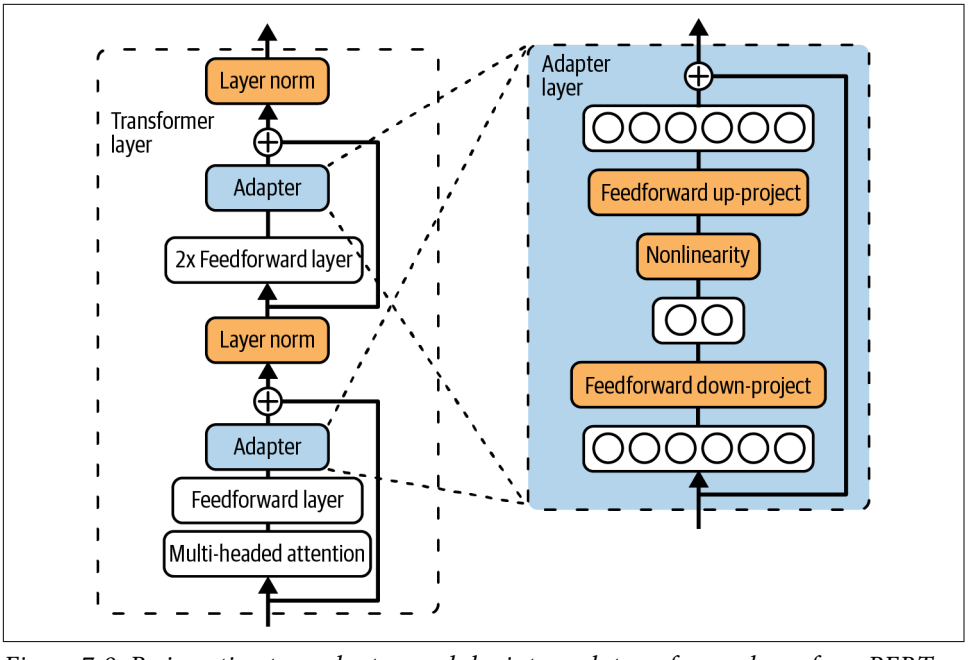

Figure 7-8. By inserting two adapter modules into each transformer layer for a BERT model and updating only the adapters, Houlsby et al. (2019) were able to achieve strong finetuning performance using a small number of trainable parameters.

During finetuning, they kept the model's original parameters unchanged and only updated the adapters. The number of trainable parameters is the number of parameters in the adapters. On the GLUE benchmark, they achieved a performance within 0.4% of full finetuning using only 3% of the number of trainable parameters. The orange line in Figure 7-7 shows the performance delta between full finetuning and finetuning using different adapter sizes.

However, the downside of this approach is that it increases the inference latency of the finetuned model. The adapters introduce additional layers, which add more computational steps to the forward pass, slowing inference.

PEFT enables finetuning on more affordable hardware, making it accessible to many more developers. PEFT methods are generally not only parameter-efficient but also sample-efficient. While full finetuning may need tens of thousands to millions of examples to achieve notable quality improvements, some PEFT methods can deliver strong performance with just a few thousand examples.

Given PEFT's obvious appeal, PEFT techniques are being rapidly developed. The next section will give an overview of these techniques before diving deeper into the most common PEFT technique: LoRA.

PEFT techniques

The existing prolific world of PEFT generally falls into two buckets: *adapter-based methods* and *soft prompt-based methods*. However, it's likely that newer buckets will be introduced in the future.

Adapter-based methods refer to all methods that involve additional modules to the model weights, such as the one developed by Houlsby et al. (2019). Because adapter-based methods involve adding parameters, they are also called *additive methods*.

As of this writing, LoRA (Hu et al., 2021) is by far the most popular adapter-based method, and it will be the topic of the following section. Other adapter-based methods include BitFit (Zaken et al., 2021), which came out around the same time LoRA did. Newer adapter methods include IA3 (Liu et al., 2022), whose efficient mixed-task batching strategy makes it particularly attractive for multi-task finetuning. It's been shown to outperform LoRA and even full finetuning in some cases. LongLoRA (Chen et al., 2023) is a LoRA variant that incorporates attention-modification techniques to expand context length.

If adapter-based methods add trainable parameters to the model's architecture, soft prompt-based methods modify how the model processes the input by introducing special trainable tokens. These additional tokens are fed into the model alongside the input tokens. They are called *soft prompts* because, like the inputs (hard prompts), soft prompts also guide the model's behaviors. However, soft prompts differ from hard prompts in two ways:

- Hard prompts are human-readable. They typically contain *discrete* tokens such as "I", "write", "a", and "lot". In contrast, soft prompts are continuous vectors, resembling embedding vectors, and are not human-readable.
- Hard prompts are static and not trainable, whereas soft prompts can be optimized through backpropagation during the tuning process, allowing them to be adjusted for specific tasks.

Some people describe soft prompting as a crossover between prompt engineering and finetuning. Figure 7-9 visualizes how you can use soft prompts together with hard prompts to guide a model's behaviors.

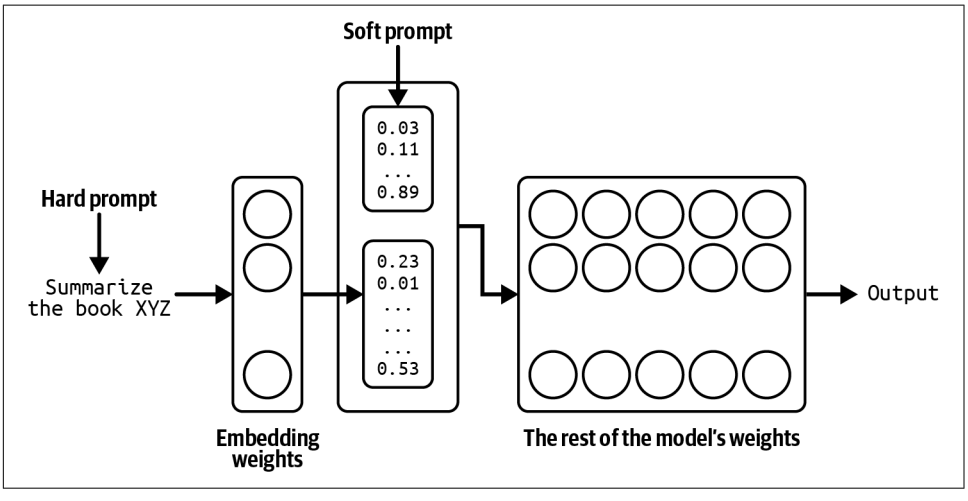

Figure 7-9. Hard prompts and soft prompts can be combined to change a model's behaviors.

Soft prompt tuning as a subfield is characterized by a series of similar-sounding techniques that can be confusing, such as prefix-tuning (Li and Liang, 2021), P-Tuning (Liu et al., 2021), and prompt tuning (Lester et al., 2021).[23] They differ mainly on the locations where the soft prompts are inserted. For example, prefix tuning prepends soft prompt tokens to the input at every transformer layer, whereas prompt tuning prepends soft prompt tokens to only the embedded input. If you want to use any of them, many PEFT frameworks will implement them out of the box for you.

To get a sense of what PEFT methods are being used, I analyzed over 1,000 open issues on the GitHub repository huggingface/peft in October 2024. The assumption is that if someone uses a technique, they are more likely to report issues or ask questions about it. Figure 7-10 shows the result. For "P-Tuning", I searched for keywords "p_tuning" and "p tuning" to account for different spellings.

