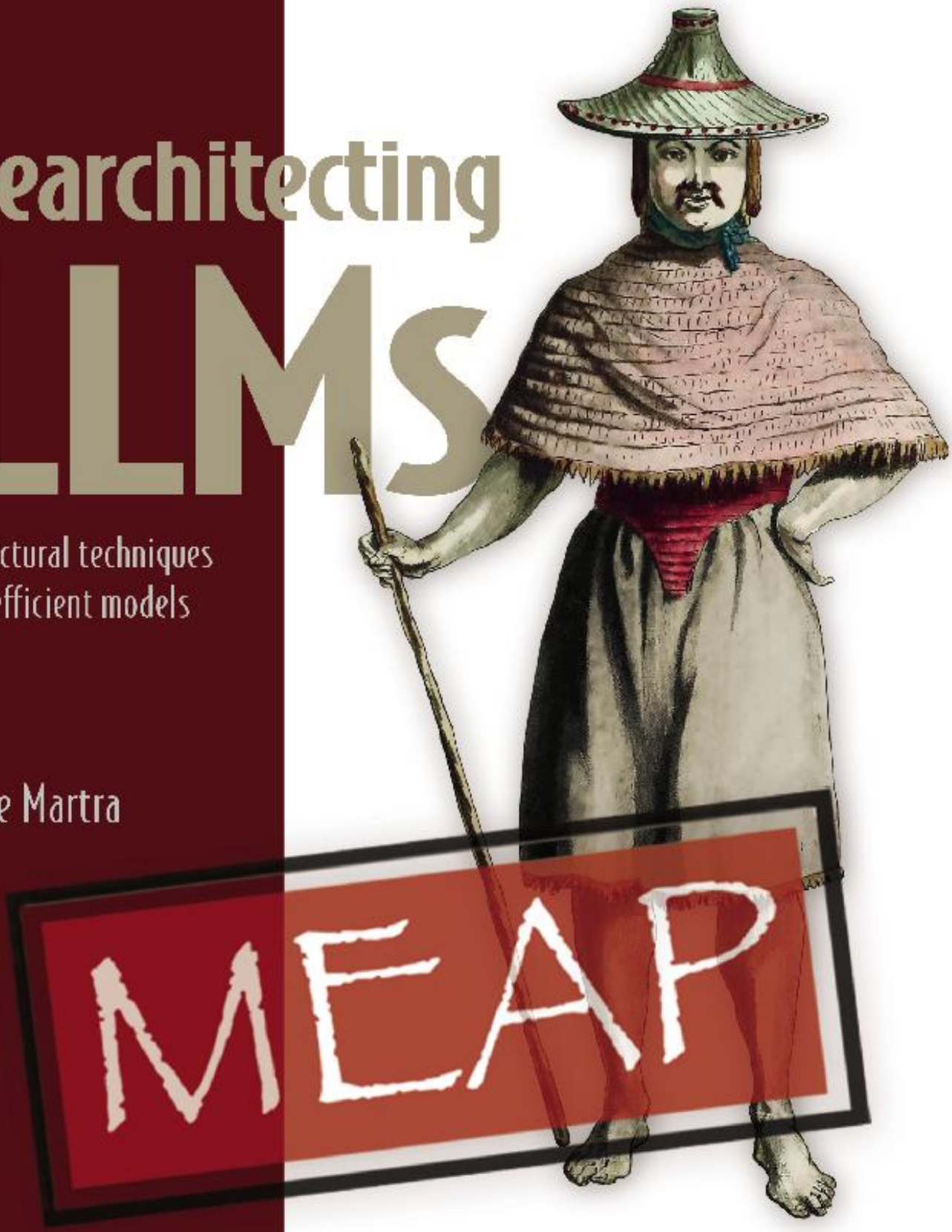
Rearchitecting LLMs

Structural techniques for efficient models

Pere Martra



MANNING

MEAP Edition
Manning Early Access Program

Rearchitecting LLMs

Structural techniques for efficient models

Version 7

welcome

Thank you for purchasing the MEAP of *Rearchitecting LLMs*. I'm glad to have you on board and look forward to your feedback in this early stage of the project.

This book is for engineers who use LLMs, whether open source or via APIs, and want to move beyond fine-tuning to optimize model architectures. If you have knowledge of PyTorch and curiosity about what happens inside a transformer, this book is for you. We'll start from concepts you already know and guide you through surgical optimization techniques currently used in research teams.

My first optimization project was for a startup detecting violent language and political analysis on the Internet. Their DistilGPT2-based model worked well, but their team needed to analyze more text in less time. Applying pruning and knowledge distillation took weeks, as the information was scattered across academic papers, many without reproducible code.

That first optimization opened up a fascinating world. I began actively seeking more optimization projects. As the industry adopted more modern open source models, the challenge grew: techniques like width pruning in GLU architectures were poorly documented, forcing me to develop my own methods combining papers like "ShortGPT" with broader approaches like "The Minitron Approach". This book exists to solve that fragmentation: unifying academic techniques into a coherent, practical pipeline.

Throughout this book, you'll learn to transform generic models into optimized solutions for your specific needs.

You'll master depth pruning and width pruning techniques to reduce size and increase speed, knowledge distillation to recover lost knowledge, and LoRA for domain specialization. You'll implement attention bypass systems that adapt the model dynamically, and apply fair pruning to build more equitable architectures.

Your feedback is fundamental to making this book as useful as possible. Each chapter includes a hands-on section with practical exercises designed for you to apply what you've learned immediately. I encourage you to share your questions, results or suggestions both in the liveBook discussion forum and in the discussion sections prepared for this purpose in the book's code repository. I'm looking forward to seeing how you apply these surgical optimization techniques in your own projects.

—Pere Martra

Brief contents

1 Why rearchitecting LLMs matters

This chapter covers

- Why generic LLMs don’t work
- The model re-architects the pipeline as a solution
- Core techniques for building specialized models
- A roadmap to architecting

Large language models (LLMs) are trained on large collections of text from many languages and subject areas. As a result, they can have hundreds of billions of parameters, sometimes approaching a trillion. They can handle many tasks, such as writing poetry, analyzing financial documents, generating code, and translating between languages. This breadth of knowledge is the source of their power, but it can also make them inefficient for specialized tasks, using more time and resources than necessary.

Let’s take an example from the financial sector: one application might need to analyze thousands of legal documents to extract key metrics, while another generates narrative reports for clients. Although both tasks share a base knowledge, their architectural demands are different. The first requires precise synthesis and deep interpretation of relationships; the second requires generative fluency. Many organizations will use the same model for both tasks, relying on *prompt engineering*, the art of crafting detailed instructions to guide the model’s behavior. This strategy has

two critical limitations: it’s not cost-effective in the long run, and it doesn’t differentiate your LLM from competitors’.

This is where small language models (SLMs) come into play: models that typically range from a few million to a few billion parameters. They are inherently lightweight and fast, designed to serve as the building blocks of an ecosystem where different SLMs collaborate and interact with other software components. This book will teach you techniques for *model rearchitecting*: adapting a pretrained model to fit your specific requirements, your data, and your deployment constraints. This structural approach physically modifies the model's architecture rather than just adjusting its behavior. Starting from an existing model, you will reshape its structure to obtain a new one, with fewer parameters, fewer layers, and a different architecture. You'll learn how to perform precise interventions on the model, starting with foundational techniques like structured pruning and knowledge distillation and advancing to state-of-the-art strategies to create efficient models.

But before we start rearchitecting models, we need to clearly understand why to do it. In the next section, we’ll analyze the main challenges facing current LLMs, ranging from computational cost and environmental impact to equity problems and a lack of specialization that slow their adoption in many companies.

1.1 Current challenges to scaling LLMs

Many proof of concepts (POCs) are moving to production, and scaling solutions for production use is a real challenge. Let’s examine the challenges across the most important dimensions for a company: operational cost and long-term sustainability, competitive differentiation, and explainability.

1.1.1 Operational costs

The first cost is overengineering. For creating POCs, models accessed via an API are the perfect solution. They're simple to use with just a few calls. Cost depends directly on the tests you run: you pay for the tokens used, and data typically isn't a concern because you're likely not using real data. The real problems emerge when we move to production. You can go from a few dozen daily API calls to hundreds or thousands, with an ever-increasing token count. What was once an economical way to validate a project can become a system whose costs scale inefficiently with usage.

Costs don't just increase; they're also hard to control. You're charged for both input and output tokens, and output is almost impossible to predict, especially in agent systems with calls to different tools and recursive calls between agents.

Cost isn't unique to API solutions; any oversized or poorly optimized system can run into the same problem. I once met with a pharmaceutical team spending astronomical amounts to keep high-end GPUs running 24/7 for a model they used "a few times a day." Their proposed solution? Use an API model.

The problem wasn't "on-premises vs. API," but a fundamental mismatch between the tool and the task. They were also using a model much larger than they needed. The solution wasn't to migrate but to replace their LLM with a much smaller, specialized model adapted to their data and tasks. The resulting SLM was transformative. Not only were costs reduced, but as inference became faster, the team started experimenting more and more. What was once an expensive, tightly rationed resource became an agile, accessible tool.

1.1.2 Competitive differentiation

Second is the generic trap. If every company uses GPT-5, Claude, Gemini, or any other generalist model via an API, financial analysts will get the same recommendations. Imagine for a moment that most investment recommendation software, created with the same models and fed with practically the same data, starts to issue the same buy recommendations for a couple of Nasdaq stocks (or any other index). These recommendations would affect the value of the stocks themselves, but we'd have another problem: the recommendation doesn't offer any value to our client. It's the same one that everyone has access to, even those who access the model through their own chat application, like ChatGPT, Gemini, or Le Chat from Mistral. When everyone has access to the same model, your differentiation comes from what only you have: your data and a model specialized around it.

For some time, prompt engineering has been used to shape a model's responses. While adjusting instructions or using well-designed prompt chains can improve results, these methods have limits. The model's underlying expertise remains the same.

You also won't get meaningfully differentiated performance by using your own data through a *retrieval-augmented generation (RAG)* solution, a technique that allows a model to access external information. RAG is useful for integrating up-to-date and private knowledge, but it doesn't transform the model; it gives the LLM access to new information. The way this information is processed remains generic. In fact, RAG and the techniques in this book are complementary: many rearchitecting projects exist precisely to adapt a model to work correctly within a RAG pipeline, improving how it retrieves, interprets, and uses external information.

The next logical step is *fine-tuning*, a process in which a model is further trained on a specific dataset to adapt it to a particular task. But fine-tuning a closed model such as Gemini or GPT means using the provider's own tools, and it is extremely expensive. Even with efficient fine-tuning techniques, training requires running data through models with hundreds of billions of parameters, and you are paying for every token processed in those training passes. And you still don’t have access to the base model. Once you have the fine-tuned model (which you’ve paid for), the cost per inference is also higher. So you’ve paid to adapt the model, and you’ll pay more each time you need to use it.

NOTE

Model providers’ pricing policies change constantly, so it doesn’t make sense to give even an order-of-magnitude estimate of the cost difference between using a standard model and a fine-tuned model, because for some providers that difference could disappear at any moment. It’s also worth noting that OpenAI has offered significant discounts on fine-tuning its models when the data used in the process is shared.

Fine-tuning on closed models is like building a custom house on leased land: your fine-tuned model is a hosted artifact tied to the provider’s base model. You can’t export the weights. You can pin a specific version for stability, but when the base model is upgraded or retired, you should expect to rerun the fine-tuning using the same dataset and evaluations. Combined with the pricing unpredictability mentioned earlier, this creates a classic case of AI-specific vendor lock-in, where switching providers becomes prohibitively expensive not only financially, but also technically.

This problem isn't limited to closed API models. Open-source models often fall into the same pattern, and for a structural reason. Many of them, especially the largest and most powerful ones, are designed to perform well on general benchmarks and top leaderboards. Features like massive context windows, the amount of text the model can consider at once, or broad multilingual coverage often go unused in domain-specific models and can even hurt performance

This obsession with benchmarks stems from where these foundation models are built. Most come from academic labs or big tech research teams—environments that prioritize general capabilities and publishable results over solving specific business problems. When your goal is to advance the field rather than optimize for legal document analysis or medical diagnosis, standardized benchmarks become your primary success metric. That’s logical, but it creates models that excel at tests rather than real-world tasks.

Large models aren’t the only ones that can benefit from a targeted rearchitecture process. Smaller 7B models can also specialize and achieve significant performance improvements by adapting their architecture.

A good example is the FineScope paper (https://arxiv.org/abs/2505.00624). The authors use a domain-specific dataset to guide structured pruning, so that the components most relevant to the target domain are the ones preserved. On a Llama 3.1 8B model, this domain-guided pruning keeps accuracy stable up to around 35% pruning, while pruning guided by generic data already starts to degrade at 25%. We'll return to this idea, using a dataset to decide what to remove, as a central theme of the book.

1.1.3 Explainability

Finally, there is the black-box problem. In regulated sectors like finance and healthcare, explainability isn't optional; it's legally required. With API-based models, the problem is absolute: you have access to nothing but an API endpoint. Even worse, unannounced updates can subtly alter behavior, breaking production systems without warning.

Traditionally, open-source models suffered from similar opacity. Although you could access the weights, the neural network’s complexity meant the models were treated as black boxes in practice. Surgical optimization offers a practical advantage here. Instead of accepting opacity, we’ll teach you to analyze *neuron activations* (the numerical internal outputs of the model’s layers) and observe which components “fire” as the model processes information. This technique helps you understand and even influence model behavior.

These challenges aren’t just isolated problems; they’re symptoms of a fundamental mismatch between generic models and specific business needs. One solution is model rearchitecting: a systematic approach to structural optimization that we’ll explore in detail in the next section.

NOTE

Models don’t specialize only by adjusting their knowledge. They also specialize by modifying their structure: reducing unnecessary layers, reconfiguring attention, and adapting the size and shape of their internal blocks to the task.

1.2 The solution: the model rearchitecting pipeline

The problems we've seen in the previous section have a common origin: using generalist models built with benchmarks in mind rather than real-world functionality, and with needs completely different from those in your projects.

So far, the most common way to modify a model and adapt it to a new domain or project has been only fine-tuning.

The approach proposed in this book goes further, addressing the problem at its root: the LLM's architecture. It does this with a combination of techniques: *pruning*, the targeted removal of model components; *knowledge distillation*, where a smaller "student" model learns from a larger "teacher"; *fine-tuning* for domain specialization; and studying the model's internal behavior to precisely correct unwanted responses.

The process follows three sequential phases, shown in figure 1.1, each with a specific objective and building on the previous phase.

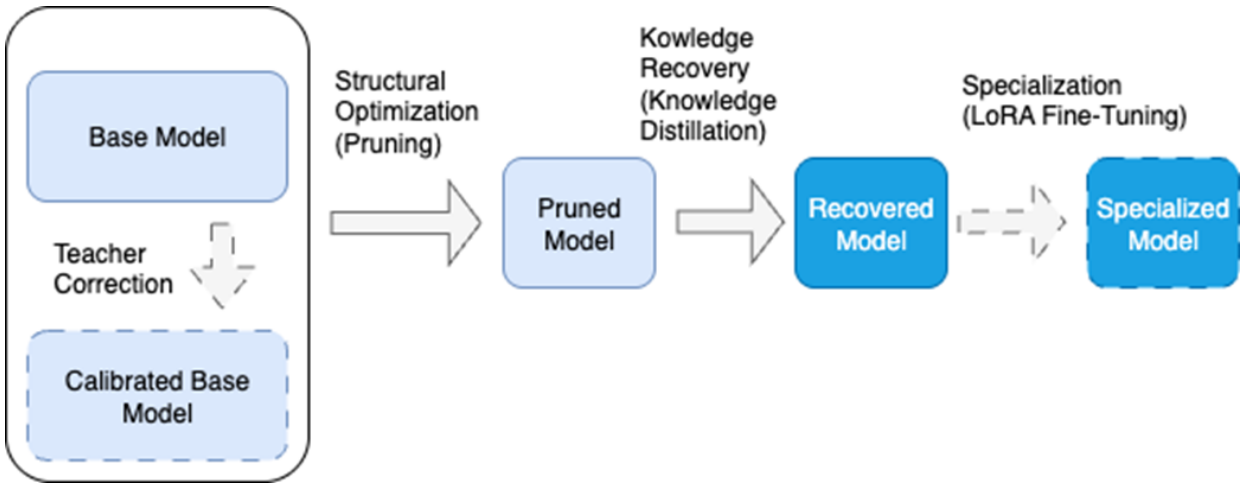


Figure 1.1 The model tailoring pipeline consists of core phases (shown with solid arrows) and optional phases (shown with dashed arrows). In the first phase, we adapt the structure to the model's objectives through pruning. Next, we recover capabilities it may have lost through knowledge distillation. Finally, we can optionally specialize the model through fine-tuning. An optional initial phase calibrates the base model, via a brief fine-tuning pass on the target dataset.

The pipeline is modular by design, not every project requires all three phases. The goal isn't always specialization. In many cases, the goal is simply to reduce a model's computational requirements so it can run on edge devices while maintaining its general capabilities. In such scenarios, you can use a minimal pipeline with just two core phases: structural optimization and knowledge recovery.

1.2.1 Structural optimization

As shown in the overall pipeline, the first phase is structural optimization of the model, which we'll do using pruning. During this process, the model removes the parts that contribute the least to the task it's being prepared for.

Removing parts of the model's structure will affect its performance depending on the technique used and, especially, on the extent of the restructuring. In this book, we'll see techniques ranging from aggressive ones that remove entire blocks of the model and require a recovery process to surgical interventions that remove just a few neurons to modify the model's behavior without changing its general capabilities.

1.2.2 Knowledge distillation

We recover the capabilities lost during this phase through knowledge distillation, where the base model acts as the teacher for the pruned model (the student). The student learns not just the correct answers, but also to replicate the teacher's internal reasoning patterns—how it processes information step by step.

This way, the student learns not only *what* to respond, but also *how* the teacher arrives at the final answer, step by step. As we'll see in an initial practical approach in chapter 2

and in more depth in chapter 6, this allows much deeper knowledge to be transferred, so recovery happens more efficiently and uses less computation time.

1.2.3 Specialization

Once the model has recovered its general capabilities, we specialize it with Low-Rank Adaptation (LoRA). This parameter-efficient technique trains only a small subset of parameters, reducing both computational cost and training time compared to traditional fine-tuning.

This order saves the most compute because each fine-tuning phase runs on smaller, cheaper-to-train models.

1.2.4 Optional: Teacher correction

At the beginning of the process, there's an optional phase called "Teacher Correction." NVIDIA introduced this term when creating its model families. A light fine-tuning pass is applied to the base model using the same data later used to identify which parts of the model to remove and to perform specialization fine-tuning. This aligns the base model so it works better during the knowledge recovery phase.

1.2.5 How data moves through the pipeline

Suppose you want to build a model for a specific type of data. Whether it's medical, financial, or biological, the data will play a central role throughout the process.