23 I've never met a single person who could explain to me, on the spot, the differences between these techniques.

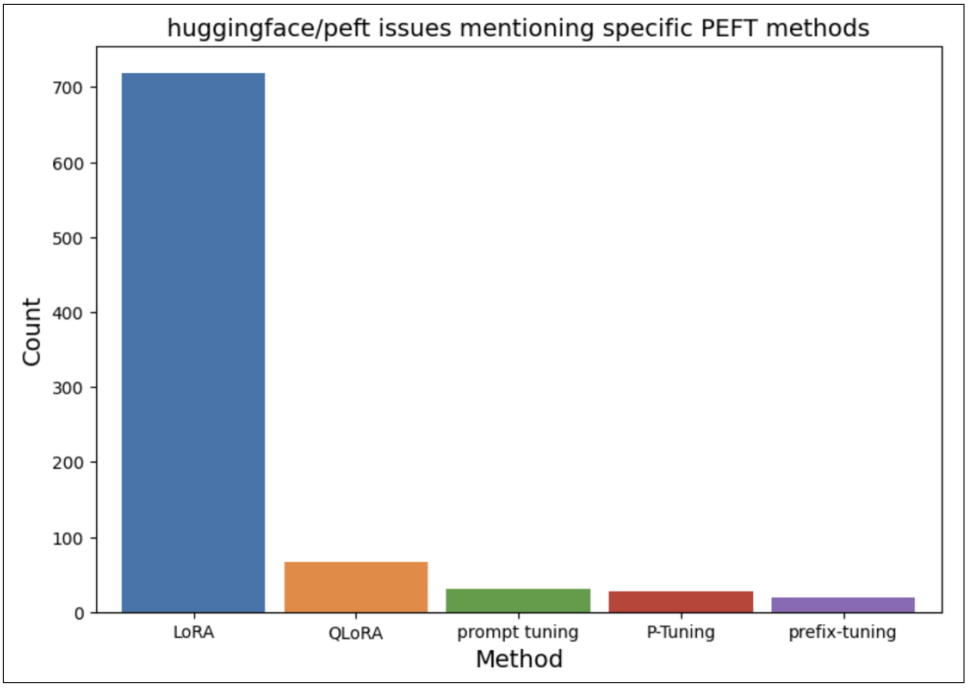

Figure 7-10. The number of issues corresponding to different finetuning techniques from the GitHub repository huggingface/peft. This is a proxy to estimate the popularity of each technique.

From this analysis, it's clear that LoRA dominates. Soft prompts are less common, but there seems to be growing interest from those who want more customization than what is afforded by prompt engineering but who don't want to invest in finetuning.

Because of LoRA's popularity, the next section focuses on how LoRA works and how it solves the challenge posed by early adapter-based methods. Even if you don't use LoRA, this deep dive should provide a framework for you to explore other finetuning methods.

LoRA

Unlike the original adapter method by Houlsby et al. (2019), LoRA (Low-Rank Adaptation) (Hu et al., 2021) incorporates additional parameters in a way that doesn't incur extra inference latency. Instead of introducing additional layers to the base model, LoRA uses modules that can be merged back to the original layers.

You can apply LoRA to individual weight matrices. Given a weight matrix, LoRA decomposes this matrix into the product of two smaller matrices, then updates these two smaller matrices before merging them back to the original matrix.

Consider the weight matrix W of the dimension $n \times m$. LoRA works as follows:

1. First, choose the dimension of the smaller matrices. Let r be the chosen value. Construct two matrices: A (dimension $n \times r$) and B (dimension $r \times m$). Their product is W_{AB}, which is of the same dimension as W. r is the LoRA *rank*.
2. Add W_{AB} to the original weight matrix W to create a new weight matrix W'. Use W' in place of W as part of the model. You can use a hyperparameter α to determine how much W_{AB} should contribute to the new matrix: $W' = W + \frac{\alpha}{r} W_{AB}$
3. During finetuning, update only the parameters in A and B. W is kept intact.

Figure 7-11 visualizes this process.

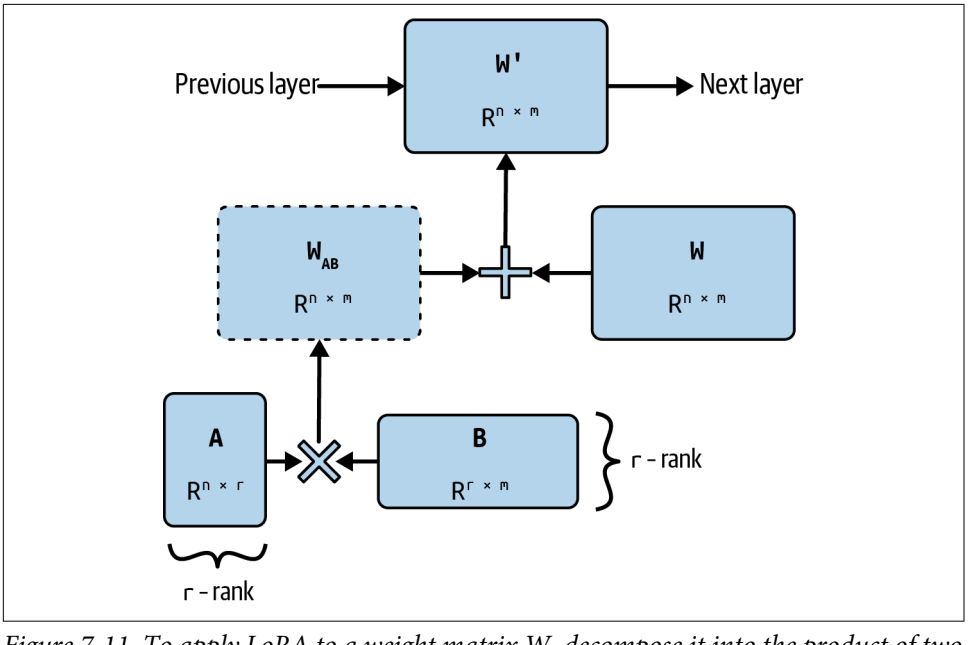

Figure 7-11. To apply LoRA to a weight matrix W, decompose it into the product of two matrices A and B. During finetuning, only A and B are updated. W is kept intact.

LoRA (Low-Rank Adaptation) is built on the concept of *low-rank factorization*, a long-standing dimensionality reduction technique. The key idea is that you can factorize a large matrix into a product of two smaller matrices to reduce the number of parameters, which, in turn, reduces both the computation and memory requirements. For example, a `9 × 9` matrix can be factorized into the product of two matrices of dimensions `9 × 1` and `1 × 9`. The original matrix has 81 parameters, but the two product matrices have only 18 parameters combined.

The number of columns in the first factorized matrix and the number of columns in the second factorized matrix correspond to the rank of the factorization. The original matrix is *full-rank*, while the two smaller matrices represent a low-rank approximation.

While factorization can significantly reduce the number of parameters, it's lossy because it only approximates the original matrix. The higher the rank, the more information from the original matrix the factorization can preserve.

Like the original adapter method, LoRA is parameter-efficient and sample-efficient. The factorization enables LoRA to use even fewer trainable parameters. The LoRA paper showed that, for GPT-3, LoRA achieves comparable or better performance with full finetuning on several tasks while using only ~4.7M trainable parameters, 0.0027% of full finetuning.

Why does LoRA work? Parameter-efficient methods like LoRA have become so popular that many people take them for granted. *But why is parameter efficiency possible at all?* If a model requires a lot of parameters to learn certain behaviors during pre-training, shouldn't it also require a lot of parameters to change its behaviors during finetuning?

The same question can be raised for data. If a model requires a lot of data to learn a behavior, shouldn't it also require a lot of data to meaningfully change this behavior? How is it possible that you need millions or billions of examples to pre-train a model, but only a few hundreds or thousands of examples to finetune it?

Many papers have argued that while LLMs have many parameters, they have very low intrinsic dimensions; see Li et al. (2018); Aghajanyan et al. (2020); and Hu et al. (2021). They showed that *pre-training implicitly minimizes the model's intrinsic dimension*. Surprisingly, larger models tend to have lower intrinsic dimensions after pre-training. This suggests that pre-training acts as a compression framework for downstream tasks. In other words, the better trained an LLM is, the easier it is to finetune the model using a small number of trainable parameters and a small amount of data.

You might wonder, if low-rank factorization works so well, *why don't we use LoRA for pre-training as well?* Instead of pre-training a large model and applying low-rank factorization only during finetuning, could we factorize a model from the start for pre-training? Low-rank pre-training can significantly reduce the model's number of parameters, significantly reducing the model's pre-training time and cost.

Throughout the 2010s, many people tried training low-rank neural networks, exemplified in studies such as "Low-Rank Matrix Factorization for Deep Neural Network Training with High-Dimensional Output Targets" (Sainath et al., 2013), "Semi-Orthogonal Low-Rank Matrix Factorization for Deep Neural Networks" (Povey et al., 2018), and "Speeding up Convolutional Neural Networks with Low Rank Expansions" (Jaderberg et al., 2014).

Low-rank factorization has proven to be effective at smaller scales. For example, by applying various factorization strategies, including replacing 3×3 convolution with 1×1 convolution, SqueezeNet (Iandola et al., 2016) achieves AlexNet-level accuracy on ImageNet using 50 times fewer parameters.

More recent attempts to train low-rank LLMs include ReLoRA (Lialin et al., 2023) and GaLore (Zhao et al., 2024). ReLoRA works for transformer-based models of up to 1.3B parameters. GaLore achieves performance comparable to that of a full-rank model at 1B parameters and promising performance at 7B parameters.

It's possible that one day not too far in the future, researchers will develop a way to scale up low-rank pre-training to hundreds of billions of parameters. However, if Aghajanyan et al.'s argument is correct—that pre-training implicitly compresses a model's intrinsic dimension—full-rank pre-training is still necessary to sufficiently reduce the model's intrinsic dimension to a point where low-rank factorization can work. It would be interesting to study exactly how much full-rank training is necessary before it's possible to switch to low-rank training.

LoRA configurations. To apply LoRA, you need to decide what weight matrices to apply LoRA to and the rank of each factorization. This section will discuss the considerations for each of these decisions.

LoRA can be applied to each individual weight matrix. The efficiency of LoRA, therefore, depends not only on what matrices LoRA is applied to but also on the model's architecture, as different architectures have different weight matrices.

While there have been examples of LoRA with other architectures, such as convolutional neural networks (Dutt et al., 2023; Zhong et al., 2024; Aleem et al., 2024), LoRA has been primarily used for transformer models.[24] LoRA is most commonly applied to the four weight matrices in the attention modules: the query (W_q), key (W_k), value (W_v), and output projection (W_o) matrices.

Typically, LoRA is applied uniformly to all matrices of the same type within a model. For example, applying LoRA to the query matrix means applying LoRA to all query matrices in the model.

Naively, you can apply LoRA to all these attention matrices. However, often, you're constrained by your hardware's memory and can accommodate only a fixed number of trainable parameters. Given a fixed budget of trainable parameters, what matrices should you apply LoRA to, to maximize performance?

When finetuning GPT-3 175B, Hu et al. (2021) set their trainable parameter budget at 18M, which is 0.01% of the model's total number of parameters. This budget allows them to apply LoRA to the following:

1. One matrix with the rank of 8
2. Two matrices with the rank of 4
3. All four matrices with the rank of 2

GPT-3 175B has 96 transformer layers with a model dimension of 12,288. Applying LoRA with rank = 2 to all four matrices would yield $(12{,}288 \times 2 \times 2) \times 4 = 196{,}608$ trainable parameters per layer, or 18,874,368 trainable parameters for the whole model.

They found that applying LoRA to all four matrices with rank = 2 yields the best performance on the WikiSQL (Zhong et al., 2017) and MultiNLI (Multi-Genre Natural Language Inference) benchmarks (Williams et al., 2017). Table 7-5 shows their results. However, the authors suggested that if you can choose only two attention matrices, the query and value matrices generally yield the best results.

24 To effectively use LoRA for a model, it's necessary to understand that model's architecture. Chapter 2 already covered the weight composition of some transformer-based models. For the exact weight composition of a model, refer to its paper.

Table 7-5. LoRA performance with the budget of 18M trainable parameters. Results from LoRA (Hu et al., 2021).

Number of trainable parameters = 18M							
Weight type	W_q	W_k	W_v	W_o	W_q, W_k	W_q, W_v	W_q, W_k, W_v, W_o
Rank r	8	8	8	8	4	4	2
WikiSQL (± 0.5%)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI (± 0.1%)	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

Empirical observations suggest that applying LoRA to more weight matrices, including the feedforward matrices, yields better results. For example, Databricks showed that the biggest performance boost they got was from applying LoRA to all feedforward layers (Sooriyarachchi, 2023). Fomenko et al. (2024) noted that feedforward-based LoRA can be complementary to attention-based LoRA, though attention-based LoRA typically offers greater efficacy within memory constraints.

The beauty of LoRA is that while its performance depends on its rank, studies have shown that *a small r, such as between 4 and 64, is usually sufficient for many use cases*. A smaller r means fewer LoRA parameters, which translates to a lower memory footprint.

The LoRA authors observed that, to their surprise, increasing the value of r doesn't increase finetuning performance. This observation is consistent with Databricks' report that "increasing r beyond a certain value may not yield any discernible increase in quality of model output" (Sooriyarachchi, 2023).[25] Some argue that a higher r might even hurt as it can lead to overfitting. However, in some cases, a higher rank might be necessary. Raschka (2023) found that $r = 256$ achieved the best performance on his tasks.

Another LoRA hyperparameter you can configure is the value α that determines how much the product W_{AB} should contribute to the new matrix during merging: $W' = W + \frac{\alpha}{r} W_{AB}$. In practice, I've often seen α chosen so that the ratio $\alpha : r$ is typically between 1:8 and 8:1, but the optimal ratio varies. For example, if r is small, you might want α to be larger, and if r is large, you might want α to be smaller. Experimentation is needed to determine the best (r, α) combination for your use case.

Serving LoRA adapters. LoRA not only lets you finetune models using less memory and data, but it also simplifies serving multiple models due to its modularity. To understand this benefit, let's examine how to serve a LoRA-finetuned model.

25 As of this writing, some finetuning frameworks like Fireworks only allow a maximum LoRA rank of 32. However, this constraint is unlikely due to performance and more likely due to their hardware's memory constraint.

In general, there are two ways to serve a LoRA-finetuned model:

1. Merge the LoRA weights A and B into the original model to create the new matrix W′ prior to serving the finetuned model. Since no extra computation is done during inference, no extra latency is added.
2. Keep W, A, and B separate during serving. The process of merging A and B back to W happens during inference, which adds extra latency.

The first option is generally better if you have only one LoRA model to serve, whereas the second is generally better for *multi-LoRA serving*—serving multiple LoRA models that share the same base model. Figure 7-12 visualizes multi-LoRA serving if you keep the LoRA adapters separate.

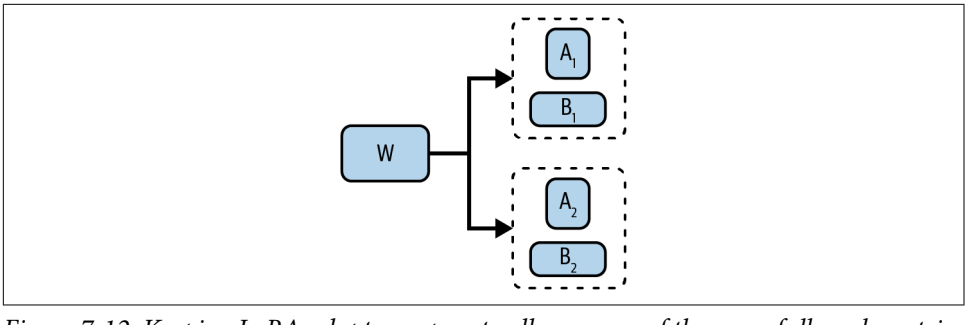

Figure 7-12. Keeping LoRA adapters separate allows reuse of the same full-rank matrix W in multi-LoRA serving.