Let's see where they fit into the pipeline.

The domain-specific dataset acts as the backbone of the process, as shown in figure 1.2. It guides each decision. It helps identify which parts of the model can be removed,

supports knowledge recovery through prior calibration, and is ultimately used to specialize the resulting model. This kind of structural modification, in which we physically alter the model's structure (removing or bypassing components) rather than just updating its weights, fundamentally differentiates model rearchitecting from traditional fine-tuning.

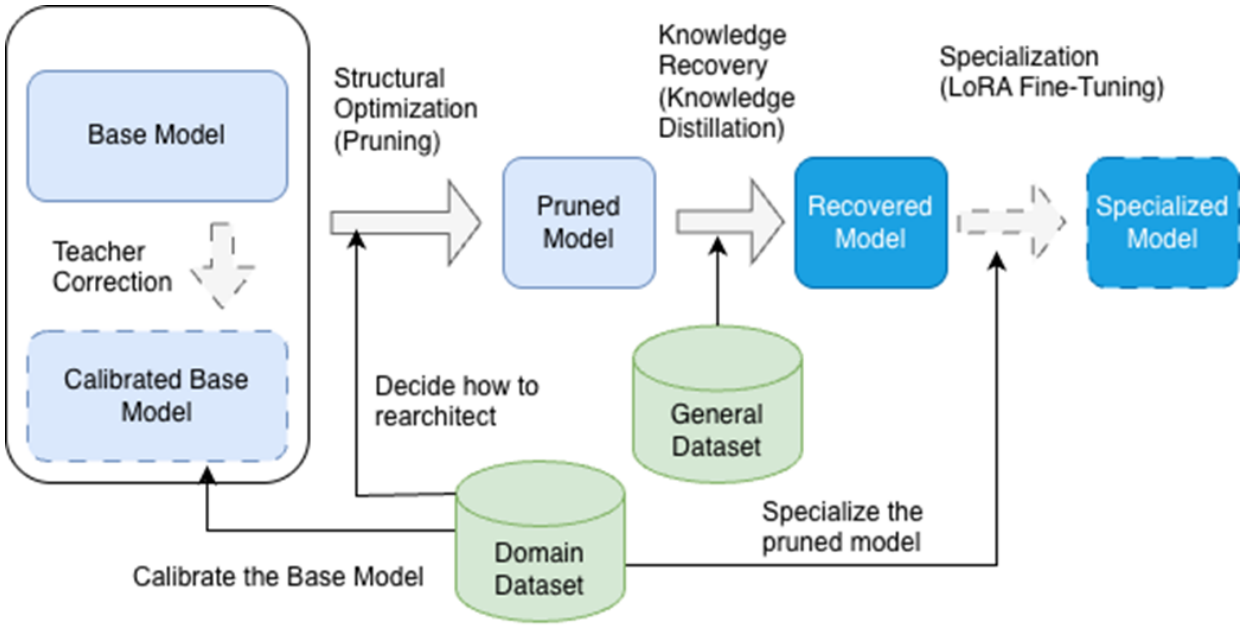


Figure 1.2 Dataset integration in the rearchitecting pipeline. The domain-specific dataset guides calibration of the base model, informs structural optimization decisions, and enables final specialization through LoRA fine-tuning. A general dataset supports Knowledge Recovery, ensuring the pruned model retains broad capabilities before domain-specific specialization. This dual approach optimizes each phase for the project's objectives.

Let's see how this pipeline works in practice with a concrete example. Imagine you work for a financial services company that needs to optimize a 7B general-purpose model to analyze quarterly earnings reports.

You'd use the financial reports dataset to calibrate the base model. Then, using the same dataset, you'd identify which

layers or components contribute least to financial analysis and prune them to lighten the model’s structure. The pruned model initially shows degraded performance on general metrics and even on the financial tasks you’re preparing it for.

Knowledge recovery through knowledge distillation restores the pruned model’s general capabilities using a general-purpose dataset. Then, we specialize the recovered model to a specific domain using LoRA fine-tuning on the financial reports dataset.

The financial reports dataset guides every phase as shown in figure 1.2: calibration, pruning decisions, distillation objectives, and final specialization. The result is a model that is substantially smaller, noticeably faster, and more accurate for your specific use case than the original general-purpose model.

As already mentioned earlier, the pipeline isn't limited only to domain specialization. You can also use it to create more efficient models by strategically pruning less critical components while preserving capabilities that matter most for your use case, achieving computational savings while maintaining or even improving performance on your target tasks.

In this second scenario, the goal isn't to specialize but to optimize. You can skip using a specific dataset and apply pruning techniques based on general structural importance metrics. The result is a model that preserves most of the original behavior but runs more efficiently.

Another, more targeted path is to use a dataset from an established benchmark to guide optimization. For example, you could use BoolQ to improve the model's reading

comprehension, or IFEval to improve its instruction-following capabilities.

IMPORTANT

This flexibility lets you adapt the pipeline to your needs: specialize with domain data, optimize for maximum efficiency, or improve specific capabilities.

1.3 Toolkit and techniques

To run the book's examples, you will need software and hardware. We'll mention up front that if you've worked with LLMs, this stack will feel familiar.

The most important hardware component is the GPU. If possible, choose an NVIDIA GPU with CUDA support. It doesn’t matter whether it’s your own GPU or one you access through cloud services such as Google Colab or Kaggle.

The example notebooks included with the book have been tested in Google Colab and, whenever possible, on its free tier, so a T4 GPU should be sufficient. For more demanding chapters, we provide an additional notebook with smaller models and datasets, optimized for the free tier, so you can always follow along even without access to more powerful hardware.

In each notebook, you'll find an indication of the recommended GPU and a button to open it directly in Google Colab.

If you prefer to use your own GPU, a mid-range model will be enough in most cases. The key requirement is about 12 GB of GPU VRAM and CUDA support.

As for the software, you'll work primarily with PyTorch and the Hugging Face ecosystem. Hugging Face (https://huggingface.co) serves as the central hub for the AI community, a platform where thousands of pre-trained models are freely available and where core libraries like Transformers are actively developed. Beyond Hugging Face, you'll also use industry-standard evaluation libraries like lm-evaluation-harness (https://github.com/EleutherAI/lm-evaluation-harness/) to benchmark model performance and measure computational efficiency. Together, these tools will give you what you need to modify, optimize, and evaluate your models.

TIP

To use the notebooks with Hugging Face, set the secret key HF_TOKEN to your Hugging Face API key in the Google Colab environment. Some models used in this book are gated and require you to request access on their Hugging Face model page before downloading. Each chapter that uses a gated model includes a reminder with the direct link. Approval is usually granted within a few hours, but it's worth requesting access before you sit down to work.

In this book, we introduce OptiPFair, an open-source library I developed that implements and wraps the techniques taught here. It’s built from the code explained throughout the book, and we’ll use it to simplify the example notebooks. We’ll always start by teaching the “manual” process so you understand each technique in depth. Then, to keep the notebooks clear, we’ll use OptiPFair to encapsulate that complexity in code that isn’t part of the chapter. Because it’s an open-source project, you’re invited not only to report bugs, but also to contribute to its development.

All the notebooks in this book share a common set of configuration variables. The most important for controlling execution time are `BENCHMARK_LIMIT`, which caps the number of samples used in lm-eval evaluations, and `RECOVERY_SAMPLES`, which controls the size of the training set in knowledge recovery experiments. The flag `RUN_FULL_BENCHMARKS` lets you skip the evaluation suite entirely when you only want to test the training pipeline. Chapters that include fine-tuning also expose `EPOCHS`. The default values in each notebook reproduce the results shown in the book. Setting `BENCHMARK_LIMIT` to `100` instead of `None` reduces evaluation time significantly at the cost of less stable results.

Throughout the book, we'll work with a range of models, from classic ones such as DistilGPT to more advanced families (which we'll study in chapter 3), such as LLaMA, Gemma, SmolLM, and Qwen. We'll focus mainly on these modern model families.

The architectures used by these models represent the cutting edge of LLMs and share structural characteristics that make them ideal for rearchitecting techniques. We'll take a deep dive into model architectures in chapter 3.

As you'll see, we'll use well-known techniques and standard libraries to create efficient models.

1.4 Your LLM rearchitecture roadmap

The goal of this book is to teach you how to rearchitect LLMs, with a focus on creating more efficient and effective models adapted to your needs. Along the way, you'll gain a hands-on, architectural understanding of how modern LLMs work, transforming you from a model user into a model architect.

To master these rearchitecting techniques each technical chapter follows a structure in two or three phases, designed to take you from the fundamentals to the cutting edge of applied research:

1. *The fundamentals:* We'll start with the concepts behind each technique. Before we get into the code, we'll understand how the technique works, its fundamentals, and its usefulness.
2. *The practice:* Immediately after, we'll study the implementation using the example notebook, which contains all the necessary code.
3. *The deep dive*: Finally, whenever possible, we'll study the paper the implementation was based on. We'll examine the experiments and the changes we made to turn the paper's idea into a working code implementation.

You'll apply this methodology by working through a carefully designed progression that covers the entire rearchitecting pipeline.

The last part of the book goes beyond traditional optimization by opening a new dimension: analyzing internal activations to visualize how the model "thinks." This leads us to fairness pruning, where you'll apply your ability to identify behaviors and eliminate neurons to modify unwanted model behaviors without affecting performance.

This sequence is designed to guide you from fundamentals to the cutting edge of applied research. It's designed to turn you into an engineer capable of understanding and implementing the latest research in the field, and even creating new advances. We're not simply teaching you how to work with current structures; we're providing fundamental

knowledge that will be useful for studying and optimizing any future architecture.

The work also isn't limited to creating small language models. A fundamental part is understanding how LLM architectures work so you'll be able to create your own optimization techniques. These techniques can be applied both to small models and to the largest models of the moment, making them more efficient without compromising, or even improving, their performance in specific fields.

At the end of this journey, you won't just be competent in the most advanced optimization techniques. You'll have opened a door that remains closed to many: explainability in LLMs. The last part of the book will teach you to explore the model's internal activations, allowing you to visualize, diagnose, and modify its behavior.

This is the direction the cutting edge of AI research is moving in. In fact, the fundamental techniques you'll study grew out of research published by a mix of industry leaders and top-tier academic centers, such as NVIDIA, DeepSeek, the Technical University of Munich, Imperial College London, and NEC Labs. This book gives you the ability to understand and, more importantly, to *implement* that cutting-edge research.

I hope you’re looking forward to becoming an LLM architect.

1.5 Summary

- The use of oversized, generic LLMs can lead to high production costs, little differentiation from competitors, and no explainability of decisions.
- Models become more effective and efficient by adapting their architecture to a specific domain and task.

- The model-architecting process consists of three phases: structure optimization, knowledge recovery, and specialization.
- The domain-specific dataset is a key element and common thread throughout the process, ensuring each optimization and specialization phase aligns with the final objective.
- Knowledge distillation transfers capabilities from the original teacher model to the pruned student model, enabling the student to learn not only the correct answers but also the reasoning process that leads to them.
- Fine-tuning techniques such as LoRA allow domain specialization by training only a small number of parameters, drastically reducing cost and time.
- Modern architectures like LLaMA, Mistral, Gemma, and Qwen share structural traits that make them well suited to rearchitecting techniques.
- By mastering these techniques, developers can go from being model users to model architects.

2 An end-to-end rearchitecting project

This chapter covers

- Establishing baseline capab
- ilities and inference metrics
- Applying depth pruning to remove layers
- Measuring structural modification impacts
- Recovering knowledge through distillation
- Creating smaller, faster models

In the previous chapter, we introduced the rearchitecting pipeline and explained that the key differentiator in our approach to creating efficient solutions is rearchitecting the models.

In this chapter, you'll get hands-on by adapting a model. We'll treat this as an assignment, where we have a base model that does the work, and we need to reduce its size while preserving as many of its capabilities as possible.

NOTE

This is very common in the industry. For example, NVIDIA combines structural optimization (through pruning techniques) and knowledge recovery (using knowledge distillation) to create its model families. That way, it only needs to fully train the largest model in the family.

To create the model, we'll run two phases of the rearchitecting pipeline that we introduced in the previous chapter (see figure 1.2): Structural Optimization, where we'll modify the model's architecture to make it lighter and faster, and Knowledge Recovery, where we transfer knowledge from the base model to the new pruned model. Our focus here is general-purpose optimization. We're not optimizing the model for a specific domain or task, which is the third phase of the pipeline we'll explore in later chapters.

2.1 The rearchitecting workflow

You might be thinking, "How am I going to modify the structure of an architecture that I still don't understand well yet?" That's a normal feeling; if you've never worked with LLM structures, it can seem overwhelming. That's why we're going to work through a purely practical example to demystify the whole process.

The goal is for you to get familiar with the workflow before diving deep into theory. But if you'd like an introduction to the internal architecture of models, turn to chapter 3. There, you get the foundation to refine this process and tackle more complex optimizations than this chapter covers.

NOTE

We define model performance as the combination of two equally important dimensions: capabilities, the model's ability to reason and answer correctly (measured by benchmarks such as ARC or HellaSwag), and inference efficiency, the model's operational speed and resource usage (measured by tokens per second and latency).

To achieve this familiarity, we'll break down our approach into concrete steps. The workflow through the Structural Optimization and Knowledge Recovery phases, shown in figure 2.1, are:

1. *Establish a baseline.* We'll measure the performance of a standard model so we know our starting point.
2. *Apply pruning.* We'll perform the main intervention by removing a couple of complete layers from the model, a technique known as *depth pruning*, which we'll cover in detail in chapter 4.
3. *Evaluate the impact.* We'll quantify exactly what capabilities the model has lost after removing part of its structure.
4. *Recover the knowledge.* We'll use knowledge distillation to "teach" the smaller, pruned model to recover the skills it may have lost.
5. *Analyze the final result.* We'll compare the original model to our final version to show that we've created a more efficient model while sacrificing as few capabilities as possible.

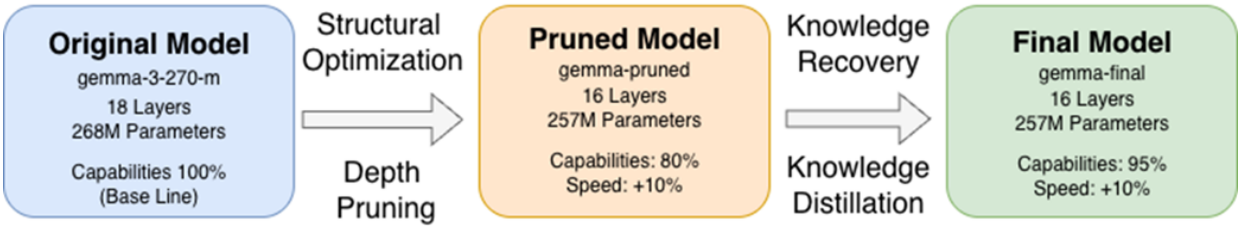


Figure 2.1 The rearchitecting cycle followed in this chapter. We start with an original model, make it smaller through pruning, and finally recover its capabilities through knowledge distillation.

For this first hands-on project, we've selected gemma-3-270m, Google's compact model designed for efficient AI applications. Its size makes it ideal for a T4 GPU, the smallest available on Google Colab's free tier.

2.2 Establishing the baseline

The goal of this step is to establish a performance baseline for the base model. Without a clear baseline, we can't tell whether the model has improved or worsened, or by how much.

NOTE

The hands-on work in this chapter is split across two notebooks. CH02_NB01_Depth_pruning_evaluation.ipynb covers sections 2.2 and 2.3: establishing the baseline and applying depth pruning. CH02_NB02_Knowledge_Recovery.ipynb covers sections 2.4 to 2.6: the distillation training and the final evaluation. Both notebooks are configured to replicate the results shown in this chapter, which makes them computationally heavy. If you want to explore the workflow without the full runtime, three variables control the trade-off: set BENCHMARK_LIMIT from None to 100 for faster benchmark evaluations in NB01, and reduce RECOVERY_SAMPLES and EPOCHS in NB02 to shorten the distillation training. Expect slightly different numbers from those in the tables, but the same directional conclusions.

We'll start by installing the required Python libraries. If you're working in Google Colab, many are already installed by default, but for consistency, we'll install everything in both the notebook and the code shown in the book.

We'll use pip to install `transformers` and `accelerate` (for working with models from Hugging Face), `torch` (the deep learning framework), `OptiPFair` (our support library for pruning), and `lm-eval` (for standardized evaluation):

```
!pip install -q \
    "transformers==4.55.4" \
    "accelerate==1.10.1" \
    "lm_eval==0.4.9.1" \
    "sentencepiece==0.2.1" \
    "sentence-transformers==5.1.0" \
    "optipfair==0.1.4"
```

INTRODUCTION TO OPTIPFAIR

Throughout this book, we'll occasionally use OptiPFair, a support library I developed to speed up certain processes. It's important to make clear that the main goal of this book is for you to learn the concepts and write the fundamental code for each technique yourself.

That said, OptiPFair isn't just a set of helpers for the book; it's a fully functional, open-source library designed to implement the pruning and bias analysis techniques taught here in real-world projects. Because it's an active project, I encourage you to explore its repository, report any problems you find, or even contribute your own improvements through pull requests.

In the context of the book, we'll use it as a supporting tool to simplify parts of the code that aren't the central focus of the chapter. This will let us concentrate on the key logic of each procedure without getting distracted by repetitive or auxiliary code.

Once the libraries are installed, we import the necessary modules and verify the availability of a GPU. This step is important because using a GPU will significantly accelerate the notebook execution.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from transformers import AutoModelForCausalLM, AutoTokenizer
from optipfair.pruning.depth import prune_model_depth
from lm_eval import evaluator
from lm_eval.models.huggingface import HFLM
import time
import json

device = torch.device('cuda' if torch.cuda.is_available() else 'cp
u')
print(f"Using device: {device}")
if torch.cuda.is_available():
    print(f"GPU: {torch.cuda.get_device_name(0)}")
```

The output of this block should be something like `GPU: Tesla T4`, or alternatively, the device that the notebook is using.

Due to the size of the chosen model, and because pruning doesn't require much computation, we could run it on a CPU. The steps that benefit most from a GPU are the inference tests and benchmark evaluations we run to establish the baseline and verify the model's behavior.

With all the libraries installed and classes imported, the next step is to load the model from the Hugging Face Hub using the popular Transformers library.

As we mentioned earlier, the model selected for this first contact is `google/gemma-3-270m`. We chose it for three practical reasons: its compact size (270M parameters) runs smoothly on a T4 GPU in Google Colab; it reflects current architectural trends (which we'll examine in detail in chapter 3); and it strikes a balance between being advanced enough to demonstrate real optimization techniques while remaining accessible for hands-on work.

NOTE

The Gemma model family has gated access on the Hugging Face Hub. To download it, you'll need to take two steps. First, visit the model page on the Hub and accept the license terms. Second, authenticate from your notebook environment. As mentioned in chapter 1, this involves configuring a secret key named HF_TOKEN with your Hugging Face User Access Token in Google Colab.

The following code loads the model and its tokenizer from Hugging Face.