For multi-LoRA serving, while option 2 adds latency overhead, it significantly reduces the storage needed. Consider the scenario in which you finetune a model for each of your customers using LoRA. With 100 customers, you end up with 100 finetuned models, all sharing the same base model. With option 1, you have to store 100 full-rank matrices W'. With option 2, you only have to store one full-rank matrix W, and 100 sets of smaller matrices (A, B).

To put this in perspective, let's say that the original matrix W is of the dimension `4096 × 4096` (16.8M parameters). If the LoRA's rank is 8, the number of parameters in A and B is `4096 × 8 × 2 = 65,536`:

- In option 1, 100 full-rank matrices W' totals `16.8M × 100 = 1.68B` parameters.
- In option 2, one full-rank matrix W and 100 sets of small matrices (A, B) totals: `16.8M + 65,536 × 100 = 23.3M` parameters.

Option 2 also makes it faster to switch between tasks. Let's say you're currently serving customer X using this customer's model. To switch to serving customer Y, instead of loading this customer's full weight matrix, you only need to load Y's LoRA

adapter, which can significantly reduce the loading time. While keeping *A* and *B* separate incurs additional latency, there are optimization techniques to minimize the added latency. The book's GitHub repository contains a walkthrough of how to do so.

Multi-LoRA serving makes it easy to combine multiple specialized models. Instead of having one big powerful model for multiple tasks, you can have one LoRA adapter for each task. For example, Apple used multiple LoRA adapters to adapt the same 3B-parameter base model to different iPhone features (2024). They utilized quantization techniques to further reduce the memory footprint of this base model and adapters, allowing the serving of all of them on-device.

The modularity of LoRA adapters means that LoRA adapters can be shared and reused. There are publicly available finetuned LoRA adapters that you can use the way you'd use pre-trained models. You can find them on Hugging Face[26] or initiatives like AdapterHub.

You might be wondering: "LoRA sounds great, but what's the catch?" The main drawback of LoRA is that it doesn't offer performance as strong as full finetuning. It's also more challenging to do than full finetuning as it involves modifying the model's implementation, which requires an understanding of the model's architecture and coding skills. However, this is usually only an issue for less popular base models. PEFT frameworks—such as Hugging Face's PEFT, Axolotl, unsloth, and LitGPT—likely support LoRA for popular base models right out of the box.

Quantized LoRA. The rapid rise of LoRA has led to the development of numerous LoRA variations. Some aim to reduce the number of trainable parameters even further. However, as illustrated in Table 7-6, the memory of a LoRA adapter is minimal compared to the memory of the model's weights. Reducing the number of LoRA parameters decreases the overall memory footprint by only a small percentage.

Table 7-6. The memory needed by LoRA weights compared to that needed by the model's weights.

	Model's weights memory (16 bits)	LoRA trainable params (r=2, query & key matrices)	LoRA adapter memory (16 bits)
Llama 2 (13B)	26 GB	3.28M	6.55 MB
GPT-3 (175B)	350 GB	18.87M	37.7 MB

26 Search for these adapters by tags "adapter", "peft", or "LoRA".

Rather than trying to reduce LoRA's number of parameters, you can reduce the memory usage more effectively by quantizing the model's weights, activations, and/or gradients during finetuning. An early promising quantized version of LoRA is QLoRA (Dettmers et al., 2023).[27] In the original LoRA paper, during finetuning, the model's weights are stored using 16 bits. QLoRA stores the model's weights in 4 bits but dequantizes (converts) them back into BF16 when computing the forward and backward pass.

The 4-bit format that QLoRA uses is NF4 (NormalFloat-4), which quantizes values based on the insight that pre-trained weights usually follow a normal distribution with a median of zero. On top of 4-bit quantization, QLoRA also uses paged optimizers to automatically transfer data between the CPU and GPU when the GPU runs out of memory, especially with long sequence lengths. These techniques allow a 65B-parameter model to be finetuned on a single 48 GB GPU.

The authors finetuned a variety of models, including Llama 7B to 65B, in the 4-bit mode. The resulting family of models, called Guanaco, showed competitive performance on both public benchmarks and comparative evaluation. Table 7-7 shows the Elo ratings of Guanaco models, GPT-4, and ChatGPT in May 2023, as judged by GPT-4. While Guanaco 65B didn't outperform GPT-4, it was often preferred to ChatGPT.

Table 7-7. Elo ratings of Guanaco models compared to popular models in May 2023 using GPT-4 as a judge. The experiment is from QLoRA (Dettmers et al., 2023).

Model	Size	Elo
GPT-4	-	1348 ± 1
Guanaco 65B	41 GB	1022 ± 1
Guanaco 33B	21 GB	992 ± 1
Vicuna 13B	26 GB	974 ± 1
ChatGPT	-	966 ± 1
Guanaco 13B	10 GB	916 ± 1
Bard	-	902 ± 1
Guanaco 7B	6 GB	879 ± 1

The main limitation of QLoRA is that NF4 quantization is expensive. While QLoRA can reduce the memory footprint, it might increase training time due to the extra time required by quantization and dequantization steps.

27 QLoRA isn't the only quantized LoRA work. Many research labs have been working on quantized LoRA without publicly discussing it.

Due to its memory-saving promise, quantized LoRA is an active area of research. Other than QLoRA, quantized LoRA works include QA-LoRA (Xu et al., 2023), ModuLoRA (Yin et al., 2023), and IR-QLoRA (Qin et al., 2024).

Model Merging and Multi-Task Finetuning

If finetuning allows you to create a custom model by altering a single model, model merging allows you to create a custom model by combining multiple models. Model merging offers you greater flexibility than finetuning alone. You can take two available models and merge them together to create a new, hopefully more useful, model. You can also finetune any or all of the constituent models before merging them together.

While you don't have to further finetune the merged model, its performance can often be improved by finetuning. Without finetuning, model merging can be done without GPUs, making merging particularly attractive to indie model developers that don't have access to a lot of compute.

The goal of model merging is to create a single model that provides more value than using all the constituent models separately. The added value can come from improved performance. For example, if you have two models that are good at different things on the same task, you can merge them into a single model that is better than both of them on that task. Imagine one model that can answer the first 60% of the questions and another model that can answer the last 60% of the questions. Combined, perhaps they can answer 80% of the questions.

The added value can also come from a reduced memory footprint, which leads to reduced costs. For example, if you have two models that can do different tasks, they can be merged into one model that can do both tasks but with fewer parameters. This is particularly attractive for adapter-based models. Given two models that were finetuned on top of the same base model, you can combine their adapters into a single adapter.

One important use case of model merging is multi-task finetuning. Without model merging, if you want to a finetune a model for multiple tasks, you generally have to follow one of these approaches:

Simultaneous finetuning
: You create a dataset with examples for all the tasks and finetune the model on this dataset to make the model learn all the tasks simultaneously. However, because it's generally harder to learn multiple skills at the same time, this approach typically requires more data and more training.

Sequential finetuning
: You can finetune the model on each task separately but sequentially. After training a model on task A, train it on task B, and so on. The assumption is that it's easier for models to learn one task at a time. Unfortunately, neural networks are prone to catastrophic forgetting (Kirkpatrick et al., 2016). A model can forget how to do an old task when it's trained on a new task, leading to a significant performance drop on earlier tasks.

Model merging offers another method for multi-task finetuning. You can finetune the model on different tasks separately but in parallel. Once done, these different models are merged together. Finetuning on each task separately allows the model to learn that task better. Because there's no sequential learning, there's less risk of catastrophic forgetting.

Model merging is also appealing when you have to deploy models to devices such as phones, laptops, cars, smartwatches, and warehouse robots. On-device deployment is often challenging because of limited on-device memory capacity. Instead of squeezing multiple models for different tasks onto a device, you can merge these models together into one model that can perform multiple tasks while requiring much less memory.

On-device deployment is necessary for use cases where data can't leave the device (often due to privacy), or where there's limited or unreliable internet access. On-device deployment can also significantly reduce inference costs. The more computation you can offload to user devices, the less you have to pay to data centers.[28]

Model merging is one way to do *federated learning* (McMahan et al., 2016), in which multiple devices train the same model using separate data. For example, if you deploy model X to multiple devices, each copy of X can continue learning separately from the on-device data. After a while, you have multiple copies of X, all trained on different data. You can merge these copies together into one new base model that contains the learning of all constituent models.

The idea of combining models together to obtain better performance started with *model ensemble methods*. According to Wikipedia, ensembling combines "multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone." If model merging typically involves mixing parameters of constituent models together, ensembling typically combines only model outputs while keeping each constituent model intact.

28 My book, *Designing Machine Learning Systems* has a section on "ML on the Cloud and on the Edge."

For example, in ensembling, given a query, you might use three models to generate three different answers. Then, a final answer is generated based on these three answers, using a simple majority vote or another trainable ML module.[29] While ensembling can generally improve performance, it has a higher inference cost since it requires multiple inference calls per request.

Figure 7-13 compares ensembling and model merging. Just like model ensembles used to dominate leaderboards, many models on top of the Hugging Face's Open LLM Leaderboard are merged models.

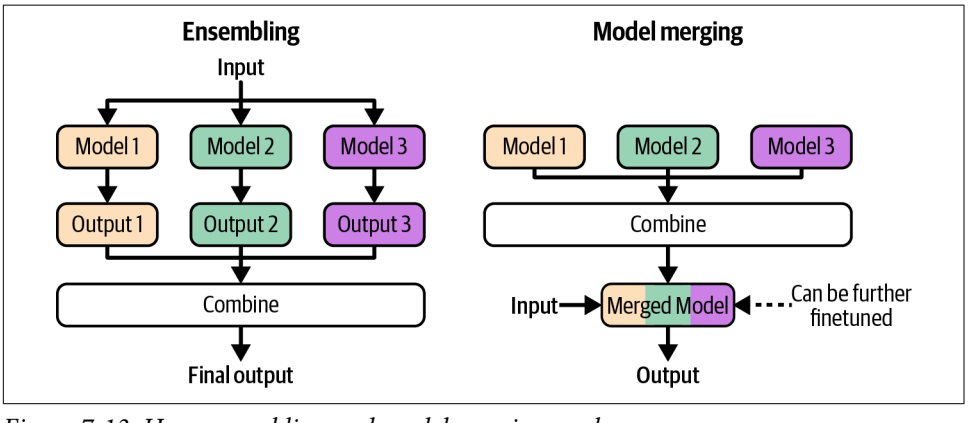

Figure 7-13. How ensembling and model merging work.

Many model-merging techniques are experimental and might become outdated as the community gains a better understanding of the underlying theory. For this reason, I'll focus on the high-level merging approaches instead of any individual technique.

Model merging approaches differ in how the constituent parameters are combined. Three approaches covered here are summing, layer stacking, and concatenation. Figure 7-14 shows their high-level differences.

29 You can read more about ensemble methods in my book *Designing Machine Learning Systems*.

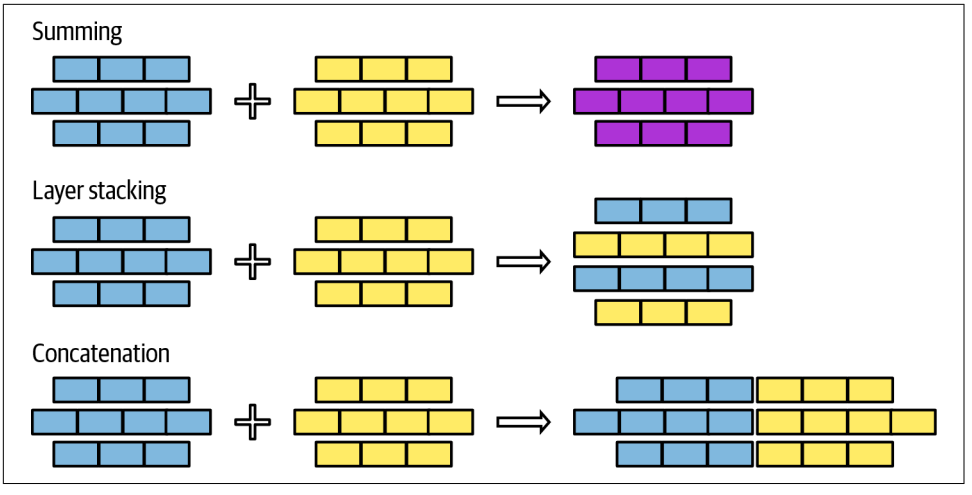

Figure 7-14. Three main approaches to model merging: summing, layer stacking, and concatenation.

You can mix these approaches when merging models, e.g., summing some layers and stacking others. Let's explore each of these approaches.

Summing

This approach involves adding the weight values of constituent models together. I'll discuss two summing methods: linear combination and spherical linear interpolation. If the parameters in two models are in different scales, e.g., one model's parameter values are much larger than the other's, you can rescale the models before summing so that their parameter values are in the same range.

Linear combination. Linear combination includes both an average and a weighted average. Given two models, A and B, their weighted average is:

$$\text{Merge(A, B)} = \frac{w_A A + w_B B}{w_A + w_B}$$

Figure 7-15 shows how to linearly combine two layers when $w_A = w_B = 1$.

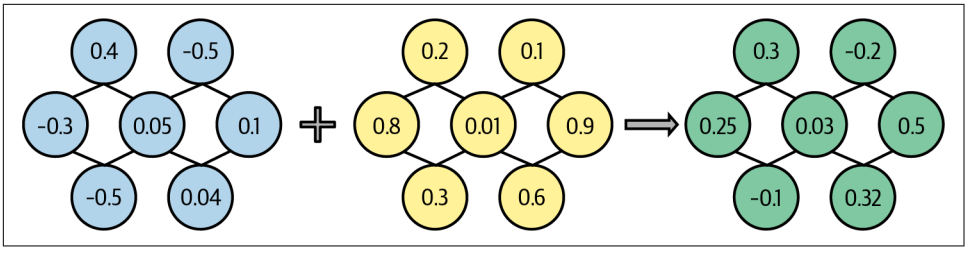

Figure 7-15. Merging parameters by averaging them.

Linear combination works surprisingly well, given how simple it is.[30] The idea that multiple models can be linearly combined to create a better one was studied as early as the early 1990s (Perrone, 1993). Linear combination is often used in federated learning (Wang et al., 2020).

You can linearly combine entire models or parts of models. Model soups (Wortsman et al., 2022) showed how averaging the entire weights of multiple finetuned models can improve accuracy without increasing inference time. However, it's more common to merge models by linearly combining specific components, such as their adapters.

While you can linearly combine any set of models, *linear combination is the most effective for models finetuned on top of the same base model.* In this case, linear combination can be viewed through the concept of *task vectors*. The idea is that once you've finetuned a model for a specific task, subtracting the base model from it should give you a vector that captures the essence of the task. Task vectors are also called *delta parameters*. If you finetune using LoRA, you can construct the task vector from the LoRA weights.

Task vectors allow us to do *task arithmetic* (Ilharco et al., 2022), such as adding two task vectors to combine task capabilities or subtracting a task vector to reduce specific capabilities. Task subtraction can be useful for removing undesirable model behaviors, such as invasive capabilities like facial recognition or biases obtained during pre-training.

Linear combination is straightforward when the components to be merged are of the same architecture and of the same size. However, it can also work for models that don't share the same architecture or the same size. For example, if one model's layer

30 Averaging works not just with weights but also with embeddings. For example, given a sentence, you can use a word embedding algorithm to generate an embedding vector for each word in the sentence, then average all these word embeddings into a sentence embedding. When I started out in ML, I couldn't believe that averaging seems to just work. It's magical when simple components, when used correctly, can create something so wonderfully perplexing, like AI.

is larger than that of the other model, you can project one or both layers into the same dimension.