```
MODEL_NAME = "google/gemma-3-270m"

model = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    device_map="auto"
)

tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
```

If you look at the previous code, we've identified the available device, but now, when loading the model, we're using `device_map="auto"`. This way, the `accelerate` library automatically optimizes model placement in memory, even dividing layers between the GPU and CPU if necessary. For gemma-3-270m this makes no practical difference since the model fits comfortably on a single T4, but the habit is worth keeping for when you switch to larger models.

Once the model is loaded into memory, the next step is to inspect its internal architecture. Before making any

modifications, we need to understand its size and structure. The following code performs two tasks:

```
original_params= sum(p.numel() for p in model.parameters())
print(f"Total parameters: {original_params}")
print(f"Original layers: {len(model.model.layers)}")
print("=" * 20)
print(model)
```

The output generated is shown in the following listing.

Listing 2.1 Gemma3 Model structure

```
Total parameters: 268098176 #A
Original layers: 18 #B
====================
Gemma3ForCausalLM(
    (model): Gemma3TextModel(
    (embed_tokens): Gemma3TextScaledWordEmbedding(262144, 640,
                                                  ➥padding_idx=0)
    (layers): ModuleList(
         (0-17): 18 x Gemma3DecoderLayer(
            (self_attn): Gemma3Attention(
              (q_proj): Linear(in_features=640, out_features=1024, b
ias=False)
              (k_proj): Linear(in_features=640, out_features=256, bi
as=False)
              (v_proj): Linear(in_features=640, out_features=256, bi
as=False)
              (o_proj): Linear(in_features=1024, out_features=640, b
ias=False)
              (q_norm): Gemma3RMSNorm((256,), eps=1e-06)
              (k_norm): Gemma3RMSNorm((256,), eps=1e-06)
            )
            (mlp): Gemma3MLP(
              (gate_proj): Linear(in_features=640, out_features=204
8, bias=False)
              (up_proj): Linear(in_features=640, out_features=2048,
bias=False)
              (down_proj): Linear(in_features=2048, out_features=64
0, bias=False)
              (act_fn): PytorchGELUTanh()
            )
            (input_layernorm): Gemma3RMSNorm((640,), eps=1e-06)
            (post_attention_layernorm): Gemma3RMSNorm((640,), eps=1e
-06)
            (pre_feedforward_layernorm): Gemma3RMSNorm((640,), eps=1
e-06)
            (post_feedforward_layernorm): Gemma3RMSNorm((640,), eps=
1e-06)
          )
        )
        (norm): Gemma3RMSNorm((640,), eps=1e-06)
        (rotary_emb): Gemma3RotaryEmbedding()
```

```
        (rotary_emb_local): Gemma3RotaryEmbedding()
      )
      (lm_head): Linear(in_features=640, out_features=262144, bias=F
alse)
)
```

#A Number of parameters.
#B Number of layers.

At this point, what matters most to us are the number of parameters and the number of layers. We can find both pieces of information in listing 2.1.

As you can see, the model, living up to its name, has 268 million parameters. What matters most right now is that it has 18 layers `(0-17): 18 x Gemma3DecoderLayer`, or transformer blocks. We'll see the technical distinction between a “layer” and a “block” in the next chapter. For now, we'll follow convention and keep calling them layers.

The first test is to get a simple response from the model: a sanity check to ensure it works as expected. To do this, we'll use a very simple prompt and let the model complete it. Because we'll repeat this test in the notebook and throughout the process with different models, I've encapsulated the code needed to get a response in a helper function called `generate_text`.

```
def generate_text(model, tokenizer, prompt , max_new_tokens):
    inputs = tokenizer(prompt, return_tensors='pt').to(device)
    with torch.no_grad():
        outputs = model.generate(
            inputs['input_ids'],
            attention_mask=inputs['attention_mask'],
            max_new_tokens=max_new_tokens,
            num_return_sequences=1,
            pad_token_id=tokenizer.pad_token_id,
            do_sample=False,
            num_beams=3,
            no_repeat_ngram_size=2
        )

    return tokenizer.decode(outputs[0], skip_special_tokens=True)
```

The parameters used in the call to `generate` were chosen with the intention of making the test deterministic; that is, the same result is returned across calls, while also ensuring that the generated text is rich and makes sense.

We've disabled random sampling (`do_sample=False`) and instead use beam search (`num_beams=3`). This method explores several probable sequences instead of just one, which produces notably more coherent, higher-quality text. To improve text quality, we also use `no_repeat_ngram_size=2`, which tells the model not to repeat the same two-token sequence consecutively.

To perform the test, we just have to call the function: `generate_text`.

```
MAX_NEW_TOKENS = 50
TEST_PROMPT = "Paris is the capital of"
generated_text = generate_text(
        model,
        tokenizer,
        TEST_PROMPT,
        MAX_NEW_TOKENS
)

print(f"Prompt: '{TEST_PROMPT}'")
print(f"Generated Text: '{generated_text}'")
```

THE RESPONSE GENERATED BY THE MODEL IS:

Paris is the capital of France. It is located in the middle of the country and has a population of about 10 million people. Paris is a city with a rich history and culture. The city is known for its beautiful architecture, art, and history. There are.

The response demonstrates coherent language generation and correct basic knowledge (Paris as France's capital). But like many LLMs, it contains factual inaccuracies about Paris's location and population. It's important to note that without specific adaptation, small models often struggle with precise factual recall in general domains. For a foundational model of this size, the overall structure and coherence are reasonable, providing a solid baseline for comparison.

Now that we've established that the model is working correctly, we'll assess its capabilities using four benchmarks that will give us a sense of its general behavior:

- *ARC-Easy:* Measures the model's reasoning and factual knowledge by answering school-level science questions. The model must choose the correct answer from four options, so random chance sits at 25%.

- *Winogrande:* Evaluates common sense by resolving pronoun ambiguity based on the context of a sentence. It presents two possible completions and asks the model to pick the right one, so random chance sits at 50%.
- *HellaSwag*: Evaluates common sense reasoning by asking the model to choose the most appropriate continuation, among four options, to complete an everyday situation or activity.
- *LAMBADA:* Evaluates the prediction of the last word of a text, requiring understanding of broad narrative context. Unlike the others, it is not multiple choice: the model must generate the correct word, making it a stricter test.

NOTE

The list of benchmarks could have been expanded with others, such as GSM8K to measure mathematical capability, MMLU for multidomain knowledge, or less well-known ones like BoolQ to measure reading comprehension through “yes or no” questions. Still, the four selected benchmarks are sufficient for the task we’ve set: controlling the model degradation produced during pruning and evaluating its subsequent recovery with knowledge distillation.

To evaluate our model against different benchmarks, we're going to use the `lm-eval` library, one of the most commonly used in model evaluation. Think of it as an automated testing harness for LLMs. Under the hood, it handles the complex, repetitive work of downloading a standard benchmark dataset (like ARC-Easy), formatting its questions into prompts, feeding them to our model, and calculating the final accuracy score based on the model's responses.

To simplify the interaction with `lm-eval`, we've created a helper function called `model_evaluation`.

NOTE

`model_evaluatio`n wraps lm-eval's `simple_evaluate` API and returns a flat dictionary of scores ready for comparison. Its full implementation is available in NB01, in the cell marked # Helper: model_evaluation.

This function takes our model and a list of tasks and returns the results in a clean, simple format:

```
benchmark_tasks = ['arc_easy', 'winogrande','hellaswag', 'lambada_op
enai']

baseline_results = model_evaluation(
    model,
    tokenizer,
    benchmark_tasks, limit=None
)
```

The results shown in this chapter use the full benchmark datasets, with `limit=None`. This produces reliable numbers but takes significantly longer to run on Colab hardware. If you need faster iterations while exploring the workflow, set LIMIT_EVAL to 100 in the notebook. Expect scores to vary by a few points from those in the tables, but the direction of change will be the same.

We take the returned results (arc_easy: 0.59, hellaswag: 0.34, lambada_openai: 0.43, and winogrande: 0.54) as the baseline against which we'll compare the models we're going to create in the process.

Finally, we just need a variable that tells us the model's inference efficiency. To avoid increasing the complexity of our baseline, we'll base it only on response time for a few test prompts.

As with the benchmarks, we've created a helper function called `measure_inference_time`. It runs a warm-up pass on the GPU followed by a series of timed prompts, returning the mean and standard deviation of the measured times. Its full implementation is available in NB01, in the cell marked `# Helper: measure_inference_time`.

NOTE

Inference times can change significantly with each run, depending on the environment where the notebook is executed. You can also find differences between different runs in the same environment due to the state of the GPU, especially in Colab, where we work with shared GPUs.

Table 2.1 summarizes all the benchmarks that form our baseline for the base model.

Table 2.1 Baseline benchmarks

model	arc_easy	hellaswag	lambada_openai	winogrande	inference
gemma-3-270m	0.59	0.34	0.43	0.54	4.611 ± 0.233

With our environment configured and a clear, quantified performance baseline, we're ready to take the scalpel and remove parts of the base model's structure. Note that Winogrande's baseline of 0.5375 is close to its chance level of 0.50, as the benchmark presents only two options.

2.3 Applying depth pruning

With the baseline now established, we're going to perform our first model restructuring. Because it's the first one, we've chosen a very direct, powerful, and simple approach: removing layers from the model to change its depth. That's precisely why it's called depth pruning.

As figure 2.2 illustrates, our objective is to go from a model with 18 layers to one with 16.

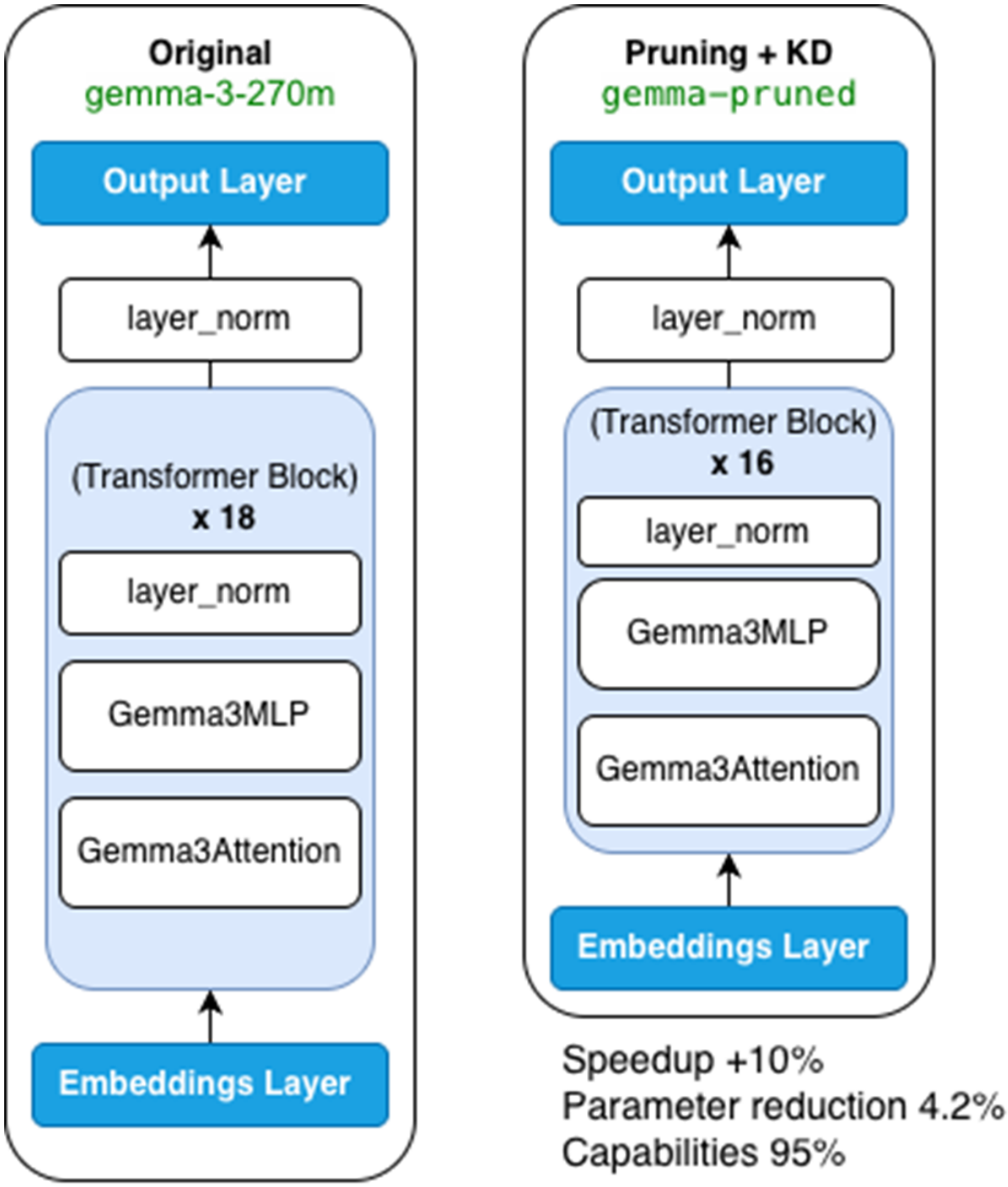


Figure 2.2 The original gemma-3-270m model is reduced from 18 to 16 layers. The capabilities results shown are achieved via the knowledge recovery phase.

But which layers do we remove? The first ones, the middle ones, the last ones? This is the central question in depth

pruning, and the answer often requires a detailed analysis of the model that we'll explore in later chapters.

For our first example, we'll adopt a direct approach and focus on the mechanics of pruning. We'll eliminate the last two transformer blocks from the model. A starting hypothesis could be that the final layers are highly specialized in refining the nuances of the output, and by eliminating them, we could preserve the model's more general knowledge.

This approach serves as an excellent, simple starting point for learning the process. It's important to highlight that this is not necessarily the optimal strategy. In chapter 4, we'll develop techniques to identify which layers are ideal candidates for pruning and thus achieve the best balance between capabilities and inference efficiency. For now, let's focus on the "how" before mastering the "which."

As we can see in listing 2.1, our model is composed by 18 layers and 268 million parameters; After pruning the resulting model should have 16 layers.

Our first step is to create a separate version of the model. We need to keep the original model intact because we'll need both the original and the modified version during the knowledge recovery phase. Let's create a copy for our pruning modifications:

```
import copy
pruned_model = copy.deepcopy(model)
```

We can now start working with the model we obtained. The layer elimination process is a very simple method: we keep only the first 16 layers (indices 0-15) of the original 18, effectively eliminating the final layers 16 and 17.

In the vast majority of modern models, especially decoder architectures like Llama, Gemma, or Mistral, this list of layers is found in the `model.layers` attribute of the main model object. What's most important for us is that this component, `pruned_model.model.layers`, is a special PyTorch container that behaves exactly like a Python list. Because it's a list, we can do things like check its length or create a shorter version.

With this first step, we create a Python list that contains only the layers we want to keep:

```
LAYERS_TO_REMOVE = 2
original_layers_count = len(pruned_model.model.layers)
new_layers_count = original_layers_count - LAYERS_TO_REMOVE
new_layers = pruned_model.model.layers[:new_layers_count]
```

The next step is to insert this new list into the model and update its configuration:

```
pruned_model.model.layers = nn.ModuleList(new_layers)
pruned_model.config.num_hidden_layers = len(pruned_model.model.layer
s)
```

This second block uses PyTorch’s `nn` module, which we imported at the beginning of the notebook. We can’t assign the list directly; we must wrap it in `nn.ModuleList` so it registers each layer as a submodule of our model. This is crucial for PyTorch to recognize those parameters, train them, and move them correctly to the GPU.

We also update `pruned_model.config.num_hidden_layers` so that the internal configuration matches the new structure.

The model's `.config` object is like its internal "instruction manual": it contains information about the model's architecture (number of layers, embedding size, or number of attention heads, for example) that PyTorch and the

`transformers` library consult. Keeping this configuration updated prevents errors when the model tries to use information that no longer corresponds to its actual structure.

It might seem surprising, but that's it! You've already performed your first model rearchitecting. In the notebook, you can find the results. We now have a model with 16 layers formed by 256 million parameters—a size reduction of 4.16%.

You might notice that removing two layers from 18 (11%) results in only a 4.16% parameter reduction. This happens because the model has other substantial components beyond the transformer layers. Notice in listing 2.1 the `embed_tokens` layer at the beginning and `lm_head` at the end. These also contain many parameters and remain unchanged when we remove transformer layers. Beyond static parameter count, removing layers also reduces the memory that grows with context length during inference, something we will explore in the following chapters.

With the new model created, we can examine its structure. The only difference from the base model is in this line:

```
(0-15): 16 x Gemma3DecoderLayer
```

You can see in listing 2.1 that our original model contains 18 Gemma3DecoderLayer blocks, compared with 16 in the pruned model.

The next step is to see how this structural modification affects its behavior and performance.

2.4 Evaluating the impact of pruning

Now is the time to see everything we've lost and gained by eliminating the last two layers of the model. It's worth remembering that our only action has been to lighten it by removing two layers; we haven't taken any adaptation or recovery action.

The first check will be the most intuitive: let's verify that our model continues generating coherent text. It's a simple sanity check using the same prompt we used previously with the base model:

PM

Paris is the capital of

Paris is the capital of France and one of the largest cities in Europe. It occupies approximately 2.5 million hectares of land surrounded by mountains and forests. Parisians love to travel abroad because they enjoy sightseeing tours abroad. Tourists visiting Paris visit museums, monuments, theaters,

It's clear the model continues to generate coherent text. We could even say that, if we didn't compare it to the base model's response, it would seem like a normal response for an SLM. But there are clear signs of degradation.

Although the model still knows that Paris is the capital of France, its factual knowledge has been compromised, leading it to generate hallucinations like stating that the city covers 2.5 million hectares or that it is surrounded by mountains. Additionally, as we see in the sentence about the

travels of Parisians, the model wanders and loses focus more easily than the original model. Now that we have a qualitative idea of this degradation, let's move on to quantifying the loss of knowledge by running the same battery of benchmarks we used for the base model.

```
pruned_results = model_evaluation(
    pruned_model,
    tokenizer,
    benchmark_tasks,
    limit=None
)
```

The results across the different benchmarks confirm our suspicion: the model has degraded in capabilities. Clearly, the obtained results, as can be seen in table 2.2, are worse than the baseline. We've created a model that's 4.16% smaller and 10% faster, but we've lost between 8 and 12 percentage points across the different benchmarks

Table 2.2 Pruned vs baseline benchmarks

Metric / Model	gemma-3-270m	gemma-pruned	Change
Parameters	268,098,176	256,950,912	-4.16%
Inference Time	5.124s	4.611s	+10 % faster
arc_easy	0.59	0.49	-16.55%
winogrande	0.54	0.54	0%
hellaswag	0.34	0.31	-9.60%
lambada_openai	0.43	0.36	-16.58%

It’s not all bad news, though: reducing the model size, and especially the number of layers, should bring an improvement in inference time. Winogrande moves from 0.538 to 0.541. This is not a sign that the model retained its reasoning intact. With only two options, random chance sits at 0.500. Being this close to the baseline, the model has little room left to degrade further.

```
speed_test_prompts = [
   "The capital of France is",
   "Machine learning is",
   "Climate change refers to"
]
pruned_timing = measure_inference_time(
    pruned_model,
    tokenizer,
    speed_test_prompts
)
```

The tests confirm the improvement: the inference time dropped to an average of 4.181 seconds, confirming that our model is now faster.

Our next step will be to recover the lost knowledge. We'll do it without touching the model's architecture, so we'll keep the speed gain. Instead, we'll adjust the existing weights to "teach" our smaller model to reason like its original, larger version again.

2.5 Recovering knowledge

So far we've modified the structure of a base model to increase its efficiency and get a smaller, faster model. But this transformation came with a considerable cost in its capabilities. Now, we'll tackle the final and most rewarding step of our first re-architecture cycle: recovering the lost knowledge.

For this phase, we'll move to a new notebook, CH02_NB02_Knowledge_Recovery.ipynb. The reason for this separation is purely practical: while pruning is a relatively light structural operation that could even be done on a CPU, the knowledge recovery process involves training, a process that requires much more computing power and should be done on a GPU.

NOTE

In this second notebook, you'll find the results of several experiments we've run while developing the model in this chapter. Two layers were removed, and the model was recovered by training on 15,000 rows from the SlimPajama dataset.

We're going to recover the knowledge through knowledge distillation (KD), where one model will act as the teacher and the other as the student.

- *Teacher model:* This will be our original model, gemma-3-270m, with its 18 layers. It’s the expert who holds all the knowledge to be transferred to the student.
- *Student model:* This will be our 16-layer pruned model. It's faster and lighter, but it needs to recover the skills it lost.

The goal is for the Student to learn to imitate the Teacher's full prediction distribution, not just its final answer, but how confident it was about other possibilities. By matching these probability patterns, the Student effectively mimics the Teacher's reasoning process.

To do this, we’ll load the teacher model the same way as in the previous section. To create the pruned model, we’ll switch to using our support library, `OptiPFair`.

In section 2.2, you learned how to remove layers manually by directly manipulating the model's `ModuleList`. The `prune_model` function encapsulates that same process but adds helpful features like progress tracking, making our workflow cleaner and more robust. The key parameters are:

- `model=copy.deepcopy(teacher_model)`: We pass a copy of our teacher model, which will be transformed into the student.
- `pruning_type="DEPTH"`: Specifies that we want to remove entire layers.
- `num_layers_to_remove`: Sets how many layers to remove.
- `layer_selection_method="last"`: removes layers from the end of the model, following our earlier strategy.

Let's see it in action:

```
teacher_model = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    device_map="auto"
)
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)

student_model = prune_model(  #A
    model=copy.deepcopy(teacher_model),  #A
    pruning_type="DEPTH",  #A
    num_layers_to_remove=LAYERS_TO_REMOVE,  #A
    layer_selection_method="last",  #A
    show_progress=True  #A
)
```

#A Apply depth pruning using OptiPFair. This function encapsulates the same manual layer removal process we implemented in Section 2.2, but with additional features like progress tracking and error handling.

For the Teacher to teach the Student, we need to establish the subjects that will be studied. In this case, because we want to create a generalist model and we're concerned about all the benchmarks in general, we'll use a subset of the SlimPajama dataset, We use SlimPajama, a large general-purpose dataset that has become a standard reference for this type of experiment. Its coverage of topics and writing styles makes it a neutral baseline for recovering general language capability:

```
from datasets import load_dataset
dataset = load_dataset(
   "DKYoon/SlimPajama-6B",
   split="train",
   streaming=True #A
)

RECOVERY_SAMPLES = 15000 #B
distillation_dataset = dataset.take(RECOVERY_SAMPLES)
```

#A Load dataset in streaming mode for efficiency
#B Reduce the number of samples to reduce time computing

With our models and the raw text dataset ready, we have one final preparatory step before KD. As you can see in the accompanying notebook, we need to transform our raw text into numerical inputs organized into small groups called *batches*. This process is fundamental for any PyTorch training, and we'll build it as a three-step data pipeline.

The first step is tokenization. We need a function that takes our text and converts it into the numerical tensors the model understands (input_ids, attention_mask, and labels). We'll define this function and then apply it to the rows we've recovered from our dataset:

```
def tokenize_for_kd(examples, max_length=128):
    texts = [ex["text"] for ex in examples]
    tokenized = tokenizer(
        texts,
        padding="max_length",
        truncation=True,
        max_length=max_length,
        return_tensors="pt"
    )
    return {
        "input_ids": tokenized["input_ids"],
        "attention_mask": tokenized["attention_mask"],
        "labels": tokenized["input_ids"].clone()
    }

# Process the raw text samples
recovery_samples = list(distillation_dataset) #1
tokenized_data = tokenize_for_kd(recovery_samples)
```

Although we set `streaming=True` to avoid downloading the full corpus, calling `list()` #1 on the result is intentional: by then, `.take(RECOVERY_SAMPLES)` has already limited the stream to 15,000 samples, so only that subset is materialized in memory.

Now that the data is tokenized, the second step is to organize it into a format that PyTorch can use for training. The way to do this is by creating a `Dataset` class.

Here is the class definition:

```
class KDDataset(torch.utils.data.Dataset):
    def __init__(self, tokenized_data):
        self.input_ids = tokenized_data["input_ids"]
        self.attention_mask = tokenized_data["attention_mask"]
        self.labels = tokenized_data["labels"]

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, idx):
        return {
            'input_ids': self.input_ids[idx],
            'attention_mask': self.attention_mask[idx],
            'labels': self.labels[idx]
        }

kd_dataset = KDDataset(tokenized_data)
```

Finally, with our data structured in a `Dataset` object, the last step is to create a `DataLoader`. This utility will automatically shuffle the data and group it into batches, which is exactly what is necessary for our training loop.

Let's create our dataloader with a batch size of 8:

```
kd_dataloader = DataLoader(
    kd_dataset,
    batch_size=8, #A
    shuffle=True
)
```

#A Small batch size due to memory constraints.

Finally, with our data structured in a Dataset object, the last step of our pipeline is to create a DataLoader. This PyTorch utility automatically shuffles the data and groups it into batches, which is exactly what we need to feed our training loop. With our data pipeline now complete, we can focus on the training strategy itself.

The core idea of knowledge distillation is simple: we don't just care about *what* the *Student* model answers, but *how* it gets there. We want it to learn the *Teacher's* reasoning process, not just its final output.

So, how do we measure the difference between the Teacher’s output distribution and the Student’s? This is where a key concept from information theory comes in handy: Kullback-Leibler divergence, or KL divergence.

You can think of KL Divergence as a way to measure the "surprise" or difference between two probability distributions. If the Student’s predicted probabilities are very similar to the Teacher’s, the KL Divergence is low (no surprises). If they are very different, the value is high. Our goal during distillation is to train the Student by adjusting its weights to minimize this KL Divergence, forcing the Student's probability distributions to align more closely with the Teacher's. The process is a continuous cycle, as shown in figure 2.3.

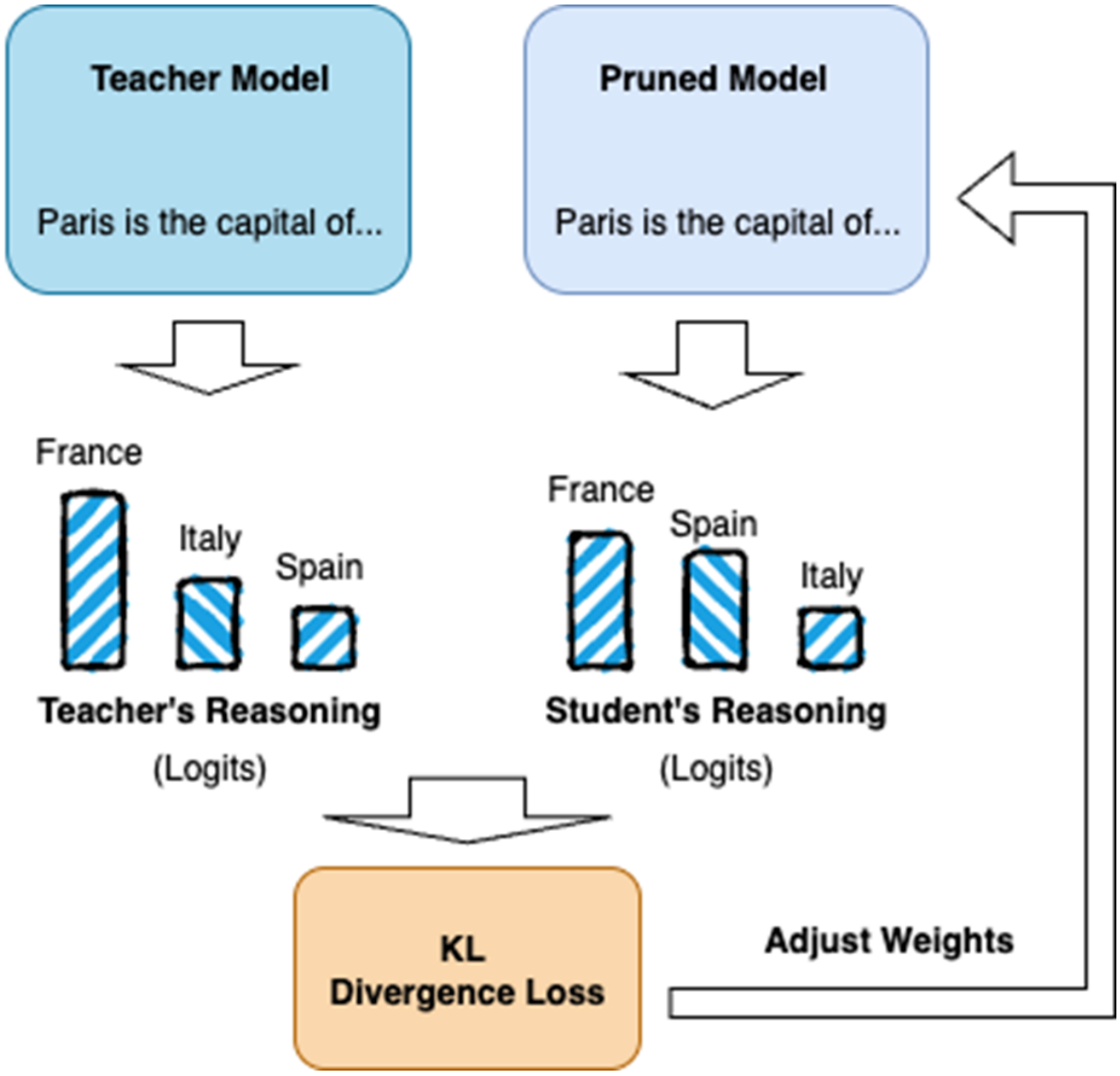


Figure 2.3 In the knowledge distillation cycle, the Teacher model generates its reasoning (logits) for a given text. The Pruned model (our student) does the same. The KL Divergence loss function measures the difference between both "reasonings," and this error is used to adjust the student's weights, making it think more like the Teacher over time.

In this first example, we use a straightforward approach: logits-only knowledge distillation where the Student learns to match the Teacher's probability distributions directly.

In the following listings, we implement this process. Think of it as a teaching cycle that repeats once per batch of data.

Although it may look complex, it's really just a matter of repeating four steps:

1. Select the lesson: Take a piece of text from our dataset.
2. Get the teacher's answer: Ask the Teacher to produce their logits (its "probability distribution") for that text.
3. Get the student's attempt: Ask the Student to do the same, producing their logits.
4. Correct the attempt: Compute the difference between the two distributions, and adjust the students' weights to minimize it.

The first step of training the model is taking a batch of examples from our DataLoader.

Listing 2.2 Step 1: Selecting the lesson

```
input_ids = batch['input_ids'].to(device)
attention_mask = batch['attention_mask'].to(device)
```

Each example in the batch mainly contains two tensors: `input_ids` and `attention_mask`. The `input_ids` are the numerical representation of our text, where each token, which can be a word or part of it, has been converted into a unique number from the model's vocabulary. The `attention_mask` is a list of 1s and 0s that tells the model which tokens are part of real words (1) and which are filler or padding (0), so it doesn't pay attention to them. We make sure that both tensors are on the same device, in this case, the GPU, ready to be processed.

Step 2 is asking the model to generate its reasoning.

Listing 2.3 Step 2: Getting the teacher's answer(logits)

```
with torch.no_grad():
    teacher_outputs = teacher_model(
        input_ids=input_ids,
        attention_mask=attention_mask
    )
    teacher_logits = teacher_outputs.logits
```

The code in listing 2.3 passes the data prepared in listing 2.2 through the Teacher model to obtain its logits.

This step runs inside a `torch.no_grad()` block to tell PyTorch not to calculate gradients for the Teacher. This means we don't want to train the Teacher model; we'll see more details in Chapter 6 on Knowledge Distillation. By disabling gradient calculation, we achieve two very important things: the operation consumes much less GPU memory and is significantly faster, since we skip all the calculations related to training.

Now that we have the Teacher's logits, the following listing adds the student's logits.

Listing 2.4 Step 3: Getting the student's attempt (logits)

```
student_outputs = student_model(
        input_ids=input_ids,
        attention_mask=attention_mask
    )
student_logits = student_outputs.logits
```

The code in listing 2.4 is very similar to the one in listing 2.3, where the teacher's logits are retrieved, changing the model, but with one big difference: we don't have a `torch.no_grad()` block. This way, PyTorch calculates the student's gradients and we can train it.

With the logits of both models calculated, we move on to step 4, correcting the attempt.

Before the training loop starts, we need to create the optimizer, the component that handles the mathematical weight adjustments during training. Think of it as the "learning engine": it takes the error we calculate at each step and determines how to nudge each weight in the right direction. We use AdamW, a standard choice in LLM training, with a small learning rate that controls how large those nudges are:

```
optimizer = AdamW(student_model.parameters(), lr=1e-5)
```

Listing 2.5 shows the process of calculating the difference between the models and applying the corresponding weight adjustments.

Listing 2.5 Step 4: Correcting the attempt

```
loss = F.kl_div(
   F.log_softmax(student_logits, dim=-1),
   F.softmax(teacher_logits, dim=-1),
   reduction='batchmean'
)
loss.backward()      #A
optimizer.step()     #B
optimizer.zero_grad()  #C
```

#A Calculate how to adjust the weights
#B Apply the adjustments using the optimizer
#C Reset for the next iteration.

The code calculates the "distance," or difference, between the Teacher's and Student's reasoning (logits) using a loss function specific to this task: `F.kl_div`. This function belongs to PyTorch's functional module, which we imported at the beginning of the notebook and which contains a wide range of loss functions. We then use that "error score" to adjust the Student's weights in the right direction, making it a bit more like the Teacher. In chapter 6, we'll go into more detail on the entire KD loop.

This set of operations, described in listings 2.2 to 2.5, forms the heart of our training loop. The loop repeats for a limited number of epochs, that is, multiple full passes over the entire dataset.

After a few epochs, the Student should have internalized much of the Teacher's knowledge, mimicking its way of reasoning.

Once the knowledge distillation process is finished, it's time to see whether the Student model has been able to recover the lost capabilities by learning the Teacher model's reasoning. We'll analyze this in the next section by subjecting the recovered model to the same benchmarks as before.

2.6 Analyzing the final result

After applying knowledge distillation, it's time for the moment of truth. We run the same evaluations as in the previous sections to measure how much our "student" model has recovered.

The results, summarized in table 2.3, give us the final verdict on our first re-architecture cycle.

Table 2.3 (Knowledge recovery analysis)

Metric / Model	gemma-3-270m	gemma-pruned	Student (Post-KD)	Retained capabilities before /after KD
Parameters	268,098,176	256,950,912	256,950,912	-4.16
Inference Time	5.12s	4.61s	4.61s	10% Faster
arc_easy	0.59	0.49	0.54	83.1 / 91.5 %
winogrande	0.54	0.54	0.55	100 / 102 %
hellaswag	0.34	0.31	0.33	91.2 / 97.6 %
lambada_openai	0.43	0.36	0.38	83.7 / 87.2%

Honestly, for a process that used only a portion of the available data from a standard general-knowledge dataset, the results are quite good: We have a 10% faster and 4.16% smaller model that recovered between 87% and 98% of the original model's capabilities on all the controlled benchmarks.

One final test is missing: seeing what our new model thinks about Paris.