Some people proposed aligning models before averaging to ensure that functionally related parameters are averaged together, such as in "Model Fusion via Optimal Transport" (Singh and Jaggi, 2020), "Git Re-Basin: Merging Models Modulo Permutation Symmetries" (Ainsworth et al., 2022), and "Merging by Matching Models in Task Parameter Subspaces" (Tam et al., 2023). While it makes sense to combine aligned parameters, aligning parameters can be challenging to do, and, therefore, this approach is less common on naive linear combinations.

Spherical linear interpolation (SLERP). Another common model summing method is SLERP, which is based on the mathematical operator of the same name, Spherical LinEar inteRPolation.

Interpolation means estimating unknown values based on known values. In the case of model merging, the unknown value is the merged model, and the known values are the constituent models. Linear combination is one interpolation technique. SLERP is another.

Because the formula for SLERP is mathy, and model-merging tools typically implement it for you, I won't go into the details here. Intuitively, you can think of each component (vector) to be merged as a point on a sphere. To merge two vectors, you first draw the shortest path between these two points along the sphere's surface. This is similar to drawing the shortest path between two cities along the Earth's surface. The merged vector of these two vectors is a point along their shortest path. Where exactly the point falls along the path depends on the interpolation factor, which you can set to be between 0 and 1. Factor values less than 0.5 bring the merged vector closer to the first vector, which means that the first task vector will contribute more to the result. A factor of 0.5 means that you pick a point exactly halfway. This middle point is the blue point in Figure 7-16.

SLERP, as a mathematical operation, is defined with only two vectors, which means that you can merge only two vectors at a time. If you want to merge more than two vectors, you can potentially do SLERP sequentially, i.e., merging A with B, and then merging that result with C.

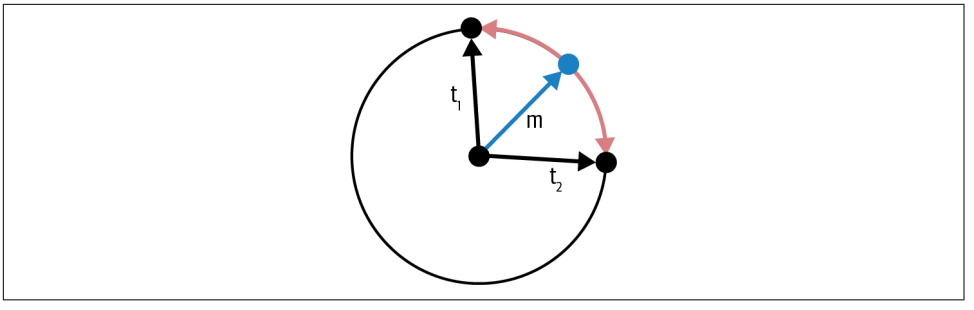

Figure 7-16. How SLERP works for two vectors t1 and t2. The red line is their shortest path on the spherical surface. Depending on the interpolation, the merged vector can be any point along this path. The blue vector is the resulting merged vector when the interpolation factor is 0.5.

Pruning redundant task-specific parameters. During finetuning, many models' parameters are adjusted. However, most of these adjustments are minor and don't significantly contribute to the model's performance on the task.[31] Adjustments that don't contribute to the model's performance are considered *redundant*.

In the paper "TIES-Merging: Resolving Interference When Merging Models", Yadav et al. (2023) showed that you can reset a large portion of task vector parameters with minimal performance degradation, as shown in Figure 7-17. Resetting means changing the finetuned parameter to its original value in the base model, effectively setting the corresponding task vector parameter to zero. (Recall that the task vector can be obtained by subtracting the base model from the finetuned model.)

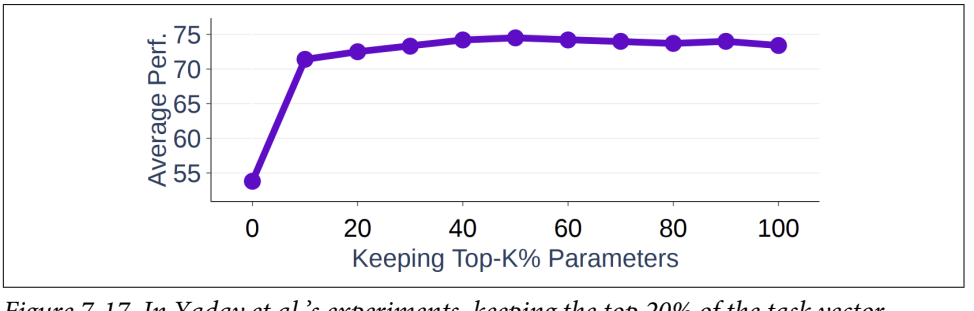

Figure 7-17. In Yadav et al.'s experiments, keeping the top 20% of the task vector parameters gives comparable performance to keeping 100% of the parameters.

31 The assumption is that the parameters that undergo the most substantial changes during finetuning are the ones most crucial for the target task.

These redundant parameters, while not harmful to one model, might be harmful to the merged model. Merging techniques such as TIES (Yadav et al., 2023) and DARE (Yu et al., 2023) first prune the redundant parameters from task vectors before merging them.[32] Both papers showed that this practice can significantly improve the quality of the final merged models. The more models there are to merge, the more important pruning is because there are more opportunities for redundant parameters in one task to interfere with other tasks.[33]

Layer stacking

In this approach, you take different layers from one or more models and stack them on top of each other. For example, you might take the first layer from model 1 and the second layer from model 2. This approach is also called *passthrough* or *frankenmerging*. It can create models with unique architectures and numbers of parameters. Unlike the merging by summing approach, the merged models resulting from layer stacking typically require further finetuning to achieve good performance.

One early success of frankenmerging is Goliath-120B (alpindale, 2023), which was merged from two finetuned Llama 2-70B models, Xwin and Euryale. It took 72 out of 80 layers from each model and merged them together.

Layer stacking can be used to train mixture-of-experts (MoE) models, as introduced in "Sparse Upcycling: Training Mixture-of-Experts from Dense Checkpoints" (Komatsuzaki et al., 2022). Rather than training an MOE from scratch, you take a pre-trained model and make multiple copies of certain layers or modules. A router is then added to send each input to the most suitable copy. You then further train the merged model along with the router to refine their performance. Figure 7-18 illustrates this process.

Komatsuzaki et al. showed that layer stacking can produce models that outperform MoE models trained from scratch. Using this approach, Together AI mixed six weaker open source models together to create Mixture-of-Agents, which achieved comparable performance to OpenAI's GPT-4o in some benchmarks (Wang et al., 2024).

32 TIES is abbreviated from "TrIm, Elect Sign, and merge," while DARE is from "Drop And REscale." I know, these abbreviations pain me too.

33 When task vectors are pruned, they become more sparse, but the finetuned model doesn't. Pruning, in this case, isn't to reduce the memory footprint or inference latency, but to improve performance.

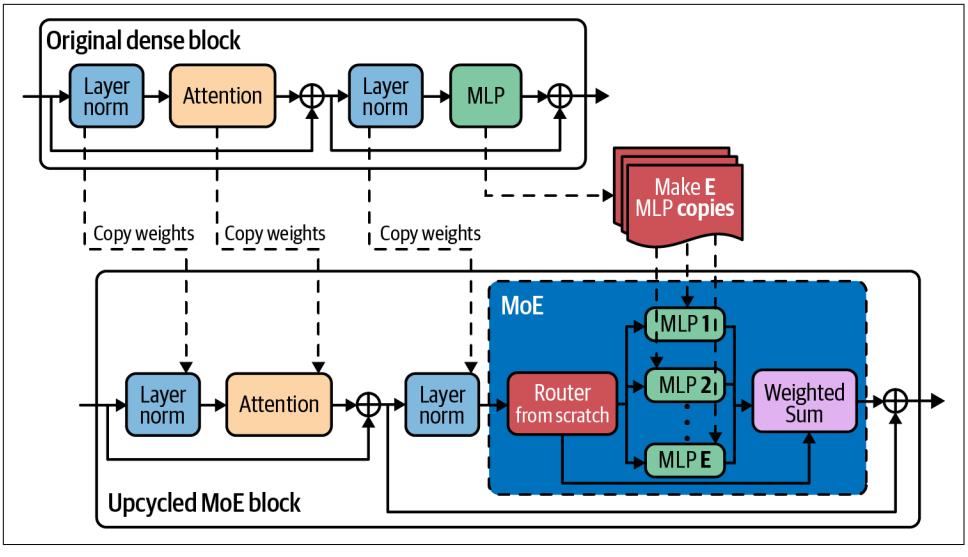

Figure 7-18. You can create an MoE model from a pre-trained model. Image adapted from Komatsuzaki et al. (2022).

An interesting use case of layer stacking is *model upscaling*. Model upscaling is the study of how to create larger models using fewer resources. Sometimes, you might want a bigger model than what you already have, presumably because bigger models give better performance. For example, your team might have originally trained a model to fit on your 40 GB GPU. However, you obtained a new machine with 80 GB, which allows you to serve a bigger model. Instead of training a new model from scratch, you can use layer stacking to create a larger model from the existing model.

One approach to layer upscaling is *depthwise scaling*. Kim et al. (2023) used this technique to create SOLAR 10.7B from one 7B-parameter model with 32 layers. The procedure works as follows:

1. Make a copy of the original pre-trained model.
2. Merge these two copies by summing certain layers (summing two layers and turning them into one layer) and stacking the rest. The layers to be summed are carefully selected to match the target model size. For SOLAR 10.7B, 16 layers are summed, leaving the final model with $32 \times 2 - 16 = 48$ layers.
3. Further train this upscaled model toward the target performance.

Figure 7-19 illustrates this process.

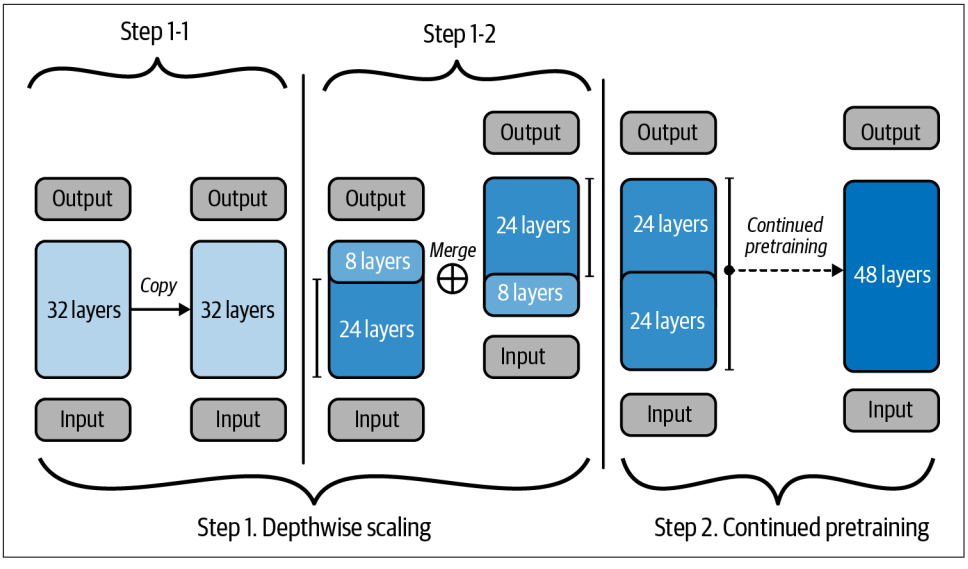

Figure 7-19. Use depthwise scaling to create a 48-layer model from a 32-layer model. The image is licensed under CC BY 4.0 and was slightly modified for readability.

Concatenation

Instead of adding the parameters of the constituent models together in different manners, you can also concatenate them. The merged component's number of parameters will be the sum of the number of parameters from all constituent components. If you merge two LoRA adapters of ranks r_1 and r_2, the merged adapter's rank will be $r_1 + r_2$, as shown in Figure 7-20.

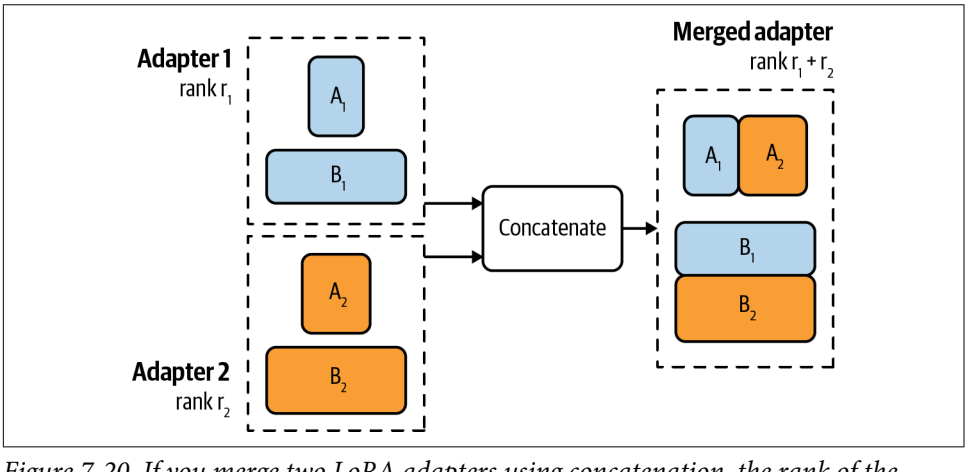

Figure 7-20. If you merge two LoRA adapters using concatenation, the rank of the merged adapter will be the sum of both adapters' ranks.

Concatenation isn't recommended because it doesn't reduce the memory footprint compared to serving different models separately. Concatenation might give better performance, but the incremental performance might not be worth the number of extra parameters.[34]

Finetuning Tactics

This chapter has discussed multiple finetuning approaches, what problems they solve, and how they work. In this last section, I'll focus on more practical finetuning tactics.

Finetuning frameworks and base models

While many things around finetuning—deciding whether to finetune, acquiring data, and maintaining finetuned models—are hard, the actual process of finetuning is more straightforward. There are three things you need to choose: a base model, a finetuning method, and a framework for finetuning.

Base models. Chapter 4 already covered the criteria for model selection that can be applied to both prompt-based methods and finetuning. Some of the criteria discussed include model size, licenses, and benchmark performance. At the beginning of an AI project, when you're still exploring the feasibility of your task, it's useful to start with the most powerful model you can afford. If this model struggles to produce good results, weaker models are likely to perform even worse. If the strongest model meets your needs, you can then explore weaker models, using the initial model as a benchmark for comparison.

For finetuning, the starting models vary for different projects. OpenAI's finetuning best practices document gives examples of two development paths: the progression path and the distillation path.

The progression path looks like this:

1. Test your finetuning code using the cheapest and fastest model to make sure the code works as expected.[35]
2. Test your data by finetuning a middling model. If the training loss doesn't go down with more data, something might be wrong.

34 I debated for a long time whether to include the concatenation technique in this book, and decided to include it for completeness.

35 In college, I made the painful mistake of letting my model train overnight, only to have it crash after eight hours because I tried to save the checkpoint in a nonexistent folder. All that progress was lost.

3. Run a few more experiments with the best model to see how far you can push performance.
4. Once you have good results, do a training run with all models to map out the price/performance frontier and select the model that makes the most sense for your use case.

The distillation path might look as follows:

1. Start with a small dataset and the strongest model you can afford. Train the best possible model with this small dataset. Because the base model is already strong, it requires less data to achieve good performance.
2. Use this finetuned model to generate more training data.
3. Use this new dataset to train a cheaper model.

Because finetuning usually comes after experiments with prompt engineering, by the time you start to finetune, ideally, you should have a pretty good understanding of different models' behaviors. You should plan your finetuning development path based on this understanding.

Finetuning methods. Recall that adapter techniques like LoRA are cost-effective but typically don't deliver the same level of performance as full finetuning. If you're just starting with finetuning, try something like LoRA, and attempt full finetuning later.

The finetuning methods to use also depend on your data volume. Depending on the base model and the task, full finetuning typically requires at least thousands of examples and often many more. PEFT methods, however, can show good performance with a much smaller dataset. If you have a small dataset, such as a few hundred examples, full finetuning might not outperform LoRA.

Take into account how many finetuned models you need and how you want to serve them when deciding on a finetuning method. Adapter-based methods like LoRA allow you to more efficiently serve multiple models that share the same base model. With LoRA, you only need to serve a single full model, whereas full finetuning requires serving multiple full models.

Finetuning frameworks. The easiest way to finetune is to use a finetuning API where you can upload data, select a base model, and get back a finetuned model. Like model inference APIs, finetuning APIs can be provided by model providers, cloud service providers, and third-party providers. A limitation of this approach is that you're limited to the base models that the API supports. Another limitation is that the API might not expose all the knobs you can use for optimal finetuning performance. Finetuning APIs are suitable for those who want something quick and easy, but they might be frustrating for those who want more customization.

You can also finetune using one of many great finetuning frameworks available, such as LLaMA-Factory, unsloth, PEFT, Axolotl, and LitGPT. They support a wide range of finetuning methods, especially adapter-based techniques. If you want to do full finetuning, many base models provide their open source training code on GitHub that you can clone and run with your own data. Llama Police has a more comprehensive and up-to-date list of finetuning frameworks and model repositories.

Doing your own finetuning gives you more flexibility, but you'll have to provision the necessary compute. If you do only adapter-based techniques, a mid-tier GPU might suffice for most models. If you need more compute, you can choose a framework that integrates seamlessly with your cloud provider.

To finetune a model using more than one machine, you'll need a framework that helps you do distributed training, such as DeepSpeed, PyTorch Distributed, and ColossalAI.

Finetuning hyperparameters

Depending on the base model and the finetuning method, there are many hyperparameters you can tune to improve finetuning efficiency. For specific hyperparameters for your use case, check out the documentation of the base model or the finetuning framework you use. Here, I'll cover a few important hyperparameters that frequently appear.

Learning rate. The learning rate determines how fast the model's parameters should change with each learning step. If you think of learning as finding a path toward a goal, the learning rate is the step size. If the step size is too small, it might take too long to get to the goal. If the step size is too big, you might overstep the goal, and, hence, the model might never converge.

A universal optimal learning rate doesn't exist. You'll have to experiment with different learning rates, typically between the range of 1e-7 to 1e-3, to see which one works best. A common practice is to take the learning rate at the end of the pre-training phase and multiply it with a constant between 0.1 and 1.

The loss curve can give you hints about the learning rate. If the loss curve fluctuates a lot, it's likely that the learning rate is too big. If the loss curve is stable but takes a long time to decrease, the learning is likely too small. Increase the learning rate as high as the loss curve remains stable.

You can vary learning rates during the training process. You can use larger learning rates in the beginning and smaller learning rates near the end. Algorithms that determine how learning rates should change throughout the training process are called learning rate schedules.

Batch size. The batch size determines how many examples a model learns from in each step to update its weights. A batch size that is too small, such as fewer than eight, can lead to unstable training.[36] A larger batch size helps aggregate the signals from different examples, resulting in more stable and reliable updates.

In general, the larger the batch size, the faster the model can go through training examples. However, the larger the batch size, the more memory is needed to run your model. Thus, batch size is limited by the hardware you use.

This is where you see the cost versus efficiency trade-off. More expensive compute allows faster finetuning.

As of this writing, compute is still a bottleneck for finetuning. Often, models are so large, and memory is so constrained, that only small batch sizes can be used. This can lead to unstable model weight updates. To address this, instead of updating the model weights after each batch, you can accumulate gradients across several batches and update the model weights once enough reliable gradients are accumulated. This technique is called *gradient accumulation*.[37]

When compute cost isn't the most important factor, you can experiment with different batch sizes to see which gives the best model performance.

Number of epochs. An epoch is a pass over the training data. The number of epochs determines how many times each training example is trained on.

Small datasets may need more epochs than large datasets. For a dataset with millions of examples, 1–2 epochs might be sufficient. A dataset with thousands of examples might still see performance improvement after 4–10 epochs.

The difference between the training loss and the validation loss can give you hints about epochs. If both the training loss and the validation loss still steadily decrease, the model can benefit from more epochs (and more data). If the training loss still decreases but the validation loss increases, the model is overfitting to the training data, and you might try lowering the number of epochs.

36 While it's commonly acknowledged that small batch sizes lead to unstable training, I wasn't able to find good explanations for why that's the case. If you have references about this, please feel free to send them my way.

37 I tried to find the first paper where gradient accumulation was introduced but couldn't. Its use in deep learning was mentioned as early as 2016 in "Ako: Decentralised Deep Learning with Partial Gradient Exchange" (Watcharapichat et al., *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016). The concept seems to come from distributed training, where gradients computed on different machines need to be accumulated and used to update the model's weights.

Prompt loss weight. For instruction finetuning, each example consists of a prompt and a response, both of which can contribute to the model's loss during training. During inference, however, prompts are usually provided by users, and the model only needs to generate responses. Therefore, response tokens should contribute more to the model's loss during training than prompt tokens.

The prompt model weight determines how much prompts should contribute to this loss compared to responses. If this weight is 100%, prompts contribute to the loss as much as responses, meaning that the model learns equally from both. If this weight is 0%, the model learns only from responses. Typically, this weight is set to 10% by default, meaning that the model should learn some from prompts but mostly from responses.

Summary

Outside of the evaluation chapters, finetuning has been the most challenging chapter to write. It touched on a wide range of concepts, both old (transfer learning) and new (PEFT), fundamental (low-rank factorization) and experimental (model merging), mathematical (memory calculation) and tactical (hyperparameter tuning). Arranging all these different aspects into a coherent structure while keeping them accessible was difficult.

The process of finetuning itself isn't hard. Many finetuning frameworks handle the training process for you. These frameworks can even suggest common finetuning methods with sensible default hyperparameters.

However, the context surrounding finetuning is complex. It starts with whether you should even finetune a model. This chapter started with the reasons for finetuning and the reasons for not finetuning. It also discussed one question that I have been asked many times: when to finetune and when to do RAG.

In its early days, finetuning was similar to pre-training—both involved updating the model's entire weights. However, as models increased in size, full finetuning became impractical for most practitioners. The more parameters to update during finetuning, the more memory finetuning needs. Most practitioners don't have access to sufficient resources (hardware, time, and data) to do full finetuning with foundation models.

Many finetuning techniques have been developed with the same motivation: to achieve strong performance on a minimal memory footprint. For example, PEFT reduces finetuning's memory requirements by reducing the number of trainable parameters. Quantized training, on the other hand, mitigates this memory bottleneck by reducing the number of bits needed to represent each value.

After giving an overview of PEFT, the chapter zoomed into LoRA—why and how it works. LoRA has many properties that make it popular among practitioners. On top

of being parameter-efficient and data-efficient, it's also modular, making it much easier to serve and combine multiple LoRA models.

The idea of combining finetuned models brought the chapter to model merging; its goal is to combine multiple models into one model that works better than these models separately. This chapter discussed the many use cases of model merging, from on-device deployment to model upscaling, and general approaches to model merging.

A comment I often hear from practitioners is that finetuning is easy, but getting data for finetuning is hard. Obtaining high-quality annotated data, especially instruction data, is challenging. The next chapter will dive into these challenges.