```
Paris is the capital of France and one of the most famous cities in the world. It is known for its beautiful architecture, rich culture, and vibrant nightlife. The Paris Fashion Week is an annual event tha t showcases the latest fashion trends and styles from across the glo be. This event
```

Unfortunately for Parisians, the city has lost the mountains that the pruned model assigned to it, and now the recovered model's response is otherwise factual and logical, on par with what the base model produced.

2.7 Hands-on lab

Now it's your turn. In the notebooks that accompany this chapter, you'll find the results of different experiments, in

which three or even four layers have been removed.

I encourage you to use the notebook as your own laboratory. Don't limit yourself to running the code; modify it to build a solid intuition about the trade-offs in model optimization.

Here are several ideas to get started:

- Vary the number of layers to prune:
  - What to do: Modify the `LAYERS_TO_REMOVE` parameter. Try removing four, six, or even more layers.
  - Questions to answer: How much capability does the model lose with more aggressive pruning? What percentage of that capability can you recover with the same amount of distillation data?
- Adjust the recovery dataset:
  - What to do: Change the number of samples used for distillation in the `RECOVERY_SAMPLES` parameter.
  - Questions to answer: What happens if you use only 5,000 samples instead of 15,000? And what if you use 30,000? Is there a point where adding more data no longer significantly improves the result?
- Change the layer selection strategy:
  - What to do: By default, the notebook removes later layers. You can investigate two variants: 1) Remove the first layers, by changing `layer_selection_method` to `"first"`. 2) Remove specific and non-adjacent layers, by passing a list of indices directly to the method.
  - Code example:

```
student_model = prune_model(
    model=copy.deepcopy(teacher_model),
    pruning_type="DEPTH",
    layer_indices=[2, 5, 8],
    show_progress=True
)
```

-
    - Questions to answer: Is the impact on the model's knowledge more drastic when removing the first layers compared to the middle or final ones?
- Specialize the model with a different dataset (Advanced):
    - What to do: Replace the distillation dataset with one that's closely aligned with a particular benchmark.
    - Questions to answer: Can you exceed the recovery percentage on the `lambada_openai` benchmark if you "train" the student with data that favors that specific task? How does this affect the results on the other benchmarks?

We've completed the cycle: structural optimization followed by knowledge recovery. You've taken a standard model, made it more efficient at the architectural level, and cured the side effects to create a custom solution, demonstrating that we don't always need bigger models, but smarter ones.

2.8 Summary

- Establishing a quantitative baseline is an indispensable first step in any optimization project because it provides an objective metric against which the success or failure of interventions is measured.
- Depth pruning removes entire Transformer layers to reduce model size and speed up inference, typically

targeting the final layers that hold refinement details rather than core knowledge.

- Knowledge distillation lets smaller “student” models recover lost capabilities by learning to mimic the reasoning patterns of larger “teacher” models.
- Combining structural pruning and knowledge distillation lets you build more efficient models that retain most of the capabilities of their larger, original versions.

3 A blueprint to modern transformers

This chapter covers

- Classical architecture with DistilGPT
- Modern optimizations: GQA and GLU
- Mapping optimizations to model components

In the previous chapter, you completed your first re-architecture project: you removed layers from a Transformer model and recovered its capabilities, through knowledge distillation, from the base model. The result was a completely new model based on a Gemma-3 model.

In this chapter, we'll study the anatomy of modern Transformers, from classic to the most advanced architectures. We'll identify the specific components that you'll optimize in the upcoming chapters, and with this, you'll gain the fundamental basis to decide which re-architecture technique to use in each project. The concepts in this chapter are the cornerstone for understanding the optimization techniques you'll apply in the following chapters.

When you load a model for inference, you could expect memory consumption to match the model size. But the component responsible for context, the attention mechanism, creates a cache that grows with each token you process, often exhausting your GPU memory before finishing a single document. Meanwhile, the MLP (Multi-Layer Perceptron), the block responsible for processing knowledge, consumes most of the model's parameters and compute

time. Understanding where these bottlenecks live, both in memory and compute, and how modern models have evolved to address them is crucial information you'll use to decide what type of rearchitecting you need and how to carry it out.

We'll start with DistilGPT2 to study the classic architecture of Transformers. We'll continue with the evolution toward modern architectures like Llama, where we'll see the modifications adopted to improve both the attention and the processing block.

By the end of the chapter, you'll have a clear map of which technique to apply to each specific component.

3.1 Classical architecture: DistilGPT2

We've selected DistilGPT2 as a starting point to understand the classic architecture of Transformers. It's a very light, 82-million-parameter model that preserves the fundamental structure that gave rise to the language model revolution, but it doesn't contain the modern evolutions that we'll see with the following models.

It'll help us learn the common parts of Transformers, explain from the ground up how they work, and see that there are optimization methods we can use on all models, regardless of their architecture.

NOTE

All the code for this chapter can be found in the notebook CH03_NB01_MODEL_structures.ipynb. There you will also find structures of models not discussed in this chapter, such as Microsoft Phi.

Let's start by taking a look at the model, and to do that, we'll load it into memory and print its structure:

```
model = AutoModelForCausalLM.from_pretrained(
        "distilbert/distilgpt2",
        torch_dtype="auto",
        device_map="cpu",
    )
print(model)
```

By running the code we get the model's structure. This structure can be divided into three main sections: input, the set of Transformer blocks, and output.

Listing 3.1 DistilGPT2 structure

```
GPT2LMHeadModel(
    (transformer): GPT2Model(
        (wte): Embedding(50257, 768) #A
        (wpe): Embedding(1024, 768) #B
        (drop): Dropout(p=0.1, inplace=False)
        (h): ModuleList(
            (0-5): 6 x GPT2Block( #C
                (ln_1): LayerNorm((768,), eps=1e-05,
                            ➥elementwise_affine=True)
                (attn): GPT2Attention(
                    (c_attn): Conv1D(nf=2304, nx=768)
                    (c_proj): Conv1D(nf=768, nx=768)
                    (attn_dropout): Dropout(p=0.1, inplace=False)
                    (resid_dropout): Dropout(p=0.1, inplace=False)
                )
                (ln_2): LayerNorm((768,), eps=1e-05,
                             ➥elementwise_affine=True)
                (mlp): GPT2MLP(
                    (c_fc): Conv1D(nf=3072, nx=768)
                    (c_proj): Conv1D(nf=768, nx=3072)
                    (act): NewGELUActivation()
                    (dropout): Dropout(p=0.1, inplace=False)
                )
            )
        )
        (ln_f): LayerNorm((768,), eps=1e-05, elementwise_affine=Tru
e)
    )
    (lm_head): Linear(in_features=768, out_features=50257, bias=Fals
e)
)
```

#A 50257 Model's vocabulary size. 768 Vector size to represent the information.
#B 1024 is the context window.
#C The model has 6 Transformer blocks.

The structure in listing 3.1 is a map of all the model's components, but it's a static map. To understand how it comes to life, we must remember that a PyTorch model is

nothing more than a Python class. This class has two fundamental methods that define its behavior: `__init__()` and `forward()`.

The `__init__()` method acts as the "constructor" where all the layers the model will use are declared and initialized. By inspecting the `GPT2Model` source code, we find this constructor:

```
   def __init__(self, config):
        super().__init__(config)

        self.embed_dim = config.hidden_size

        self.wte = nn.Embedding(config.vocab_size,
                                                ➥ self.embed_d
im) #A
        self.wpe = nn.Embedding(config.max_position_embeddings,
➥ self.embed_dim) #A

        self.drop = nn.Dropout(config.embd_pdrop) #B
        self.h = nn.ModuleList([GPT2Block(config, layer_idx=i)
➥ for i in range(config.num_hidden_layers)]) #C
        self.ln_f = nn.LayerNorm(self.embed_dim,
➥ eps=config.layer_norm_epsilon) #D
```

#A The embedding layers correspond to wte and wpe.
#B The dropout layer corresponding to drop.
#C The six Transformer blocks corresponding to 6 x GPT2Block.
#D The normalization layer ln_f.

As you can see, this section of the code corresponds to the model's structure in listing 3.1. The `lm_head` layer, visible as the last line of that listing, is absent from this `init()` because the code shown belongs to `GPT2Model`, the base class. The `lm_head` lives one level up, in `GPT2LMHeadModel`, the Hugging Face wrapper that adds the language modeling head on top. We'll see what it does when we complete the data flow analysis in the next section.

With this distinction in mind, we can now start to analyze the main modules of DistilGPT2.

3.1.1 General behaviour of a Transformer model

As we've seen, the model's components are defined in the `__init__()` method. Their execution order is decided in the `forward()` method, which controls the flow of the whole process.

The main `forward()`, the one in the `GPT2Model` class, defines the high-level data flow: it takes the input embeddings, passes them through the list of Transformer blocks (h), and finally applies the final normalization layer (ln_f). However, it doesn't handle the details of each stage.

There's more than one `forward()` method. Each complex module we saw in the `__init__()`, like `GPT2Block`, has its own `forward()` method. The process is controlled by a chain of nested calls:

1. The `GPT2Model`'s `forward()` is executed.
2. When it reaches the Transformer blocks, it iterates over the list `(h)` and sequentially calls the `forward()` method of each `GPT2Block`.
3. In turn, the `GPT2Block`'s `forward()` calls the `forward()` of its own submodules: first the one for the Attention Module `(GPT2Attention)` and then the one for the MLP Module `GPT2MLP`.

Later in this chapter, we'll inspect the `forward()` method of the MLP Module. By doing so, we'll discover the exact order of its internal operations. Information that isn't visible in the general structure, since they've been defined in a different order in the `init()` method.

Now that we understand this execution hierarchy, let's follow the data flow from the beginning. Let's start with the input block, made up of the embedding and dropout layers, whose flow can be seen in figure 3.1.

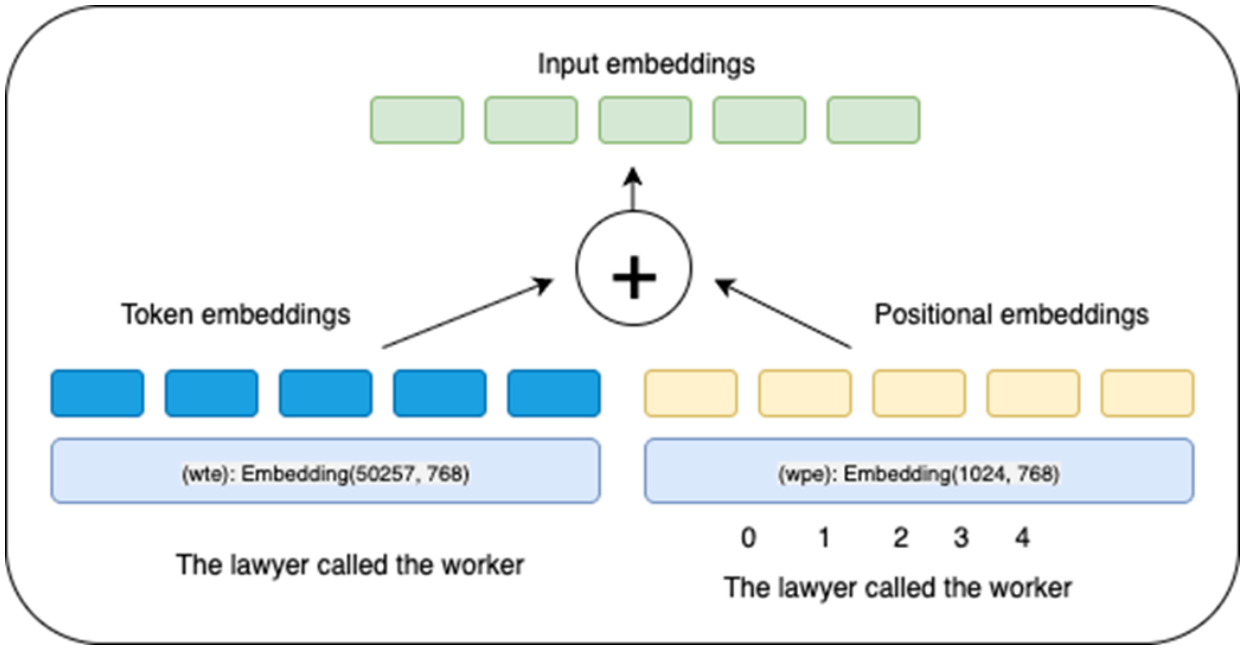


Figure 3.1 Embedding creation process in the input block. The `wte` layer converts each text token into a Token Embedding based on its meaning. The `wpe` layer generates a Positional Embedding based on the token's position in the sequence. The combination of both produces a vector that contains both semantic and positional information.

The `wte` and `wpe` layers are the entry point to the model. The first one, `wte` (*Word Token Embedding*), is in charge of converting token identifiers into vectors, generating what is technically known as the token embedding. The process begins when a Tokenizer external to the model converts each token from the input text into a unique ID. This layer `wte` receives these IDs (from 0 to 50256) and acts like a large dictionary that associates a vector with each ID.

The first number we find in `wte`: 50257, represents the vocabulary size (the total number of available IDs), while the second: 768, tells us the size of that vector. This means that

at the output of this layer we have a 768-dimension embedding representing each of the input tokens.

The second layer, `wpe` (*Positional Embedding*), is responsible for encoding the position of each token within the sequence. For each possible position (from 0 to 1023), this layer generates a unique 768-dimension vector. The limit of 1024 positions marks the context window of DistilGPT2, the maximum number of tokens the model can process at once. This vector, known as the *positional embedding,* is then added directly to the token embedding from the previous step. This vector addition is possible because both vectors share the same dimensionality.

The result is the input embedding, which contains information about both the token and its position in the sentence, since "The worker called the lawyer" has a very different meaning from "The lawyer called the worker.”

Right after, the Dropout layer acts as a regularization mechanism during training, randomly "turning off" some neurons to prevent the model from becoming too dependent on specific patterns, which is known as *overfitting.* At inference time it becomes inactive.

Finally, the generated input embeddings are passed to the heart of the model: the set of Transformer blocks, which will be the focus of our optimization techniques. The line `(0-5): 6 x GPT2Block` tells us that we have six stacked Transformer blocks, each one processing and refining the information it receives from the previous one.

Figure 3.2 offers a view of this structure, showing both the general architecture and the internal detail of a single block.

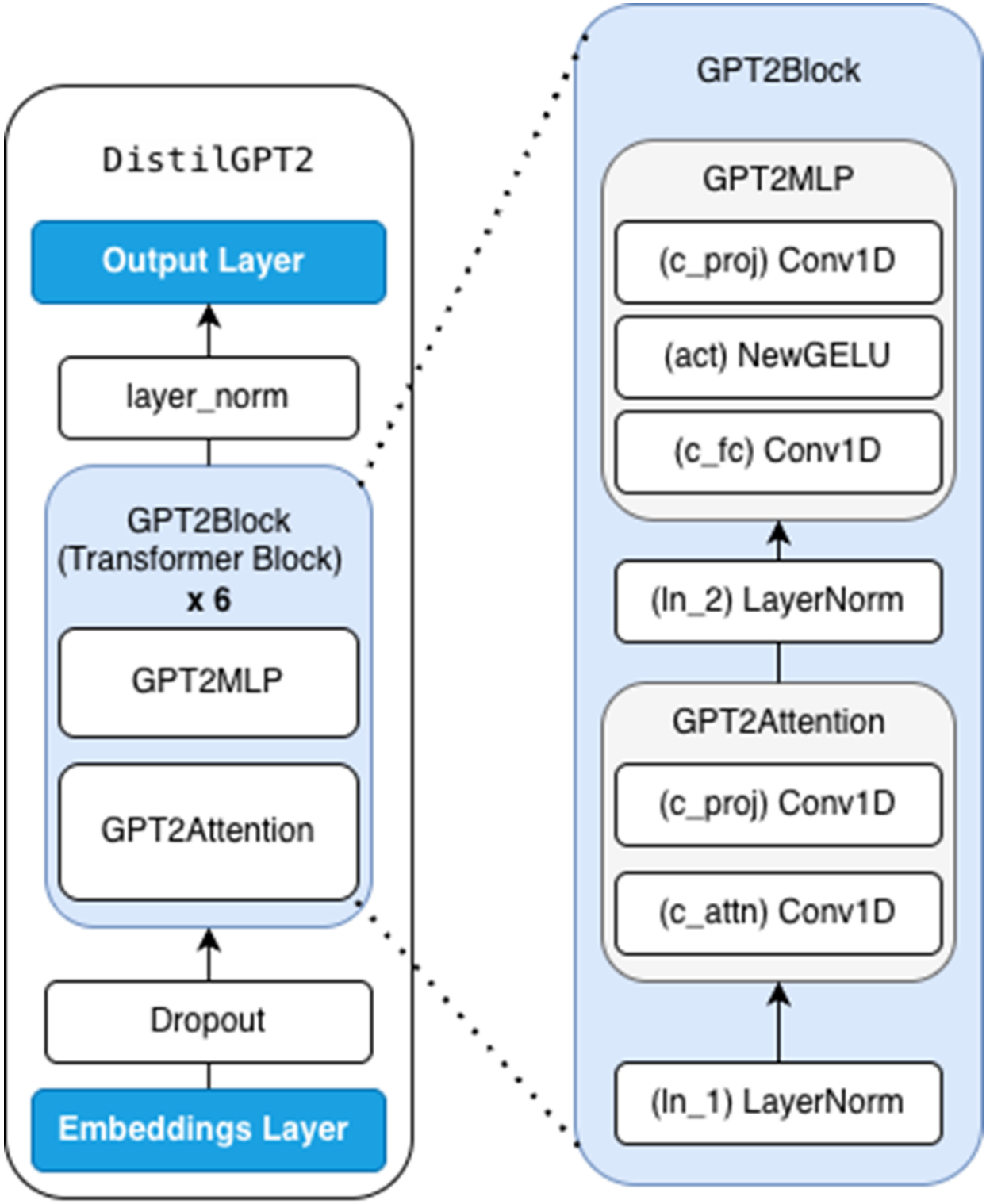


Figure 3.2 Anatomy of DistilGPT2. On the left, the general architecture is shown: data flows from the Embeddings layer, passes through the stack of six Transformer Blocks, and ends at the output layer. On the right, a "zoom" into one of those blocks reveals a `LayerNorm` → `GPT2Attention` →

`LayerNorm` → `GPT2MLP` flow, where each module processes the information before passing it to the next.

As you can see in Figure 3.2 (right), each `GPT2Block` Transformer block contains four components in the following order: the `ln_1` layer (normalization before the attention mechanism), the `GPT2Attention` module (the attention mechanism), the `ln_2` layer (normalization before MLP processing), and the `mlp` module (a feedforward neural network).

The attention mechanism, which we'll explore in detail in the next section, is responsible for contextualization: it gives each word the ability to "look at" and connect with the other words in the sequence to better understand its meaning. For example, in the sentence "Fresh water flows from the mountain spring daily", the word "spring" must attend to "water," "flows" and “mountain” to understand that it doesn't refer to the season. Attention calculates these connections automatically, determining which words are relevant to understand each position.

The MLP block acts as the knowledge processor: it takes the information that the attention layer has contextualized and transforms it by applying the knowledge the model has learned during training. Attention says, the word “spring” in this context is related to “mountain” and “water”, and the MLP uses its knowledge to understand that this implies concepts like "nature," "freshness," and "geography," and other similar terms. It's a traditional neural network that maps input patterns to richer representations.

This structure will be repeated in all the Transformers we will see, with variations in implementation but always maintaining this fundamental division of responsibilities: contextualization (attention) and knowledge processing (MLP).

Once the information has been processed by the six Transformer blocks, the model must convert its internal representations into a final prediction. This process occurs in two steps: first, the `ln_f` (final layer norm) layer stabilizes the output vectors. Next, the `lm_head` (language modeling head) projects each vector to the full vocabulary space (50,257 dimensions).

The result of this projection isn't a single token, but a vector of logits (scores) that represents a probability distribution over all possible tokens. For example, for the phrase "The mountain was covered in...", these logits might assign a 30% probability to "snow," 15% to "trees," and 10% to "fog." While 30% might seem low for a likely answer, it represents a strong prediction when distributed across a vocabulary of over 50,000 possible tokens. The remaining probability is spread thinly across thousands of other, far less likely options. It's from this distribution that the next token is finally selected.

NOTE

In the text, we've maintained a one-to-one relationship between Token and Word to simplify the explanation. This relationship isn't really the case, as a word doesn't translate directly into a token. The conversion depends on the language, among other things. In English, a ratio of 1.3 - 1.5 tokens per word is usually considered correct.

Now that we have an overview of what happens inside a transformer, let's dig deeper into the two main components: the attention mechanism and the MLP block.

3.1.2 Classical attention mechanism

Now that we have an overview of DistilGPT2's structure, let's dive deeper into its two main processing modules: the attention module (attn) and the neural network module (mlp). We'll start with the attention module, which characterizes modern Transformers.

You probably know Google's classic 2017 paper: "Attention is all you need" (https://arxiv.org/abs/1706.03762). This work is considered by many as the starting point of the modern LLM era. It introduced the self-attention mechanism that has been, and continues to be, the key piece that allows models to understand so well the context of the texts they process.

In this section, we'll analyze how DistilGPT2 implements attention in one of its most classic forms, with the goal of understanding not only its importance, but also how it can be optimized.

The input embedding we saw in the previous section contains information about the token's identity and its position, but it has a fundamental limitation: it doesn't understand how it relates to the other tokens in the text. For example, the part of the vector that represents the meaning of the word "spring" (its token embedding) is identical in "Fresh water flows from the mountain spring daily" and in "The garden comes alive again during the spring". While the positional embedding makes the final vector for "spring" different in each sentence, the model has not yet used the information from neighboring words to disambiguate its meaning. In other words, it knows the word used, its position in the sentence, but not what accompanies it.

This is where the attention mechanism comes into play. Its function is to enrich each token's input embedding with information from the rest of the sequence. This way, the vector for the word "spring" is modified and differentiated if

it is contextualized by "water" and "mountain" or by "flowers" and "season".

This contextualization happens inside the (attn) module, `GPT2Attention`. Using multiple sub-mechanisms that work in parallel: the attention heads. Each attention head is, in essence, a complete and independent attention mechanism that has specialized in looking for a specific type of relationship.

The specialization of the attention heads arises during the model's training. As the model adjusts its weights to minimize the prediction error, each head tends to specialize in different types of patterns as a result of the optimization. For example, one head might become an expert in detecting syntactic relationships (subject-verb), while another might specialize in semantic relationships (synonyms) or long-range dependencies.

You can check the number of attention heads in a model by directly accessing its configuration file. For example, the following code shows you this information for DistilGPT2:

```
config = model.config
print(f"Attention heads: {config.n_head}")
print(f"Head dimensions: {config.n_embd // config.n_head}")
```

Which will return:

```
Attention heads: 12
Head dimensions: 64
```

In this case, the content of the configuration file is telling us that DistilGPT2 has 12 attention heads. Each head projects the full 768-dimensional embedding into a smaller, specialized 64-dimensional subspace (768 / 12 = 64). The

`c_attn` layer is designed to very efficiently generate the necessary projections for all these heads simultaneously.

Figure 3.3 illustrates this division and specialization process.

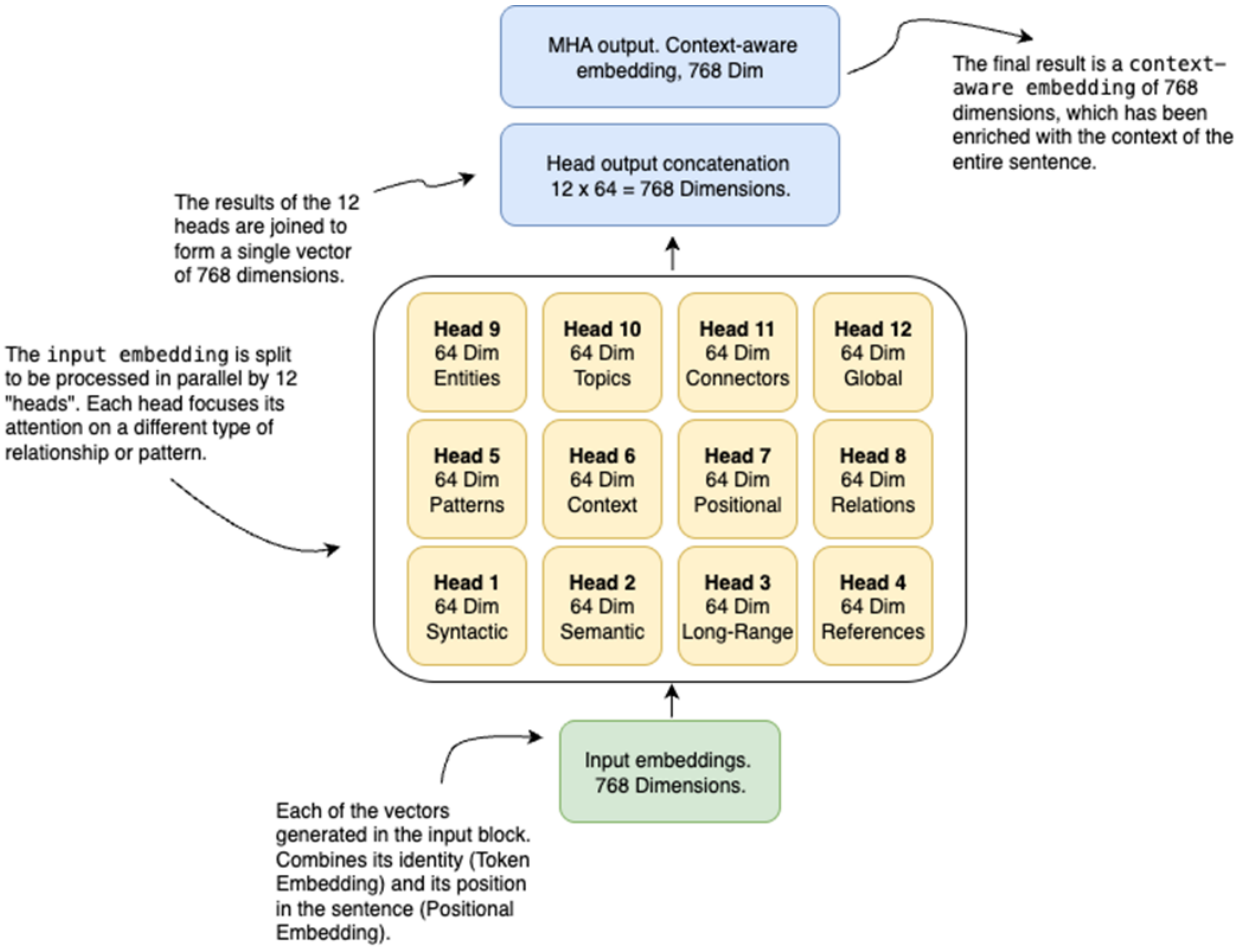


Figure 3.3 Multi-Head Attention architecture. The input embedding (768 dimensions) is divided into 12 specialized attention heads, each processing 64 dimensions in parallel. Each head focuses on different contextual relationships, before concatenating their outputs into a context-aware embedding that incorporates information from the entire sequence. Each head focuses on different contextual relationships (labels are illustrative; real specialization emerges from training and is not directly interpretable).

As you can see, each individual embedding is processed by the 12 heads in parallel, where each one analyzes different contextual aspects before concatenating their results and

creating an embedding of the same dimension but with the necessary context information.

This multi-head attention mechanism is highly effective at capturing complex relationships, but it comes with a significant cost: memory requirements. To fully grasp the scale of this cost and why it has been a driver of innovation in modern architectures, we need to delve deeper into the anatomy of the Transformer. Specifically, we must examine the internal workings of each attention head: the Query, Key, and Value (QKV) projection system.

THE QKV MECHANISM

The internal mechanism of attention calculates how much attention each token should pay to the others. To do this, from each token's embedding, it generates three distinct vectors that play different roles:

- *Query (Q):* It's a vector that represents the current word that's searching for information. It acts as a question. In the sentence *"Fresh water flows from the mountain spring daily"*, the *Query* of the token "spring" actively searches for clues in the rest of the sentence to define its own context.
- *Key (K):* It's a vector that represents what a word "offers" or the key to its content. The *Key* of the token "mountain" indicates that its information is related to nature.
- *Value (V)*: It's the vector that contains the real semantic information of the word. If the *Key* is the label, the *Value* is the content.

The objective is to enrich a token's embedding by adding a portion of the Value vectors (V) from all the other tokens in the sequence.

The main question is: what portion of each Value should we add? This is decided by the Query and Key mechanism. The Query of the current token (for example, "spring") is compared with the Key of each of the other tokens in the sentence. This compatibility is calculated through a dot product, an operation that measures how "aligned" the two vectors are.

A high dot product between the Query of "spring" and the Key of "mountain" indicates a high similarity. These scores are then normalized (using a softmax function) to become the final weights that determine what fraction of each word's Value will be used in the final mix to update "spring’s" embedding. Figure 3.4 illustrates this mechanism using our example sentence.

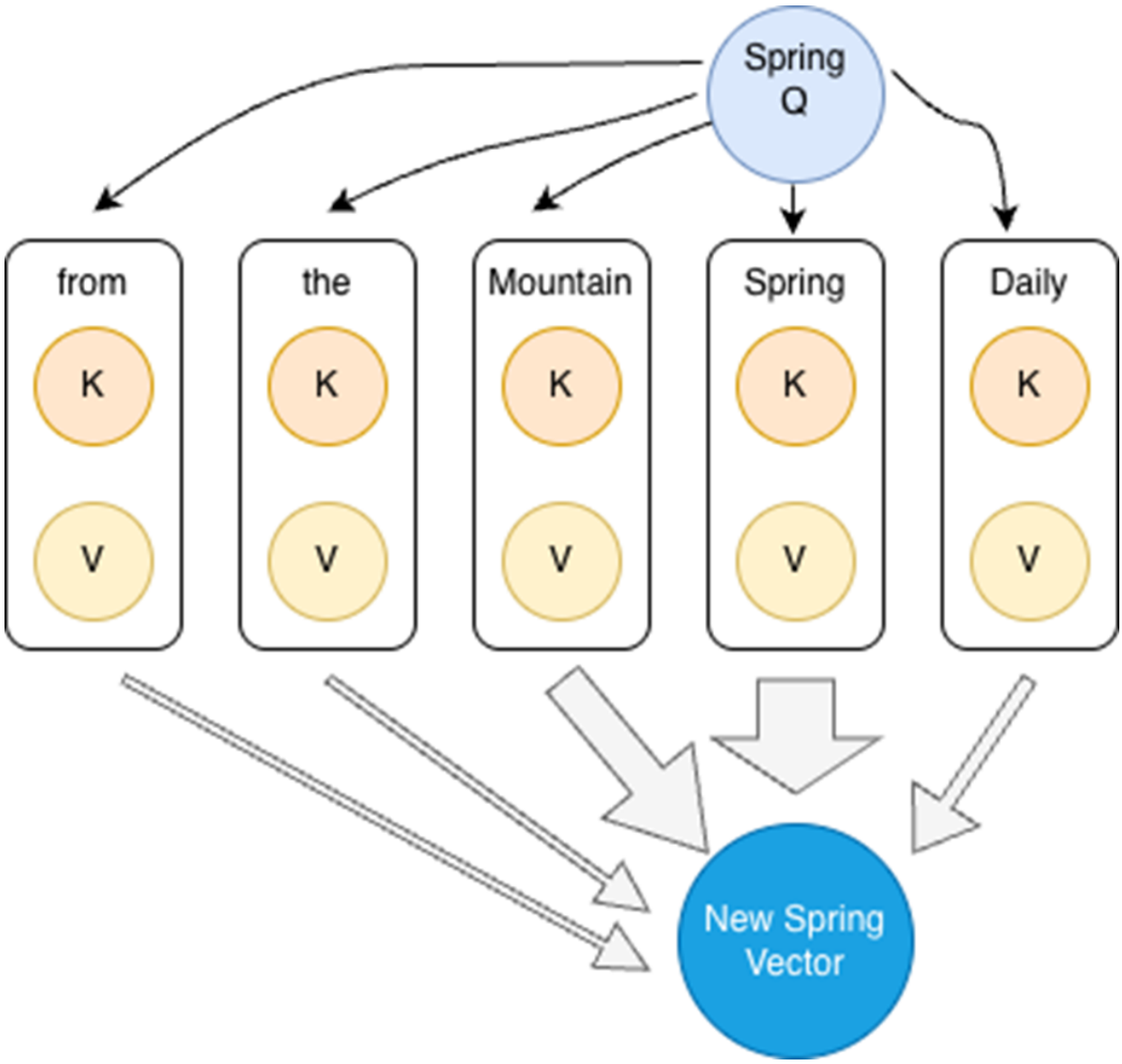


Figure 3.4 Attention mechanism for the token "spring." The Query (Q) from "spring" compares with the Keys (K) of all tokens through dot product operations. The resulting scores, after softmax normalization, determine what fraction of each Value (V) contributes to create the enriched embedding.

The classic DistilGPT2 architecture implemented this process in a very direct way: each of the 12 attention heads generated its own unique set of projections for Queries, Keys, and Values. The `(c_attn): Conv1D(nf=2304, nx=768)` layer was responsible for creating these projections all at once. This architecture is called Multi-Head Attention (MHA).

This layer takes the 768-dimensional embedding and projects it to a space of 2304, which is exactly triple the original size ($768 \times 3 = 2304$). This 2304 space is internally divided to generate the three vectors (Query, Key, and Value) for each of the 12 attention heads. Remember that all this happens within the Forward function of the Attention module.

The main problem is that during text generation, the model predicts one token at a time, and each new token needs to "attend" to all previous tokens. To avoid recalculating the Keys and Values for tokens that have already been processed, they are stored in a cache (the Key-Value Cache, or KV Cache). This cache grows with every token generated, representing a memory bottleneck that limits context window length and requires GPUs with substantial memory.

To address this issue, two attention architectures were developed, differing in their use of a reduced number of KV heads, which are grouped and shared across multiple Q heads:

- Multi-Query Attention (MQA): All heads share a single set of key-value pairs.
- Grouped-Query Attention (GQA): The KV heads are grouped, and each Q head is assigned to a specific group.

Figure 3.5 illustrates the three attention strategies and their memory trade-offs, showing how each approach manages the relationship between Query heads and the Key-Value pairs they access. The visual comparison makes clear why GQA has emerged as the practical standard: it captures most of MQA's memory savings while avoiding its quality compromises.

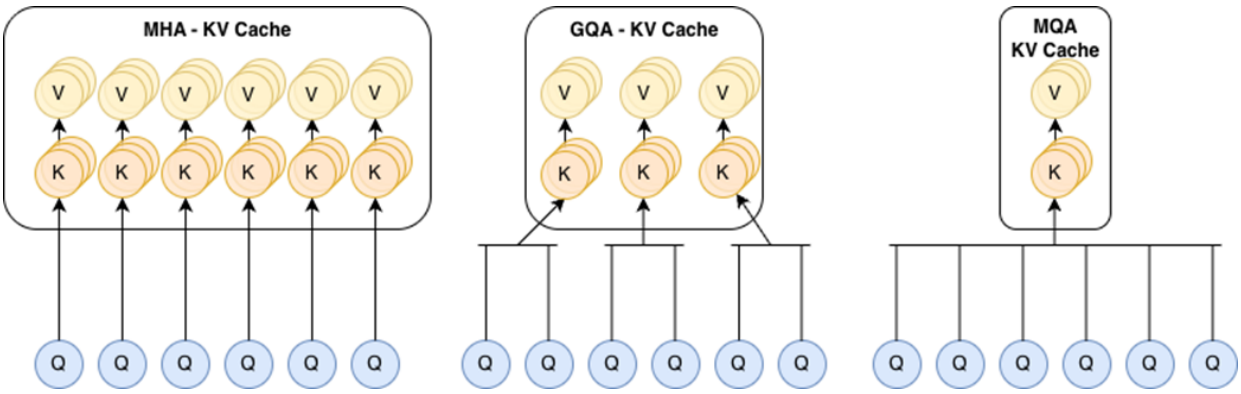


Figure 3.5 Attention mechanisms and memory optimization strategies. Multi-Head Attention (MHA) maintains independent K,V pairs for each Q head — one pair per processed token, forming a growing cache. Grouped-Query Attention (GQA) groups Q heads to share K,V pairs (here 6 Q heads in 3 groups, each sharing one K,V pair per token). Multi-Query Attention (MQA) compresses to the extreme: all Q heads share a single K,V pair per token.

To put this in perspective, let's consider our example sentence "Fresh water flows from the mountain spring daily". This sentence has 8 tokens; we'll take the liberty of considering 1 word = 1 token. To predict the next token, DistilGPT2 has to keep the Key and Value vectors for each of these 8 tokens in its KV Cache, across its 12 attention heads and 6 Transformer blocks. That is: 8 tokens × 12 heads × 2 (K and V) × 6 layers = 1152 “slots” of information. In reality, each slot takes up quite a bit more than a single vector, so the actual memory consumption grows even more than this illustrative calculation suggests. All this information must be kept in the GPU's memory just to process a single short sentence. As we increase the layers, attention heads, or the length of the text, the memory requirement keeps growing.

Figure 3.6 illustrates this memory pressure visually, showing how the KV Cache grows token by token for each attention mechanism. The dramatic difference in slopes reveals why GQA and MQA have become essential for modern LLMs.

Figure 3.X: KV Cache memory growth

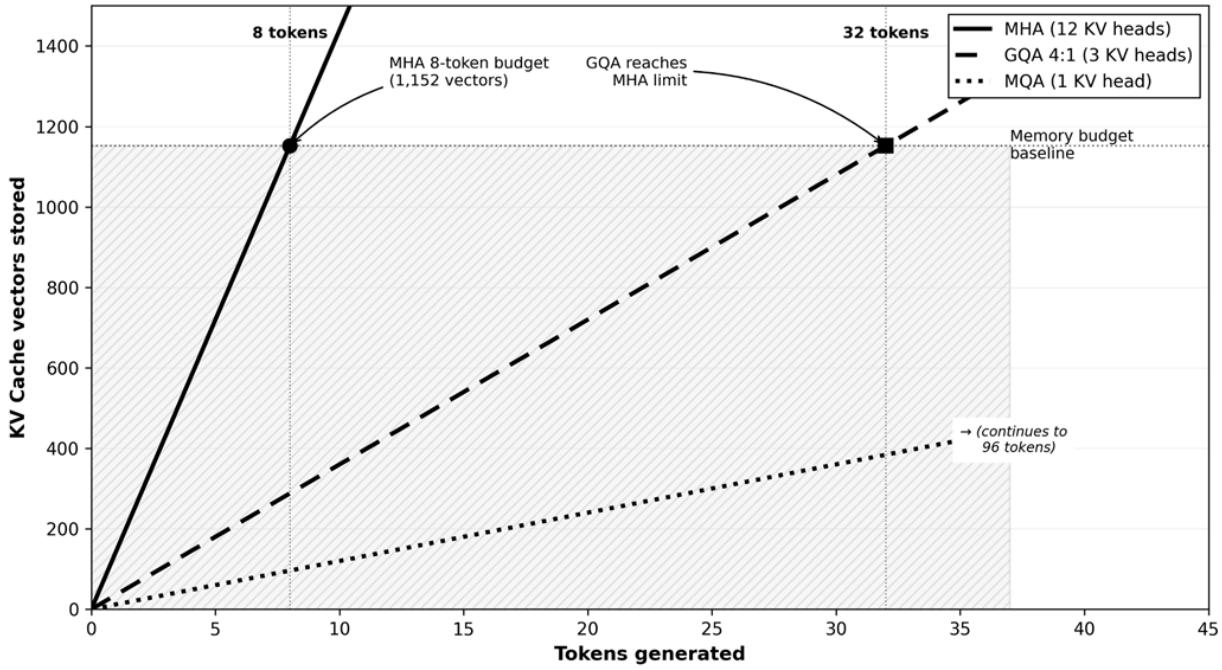


Figure 3.6 KV Cache memory growth during sequential token generation. The graph compares cache growth for three attention mechanisms using the example from the text: a 6-layer model with 12 attention heads. The solid line shows MHA storing 12 KV pairs per token, requiring 1,152 vectors to cache 8 tokens. GQA 4:1 with 3 KV pairs per token can process 32 tokens with the same 1,152-vector budget (+300%), while MQA with a single KV pair can process 96 tokens (+1100%).

Figure 3.6 illustrates how reducing KV heads enables dramatically longer context windows with identical memory resources.

Faced with this memory pressure, it might be tempting to simply disable the KV Cache, an option technically available in libraries like Hugging Face's `transformers`. However, this is almost never a good practice, even for classic architectures like DistilGPT2.

Disabling the KV forces the model to recalculate Keys and Values for every token, which introduces excessive computational load. We're trading a memory problem for a computation capacity one, and inference would become

extremely slow. The one context where disabling it is legitimate is during training, where the full sequence is processed in parallel and no cache is needed.

This knowledge of the internal architecture is crucial, as it is the first step to identify potential optimization points. Understanding that each layer repeats this costly context analysis allows us to intuit that it won't always be necessary to execute it completely. Later in the book, we'll explore techniques to optimize the model's performance based on this same idea, but for now, the fundamental thing is to understand the function of the attention layer.

Once the attention layer has enriched the embeddings with contextual information, they are passed to the second fundamental component of the Transformer block: the MLP network, responsible for processing and transforming this information.

3.1.3 The classic MLP mechanism

We know that DistilGPT2 has six Transformer blocks; each of these blocks contains an attention module followed by an MLP module. It's important to remember that this sequence runs in order: Attention → MLP → Attention → MLP, and so on through the six blocks. It's not about processing all the attention layers first and then all the MLP layers, but rather each block completes both steps before moving to the next one.

Once the attention module has enriched the embeddings with contextual information, they reach the second component of the same block: the MLP module, responsible for processing and transforming this information using the knowledge that the model has acquired during its training.

Let's go back to the example we've used previously: "Fresh water flows from the mountain spring daily." When attention determines that the word "spring" is strongly connected with "water" and "mountain", the MLP is the one that uses its internal knowledge to interpret that this combination of context implies concepts related to "nature," "geography," and "water," instead of "seasons" or "mechanics". In other words, and simplifying, the Attention layer contextualizes and the MLP layer converts contextualization into knowledge.

To understand how the MLP processes knowledge, let's observe its internal structure:

```
(mlp): GPT2MLP(
     (c_fc): Conv1D(nf=3072, nx=768)
     (c_proj): Conv1D(nf=768, nx=3072)
     (act): NewGELUActivation()
     (dropout): Dropout(p=0.1, inplace=False)
)
```

The first thing we should mention is that the order in which the layers appear doesn't always correspond to their execution order. PyTorch displays the layers as they were defined in the `__init__()` function, but the actual processing order is set by the `forward()` function of the MLP module.

The following code shows us the `forward()` function of the MLP module:

```
import inspect
mlp_layer = model.transformer.h[0].mlp
print(inspect.getsource(mlp_layer.forward))
```

This gives us the following result:

```
def forward(self, hidden_states: Optional[Tuple[torch.FloatTenso
r]]):
        hidden_states = self.c_fc(hidden_states)
        hidden_states = self.act(hidden_states)
        hidden_states = self.c_proj(hidden_states)
        hidden_states = self.dropout(hidden_states)
        return hidden_states
```

Now we can see the real processing order of the layers within the MLP module: expansion (`c_fc`) → activation (`act`) → contraction (`c_proj`) → regularization (`dropout`).

In other words, the first thing we find is that the 768-dimension embedding is received and expanded to 3072, then it's sent to an activation function, and then contracted back to 768 dimensions.

This expansion to 3072 dimensions helps the model represent and process much more complex information than would fit in the original 768 dimensions. But although the expansion is a very powerful mechanism, it has a fundamental problem: if we simply chain the expansion (`c_fc`) and the contraction (`c_proj`) with nothing in between, this whole operation would be mathematically equivalent to a single matrix multiplication. This is because linear transformations, when combined, remain linear.

For this reason, the activation function is introduced between the expansion and the contraction to break this linearity, using the layer: `NewGELUActivation()`. GELU (Gaussian Error Linear Unit) is a smooth, non-linear activation function that helps the model learn complex patterns by selectively passing or attenuating signals from each neuron. This function acts like a "switch with regulator" for each of the 3072 neurons: it decides if the signal a neuron has calculated is important enough to pass to the next layer and with what intensity it should do so. The simple act of

modifying which neurons pass through and with what intensity is what breaks the linearity.

During the expansion phase, each neuron tends to specialize in detecting specific patterns in the contextualized information. Some are activated by geographical concepts, others by temporal relationships, and others by specific syntactic structures. Something similar to what happened with the attention heads. This specialization emerges naturally during the model's training process, while its weights are being adjusted.

This neuronal specialization is key for width pruning: If a general-purpose model contains neurons specialized in thousands of topics (from history to biology, to programming), but our goal is to create an expert model solely for financial analysis, what's the point of keeping the neurons that detect patterns about cooking recipes active and consuming resources? This is where we introduce the idea of width pruning, a surgical technique that allows us to selectively eliminate the neurons that contribute the least to the model. We'll explore this technique in depth in Chapter 5.

With this, we've completed the analysis of a classic Transformer block.

3.1.4 The Transformer dimensions: depth and width

By breaking down both the attention module and the MLP module, you now have a complete view that lets you understand what happens when we apply two of the most important structural optimization techniques: depth pruning and width pruning.

- *The depth* refers to the number of Transformer blocks that we stack on top of each other. In DistilGPT2, the depth is six Transformer blocks. Modifying the depth, as we saw in the previous chapter with depth pruning, involves removing entire blocks. Generally, more depth allows for a more complex sequential refinement of information, but at the cost of higher latency.
- *The width* refers to the size of the internal layers, specifically the intermediate dimension of the MLP block. In our DistilGPT2, the width is 3072 neurons. This dimension determines the model's capacity to process knowledge at each step. Reducing the expansion capacity with width pruning directly impacts the number of parameters and memory consumption.

The shape of models varies a lot depending on the decisions made during their design and development. In Figure 3.7 we show the shape of four of the models that will accompany us throughout the book.

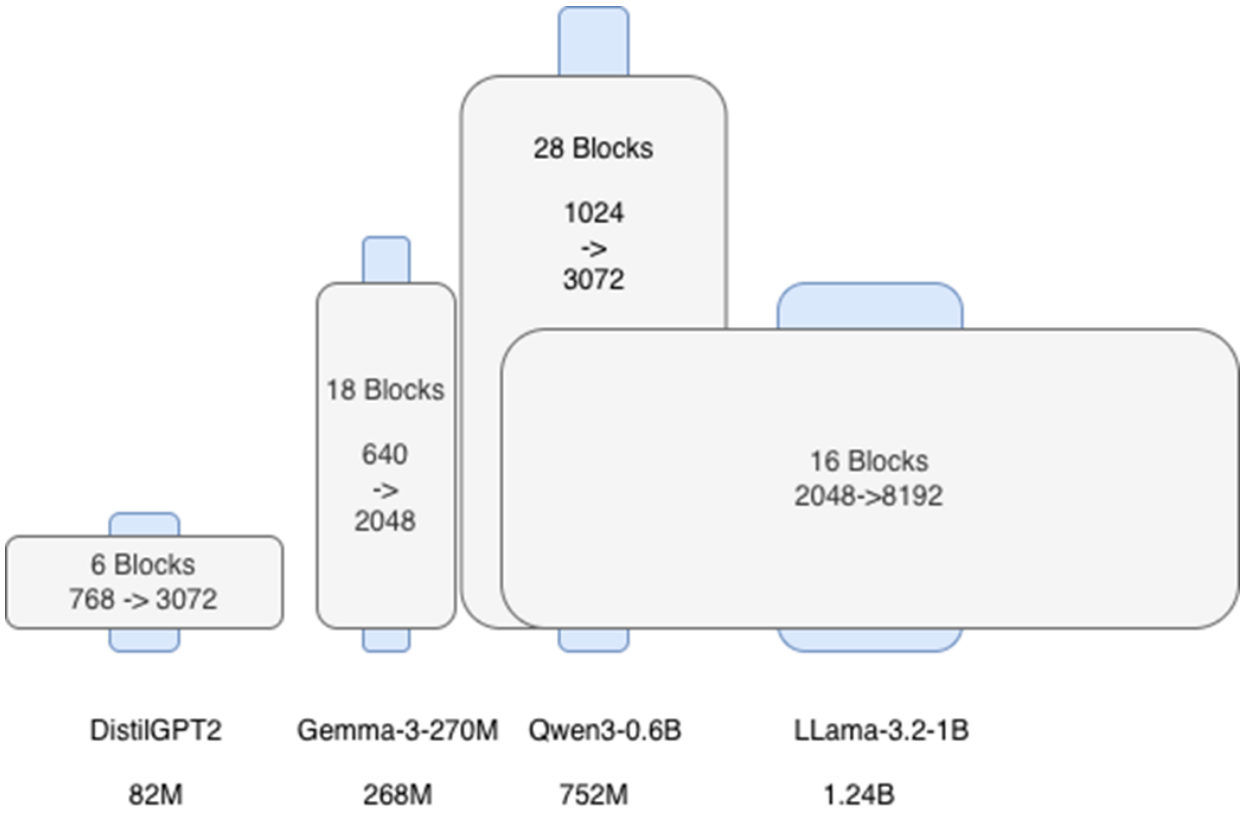


Figure 3.7 Comparison of the "shapes" of different models. The narrow column represents the embedding dimension, the main box shows the model's depth (its height, proportional to the number of blocks) and its processing width (its width, proportional to the size of the MLP expansion).

We can distinguish two main approaches:

- *Wide models*: Architectures like Llama-3.2-1B and DistilGPT2 are notably wide. They use a large input embedding and a larger MLP expansion than in the other models, in this case x4, which gives them a large processing capacity in each of their layers.
- *Deep and narrow models*: In contrast, models like Qwen3-0.6B and Gemma-3-270M, with a smaller embedding and an x3 expansion, are deeper and narrower. They prioritize a higher number of sequential transformations.

Understanding the balance between depth and width is fundamental for an LLM architect. Now that we know the basics, let's see how this structure has evolved in modern architecture.

3.2 The modern Transformer architecture

The Attention and MLP modules are the pillars of the transformer architecture. We used DistilGPT2 to explain the functionality and structure of their most classic form, which despite being revolutionary and meaning a great step forward, ended up presenting two problems when scaling the models:

- The attention mechanism became memory-intensive as context windows grew, creating a bottleneck that also impacts computational cost.
- The MLP module, although effective in its original design, became limited for deeper and more contextual knowledge processing.

The response to these challenges was the adoption of two foundational innovations that define the modern standard: the Grouped-Query Attention (GQA) mechanism we just examined, which optimizes the memory consumed by the attention mechanism, and Gated Linear Units (GLU), which revolutionize MLP processing.

To study these two optimizations, we'll use a model from the Llama family, which has popularized the structure that today predominates in most *open source* models.

NOTE

The Llama-3.2 model family has gated access on the Hugging Face Hub. To download it, you'll need to perform two steps. First, visit the model's page on the Hub and accept the license terms. Second, you must authenticate from your notebook environment. This involves configuring a secret key named HF_TOKEN with your Hugging Face User Access Token in your Google Colab environment.

Let's look at the structure of Llama-3.2-1B:

```
model = AutoModelForCausalLM.from_pretrained(
   "meta-llama/Llama-3.2-1B",
    torch_dtype="auto",
    device_map="cpu",
    trust_remote_code=True
)
```

The result immediately reveals the structural differences compared to DistilGPT2:

Listing 3.2 Llama-3.2-1B structure

```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(128256, 2048) #A
    (layers): ModuleList(
      (0-15): 16 x LlamaDecoderLayer( #B
        (self_attn): LlamaAttention(
          (q_proj): Linear(in_features=2048, out_features=2048, bias
=False)
          (k_proj): Linear(in_features=2048, out_features=512, bias=
False)
          (v_proj): Linear(in_features=2048, out_features=512, bias=
False)
          (o_proj): Linear(in_features=2048, out_features=2048, bias
=False)
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=2048, out_features=8192, b
ias=False)
          (up_proj): Linear(in_features=2048, out_features=8192, bia
s=False)
          (down_proj): Linear(in_features=8192, out_features=2048,
                                                ➥bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
        (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
      )
    )
    (norm): LlamaRMSNorm((2048,), eps=1e-05)
    (rotary_emb): LlamaRotaryEmbedding()
  )
  (lm_head): Linear(in_features=2048, out_features=128256, bias=Fals
e)
)
```

#A Expanded vocabulary and width.
#B Depth increased to 16 blocks.

At first glance, the structure of Llama-3.2-1B reveals a clear evolution compared to DistilGPT2. Some differences are

related to scale, but the truly significant changes are architectural.

One initial difference is that the vocabulary has been expanded and the model has a greater width. We can see that Llama supports 128,000 tokens in its vocabulary (embed_tokens): Embedding(128256, 2048), compared to the 50,000 of DistilGPT2 (wte): Embedding(50257, 768). In the same lines, you can also see that the embedding is now 2048 dimensions versus the 768 of DistilGPT2. Also, Llama is deeper, with 16 blocks compared to DistilGPT2's 6.

You may also notice two unfamiliar components in the architecture. The first is `rotary_emb (LlamaRotaryEmbedding)`, which replaces the positional embedding layer `(wpe)` present in DistilGPT2. Both serve the same purpose: telling the model where each token sits in the sequence. The key difference is that classic embeddings encode absolute positions ("you are word number 5") and have a fixed limit, while `RoPE` encodes relative distances between tokens, enabling much longer context windows. This is why modern models can process 128K tokens, whereas DistilGPT2 was limited to 1024.

The second is `LlamaRMSNorm`, which replaces the classic `LayerNorm`. Both are normalization techniques that stabilize training, but `RMSNorm` is computationally cheaper while achieving similar results.

However, the truly architectural evolution is within the Transformer blocks. It's in the Attention and MLP modules where we find new layers and designs that optimize performance.

We’re going to study these advancements in detail, starting with the heart of the Transformer: the attention mechanism.

3.2.1 Optimized attention: from Multi-Head Attention (MHA) to Grouped-Query Attention (GQA)

We have seen the theory and the memory benefits of GQA. Now, let's look at the anatomy of a modern implementation. By inspecting the layers of Llama-3.2, we can see exactly how this grouping logic is hardcoded into the model's structure.

Let's compare the DistilGPT2 attention module with a classic MHA:

```
(c_attn): Conv1D(nf=2304, nx=768)  #A
```

#A A single layer for QKV

With the GQA implementation in Llama-3.2:

```
(q_proj): Linear(in_features=2048, out_features=2048,
                                 ➥ bias=False)  #A
(k_proj): Linear(in_features=2048, out_features=512, bias=False)
(v_proj): Linear(in_features=2048, out_features=512, bias=False)
(o_proj): Linear(in_features=2048, out_features=2048, bias=False)  #
B
```

#A Separate projection layers for Q, K, V
#B Output projection combines all attention heads into final result

Let's remember that DistilGPT has the three Q, K, and V vectors into one layer, and it's the PyTorch code of the forward method that handles the separation when necessary. Llama uses separate projections for each. The `o_proj` layer serves a different purpose: after all attention heads compute their outputs, this layer recombines them back into a single embedding representation.

At first glance, the most revealing change is in the dimensions. Instead of a single monolithic layer, Llama-3.2-

1B uses separate projections, and here is the key: `q_proj` has 2048 `out_features`, but `k_proj` and `v_proj` only have 512. This is due to how GQA is optimizing the architecture of the Attention module.

As introduced in the previous section, the idea behind GQA is simple: instead of each attention head generating and operating with its own unique set of Keys and Values, "groups" of heads are now created. Within each group, all the heads still generate their own unique Queries, but they all share a single set of Keys and Values.

As we've already seen in figure 3.6, GQA can process 32 tokens with the same memory that MHA processed only 8.

- *MHA:* 8 (tokens) × 12 (KV heads) × 2 (K and V vectors) × 6 (layers) = 1,152 vectors
- *GQA*: 32 (tokens) x 3 (KV heads) x 2 (K and V vectors) x 6 (layers) = 1,152

Understanding this evolution from MHA to GQA allows us to comprehend why simple attention head pruning, an approach that was intuitive in models with MHA architecture, has lost relevance. In classic MHA, each head had its own independent weights for Queries, Keys, and Values. Pruning a head simply meant eliminating those projections and adjusting the output dimension, a localized and straightforward modification.

With GQA, heads within a group share Key and Value projections. This interdependence makes eliminating an individual head not only less beneficial, but also riskier at the implementation level. *Removing a single head affects shared tensors used by others, and a misalignment can break the model's integrity*. That's why we'll need different techniques, like the Attention Bypass we'll explore in

Chapter 8, to optimize these modern attention modules safely and effectively.

IMPLEMENTING AN ATTENTION MECHANISM

In this chapter, you've seen the functionality of the QKV mechanism and the improvements that GQA has brought. If you want to understand the code behind the implementation, I highly recommend *Build a Large Language Model (From Scratch)* by Sebastian Raschka.

Now that we've seen how memory has been saved in attention processing, which has allowed for ever-larger context windows, we can move on to study the other major evolution: the improvement of the MLP module.

3.2.2 The evolution of the MLP: from simple expansion to Gated Linear Units (GLU)

The traditional MLP module I discussed earlier in this chapter has a limitation that's more conceptual than computational. This isn't about memory consumption or massive resource usage. We're talking about how information is filtered. A fixed activation function like GELU applies the same mathematical criteria to every input, with no learned control over which neurons should be amplified or suppressed. GLU addresses this by introducing a learned multiplicative gate per neuron, a second projection that learns to modulate the first based on the input itself.

For example, when the model processes the word "spring" in the context of a “water" or "season", the GELU activation doesn't change. The context is present in the inputs, but the MLP doesn't adjust its filtering dynamically.

This lack of contextual adaptation in filtering is what the Gated Linear Units (GLU) architecture comes to solve, replacing the static filter of the activation layer with a filter that learns to adapt to the specific content of the input.

Let's look at the structural difference between the two solutions. Let's recall the classic MLP architecture of DistilGPT2:

```
(mlp): GPT2MLP(
    (c_fc): Conv1D(nf=3072, nx=768)
    (c_proj): Conv1D(nf=768, nx=3072)
    (act): NewGELUActivation()
    (dropout): Dropout(p=0.1, inplace=False)
 )
```

Compare it with the structure of the Llama-3.2-1B MLP Module, which implements GLU:

```
(mlp): LlamaMLP(
    (gate_proj): Linear(in_features=2048, out_features=8192, bias=Fa
lse)
    (up_proj): Linear(in_features=2048, out_features=8192, bias=Fals
e)
    (down_proj): Linear(in_features=8192, out_features=2048, bias=Fa
lse)
    (act_fn): SiLU()
)
```

The difference is clearly visible; DistilGPT has two main linear layers, the expansion and the contraction. Llama-3.2-1B has three layers, and an activation function, each with a specific task.

NOTE

Llama's specific combination of SiLU as the activation function and the GLU architecture has its own name in the literature: SwiGLU. Other models use variants like GeGLU (GELU gate) or ReGLU (ReLU gate). You'll encounter these terms in papers and model cards. Throughout this book we use the family name GLU, because the width pruning techniques we apply in Chapter 5 work identically across all variants.

Figure 3.8 compares the processing flow of the classic MLP versus the modern GLU mechanism.

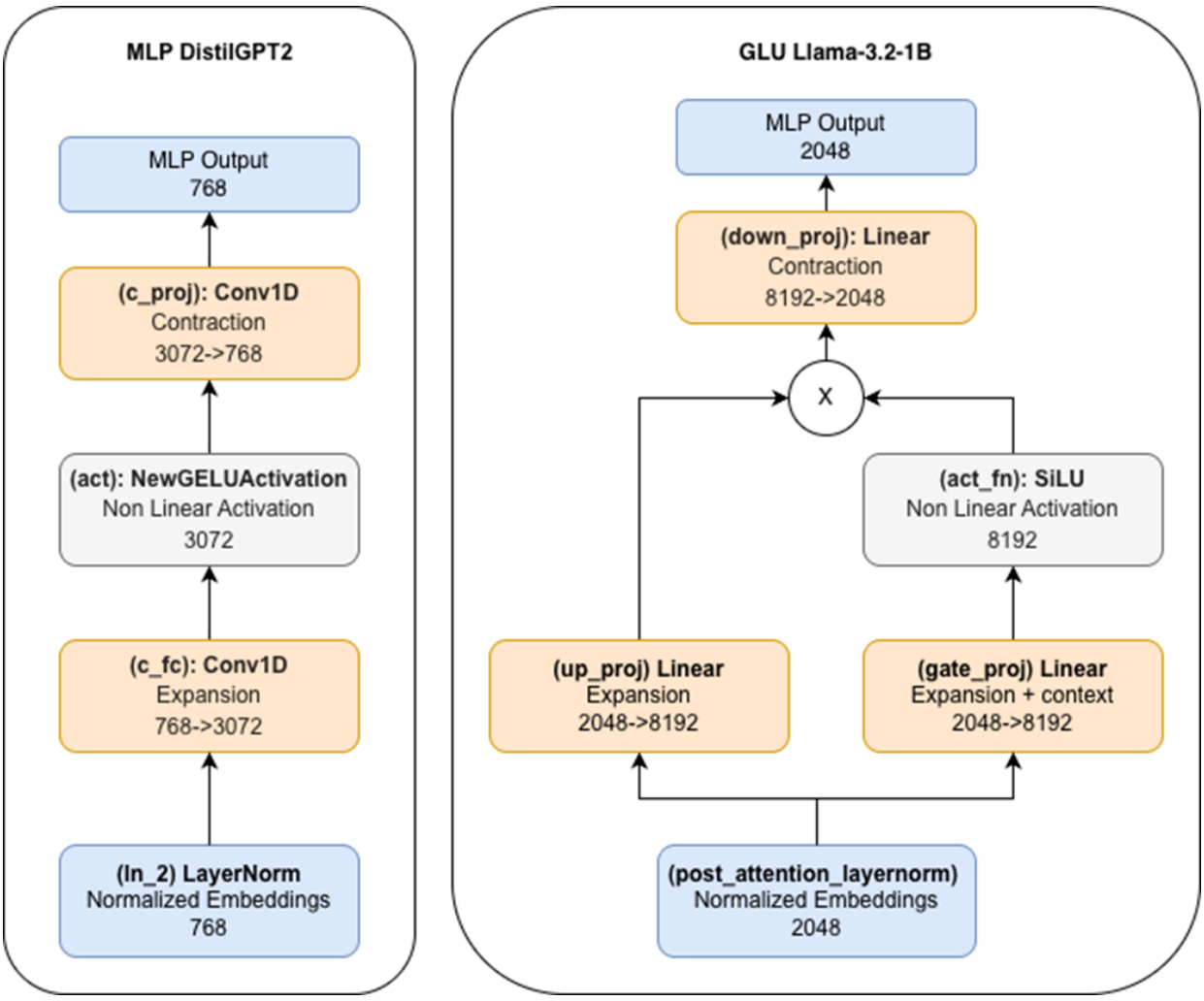


Figure 3.8 Processing flow in MLP Modules: classic vs. GLU. While the classic MLP follows a simple expansion-activation-contraction sequence, GLU uses two parallel projections: one `up_proj` that transforms the content and another `gate_proj` that creates a dynamic filter.

The execution flow of DistilGPT was simple. First the information is expanded, then activation is applied, and finally the resulting representation is contracted back to its original size.

The information flow through Llama-3.2-1B layers, or any model that implements a GLU architecture, is more complex and follows these steps:

- *Content projection:* A projection occurs in the layer `up_proj`. This generates a transformation of the content, creating an intermediate representation rich in information, analogous to the expansion of the classic MLP. There's nothing new here; this layer corresponds to `(c_fc)` in the DistilGPT2 model.
- *Gate projection*: In parallel, `gate_proj`, which generates a control vector or *gate*. This vector doesn't contain the knowledge itself, but rather, after passing through the `SiLU` activation function, learns to determine which elements of the transformation performed by `up_proj` are relevant for the current context.
- *Element-wise combination:* The two results are combined through element-wise multiplication, generating a vector of the same size but already filtered based on context. The activation function tends to compress very negative inputs close to 0 and smooth out positive ones, which in practice allows the gate to not only attenuate, but also modulate information more flexibly.
- *Contraction:* The information contracts again in the `down_proj` layer, which is equivalent to the contraction layer `c_proj` that we find in DistilGPT2.

Let's explain it using our example sentence: "Fresh water flows from the mountain spring daily". When the MLP Module processes the token "spring," which has already been enriched by the attention layer with the context of "water" and "mountain." The content projection `up_proj` expands its representation, activating neurons associated with its multiple meanings: nature concepts (water source, stream) and also temporal concepts (the season) or mechanical ones (a metal coil).

Simultaneously, the control projection, `gate_proj`, having processed the same context, learns to generate a "gate" that will assign high values to geographic neurons and low values (close to zero) to financial ones.

The multiplication between both has a filtering effect that takes context into account: information about "gardens" and "flowers" is attenuated or eliminated, while that of "mountains" and "water" is enhanced. This way, the control vector modulates the content representation before it passes to the final contraction layer: `down_proj`.

As we can see in figure 3.8, both architectures maintain a common element: the expansion and contraction of information. Just as we indicated when studying the MLP module of the DistilGPT2 model, we can target this expansion to perform a width pruning process that reduces the model's width.

IMPORTANT

The GLU architecture introduces a new consideration: the gate_proj and up_proj layers work in pairs. We can't remove a neuron from one layer without removing its corresponding one in the other, as this would break the necessary correspondence for the element-wise multiplication. Therefore, the calculation to decide which neurons to prune must be different, evaluating the importance of each pair of neurons jointly.

One of the tasks we'll need to do when rearchitecting an LLM is finding what expansion percentage is necessary for our needs. Through the application of width pruning on Llama-3.2-1B and 3B models, we discovered that there's a sweet spot in the expansion ratio. Our experiments demonstrated

that a lower expansion ratio not only allows drastically reducing the model size and its computational cost, but in certain tasks, can even improve its performance compared to the original model. But it's not the same in all models, experiments with the Gemma family have given completely different results.

In Chapter 5, we'll explore the width pruning technique in architectures with GLU, which allows us to apply these discoveries in practice.

3.3 Connecting structure, behavior, and optimization

In the previous sections we have analyzed in detail the structure of Transformers, starting from the DistilGPT model that has a classic modern structure, and we have introduced the most important improvements that have been produced by studying LLAMA-3.2.

You have obtained a detailed knowledge of the internal anatomy of modern Transformers. You know the functionality, shape, and operation of the two main modules of a Transformer: Attention and the MLP. You have a very complete vision of the static form of a model, which throughout the book you will complement with the dynamic vision of what happens inside the model at inference time.

To acquire this knowledge, we rely on activations. An activation is the output value produced by a neuron after it processes its input through a mathematical function. For example, in Llama-3.2-1B we have 8,192 neurons in each MLP layer, and each neuron applies its own learned weights; one weight for each dimension of the input embedding, so 2,048 weights per neuron. Put simply, the model's weights are static parameters learned during training, while

activations are the dynamic values that propagate through the network when it processes each specific input. As you can see in figure 3.9, a layer transforms its input vector into a new activation vector, where some neurons produce higher values and others remain close to zero.

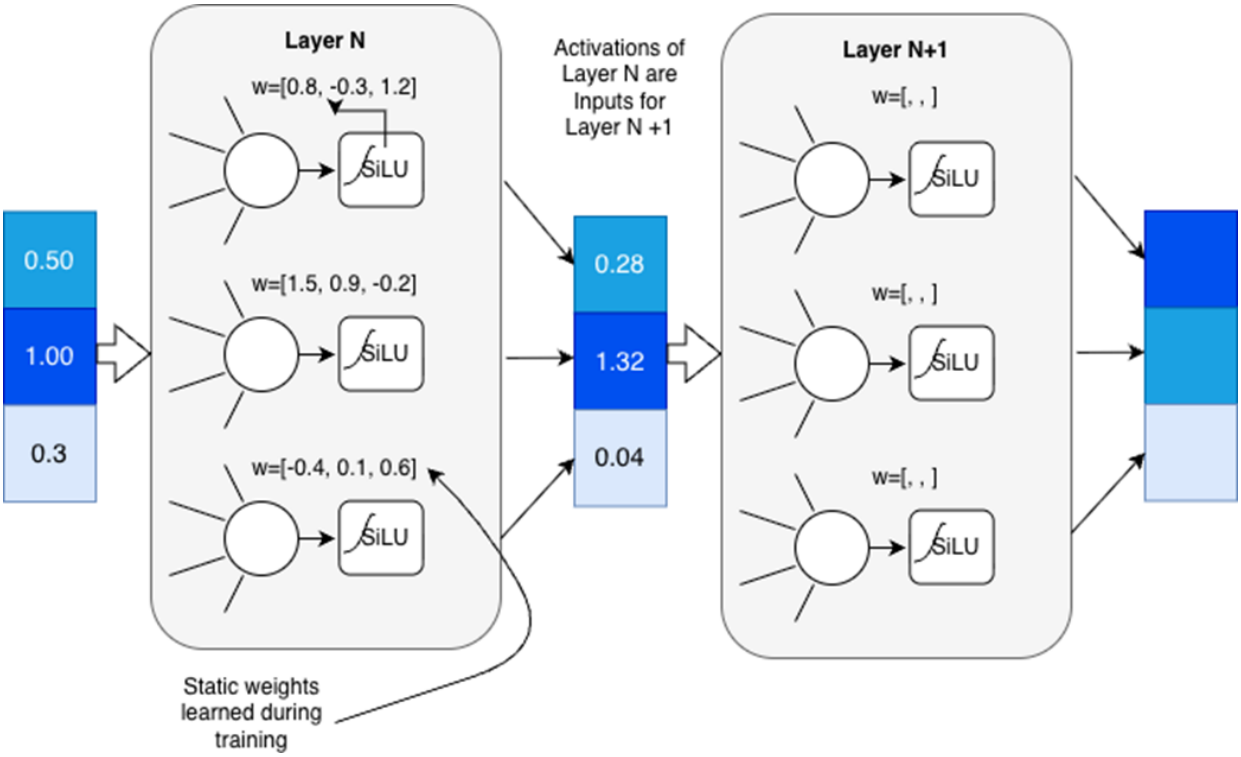


Figure 3.9 How activations flow between layers. Each neuron applies its static weights (learned during training) to the input values and passes the result through an activation function (SiLU). The output values, the activations, vary in intensity: high values propagate strong signals, while low values suppress information. These activations become the inputs for the next layer, creating the continuous flow of information through the model.

The structure tells you what components the model has:

- It has 16 layers with 4x expansion MLP modules.
- It uses Grouped Query Attention with a 4:1 ratio.
- It implements GLU architecture with SiLU function.

The behavior reveals how these components react to specific inputs. The following examples are fictional but

representative of the kind of patterns we will actually measure in later chapters:

- When processing “doctor,” neurons 1247 and 2891 activate intensely.
- The activation difference between “he” and “she” is 300% higher in layers 8-12.
- This activation pattern suggests that the model associates certain demographic groups with specific roles.

Studying behavior through activations allows us to identify which part of the model produces biases, enabling us to act on those parts to correct it.

In short: analyzing activations lets us diagnose how the model "thinks," opening the door not only to optimize efficiency, but also to correct its behavior with surgical interventions.

With this understanding of structure and behavior, you can see the full arsenal of re-architecture techniques. Table 3.1 shows the relationship between the components you just studied and the techniques you’ll master in the coming chapters.

Table 3.1 Optimization map

Structural component	Optimization technique	Description	Chapter
Transformer blocks	Depth Pruning	Removing entire Transformer blocks	4
Classic MLP module	Width Pruning	Expansion reduction	5
GLU MLP module	Width pruning	Synchronized expansion reduction	5
Attention module	Attention Bypass	Layer Attention Bypass	8
	Adaptive Attention Bypass	Adaptive Layer Attention Bypass	9
Activations	Fair Pruning	Behavior-driven pruning	10 / 11

This table establishes a direct connection between the anatomy you just studied and the practical techniques you'll learn. Each component of the model has specific techniques designed to optimize it, and now you know exactly where to apply each one depending on what you want to achieve with your model.

3.4 Hands-on lab

In the notebook that goes along with the chapter you'll find the structures of the models we've seen here. But at the end, in a section called "Exploring Other Notable Architectures" you'll also find models we haven't explored, like QWen3-06B or Microsoft Phi-4. I encourage you to use it as your own lab.

Here are some experiments to get you started:

- Compare GLU implementations across models:

    - What to do: Load and examine the MLP structures of Qwen-3-0.5B and Gemma-3-270M. Compare them with the Llama-3.2-1B architecture.
    - Questions to answer: What architectural element (related to the MLP) do Qwen and Gemma share that differs from Llama? Why do you think this design decision was made?
- Analyze expansion ratios and model design philosophy:
    - What to do: Calculate the expansion ratios for different models by examining their `gate_proj` and hidden dimensions.
    - Questions to answer: What expansion ratio does Qwen-3 use? Based on its number of layers vs. MLP size, would you classify it as a “wide” or “deep” model? What are the advantages and disadvantages of each approach?
- Explore advanced attention mechanisms:
    - What to do: Examine Phi-4’s attention implementation by looking at its `qkv_proj` structure and comparing it with the separate Q, K, V projections used in other models.
    - Questions to answer: What attention mechanism does Phi-4 implement? What makes it special compared to the standard multi-head attention we’ve studied?
- Explore a multimodal model:
    - What to do: Go to the Hugging Face Hub and search for a multimodal model like gemma-3n-E4B-it. Load it and review its config.json file.
    - Questions to answer: How does the structure of its configuration differ from the text-only models we've analyzed? Can you identify keys or entire sections that clearly aren't part of a standard text Transformer?

3.5 Summary

- Transformers consist of repeating blocks, each containing two core modules: an attention module for contextualizing information and an MLP module for processing knowledge.
- The two fundamental dimensions for rearchitecting a model are its depth (the number of stacked Transformer blocks) and its width (the size of the expansion within the MLP modules).
- Modern architectures like Llama evolved from the classical design by incorporating key innovations such as Grouped-Query Attention (GQA) to reduce memory consumption and Gated Linear Units (GLU) to create more sophisticated MLP processing.
- A model's static structure defines its components, while its dynamic behavior is revealed through its activations during inference.
- By understanding the specific function of each component, you can strategically map techniques like Depth Pruning, Width Pruning, and Attention Bypass to the parts of the model where they will have the most impact.

4 Building smaller and faster LLMs with depth pruning

This chapter covers

- Fundamentals and benefits of depth pruning
- Static vs. data-driven layer selection strategies
- Measuring layer contribution with cosine similarity
- Hands-on analysis using PyTorch hooks
- Research insights from "Shortened LLaMA" paper

In my first re-architectures I found myself wondering which transformer blocks I should remove. As is commonly done, I used the heuristic of always leaving the first and last blocks untouched, something that worked especially well in general-purpose models. But when creating a model with a specific task, I had doubts about whether I was removing the right blocks. That doubt points to a deeper question: what if the right blocks to remove depend on the data your model will actually process?

To answer that question, and help you make more informed block selection decisions, in this chapter we continue the work started in Chapters 2 and 3. We'll not only look at methods based on the model's static structure, but we'll also adopt a data-driven approach. You'll learn to measure each block's contribution depending on the data the resulting model will work with. To do this we'll need to "spy on" the model's internal activations.

The way we'll do this will be eminently practical. We'll calibrate the data-driven method using WinoGrande, a coreference resolution benchmark, so we can measure directly how well the selected blocks preserve performance on that exact task. We'll then put this selection head-to-head against the static approaches from section 4.2 on the same benchmarks.

You'll finish with the study of the papers "Shortened LLaMA" and "ShortGPT" where you'll contrast the solution developed in the chapter with the more academic approach of the papers.

Let's start by understanding the main characteristics of depth pruning.

4.1 Fundamentals and benefits of depth pruning

Depth pruning is the process of removing entire Transformer blocks. Well executed, it is an effective first step in rearchitecting that can reduce the cost of subsequent knowledge recovery. Its main benefits are reduced latency and lower memory consumption.

It’s important that you have a solid foundation of depth pruning, including the "why" of its advantages and limitations and not only the “how”. This foundation prepares you to tackle the central challenge later in the chapter: developing smart block selection strategies.

NOTE

In this book, we use a widely adopted terminology in the Transformers literature: a *block* is the complete repetitive unit (Attention + MLP), a *module* is a functional component (self_attn, mlp), and a layer is an individual operation (Linear, LayerNorm). Recent papers like "ShortGPT" use this same nomenclature (for example, "Block Influence"). In the code, you'll see that HuggingFace stores these blocks in `model.model.layers`.

4.1.1 How depth pruning works

Each block of a Transformer is made up of two main modules (figure 4.1): the attention and the MLP, accompanied by normalization layers, and alternatively other types of layers like dropout.

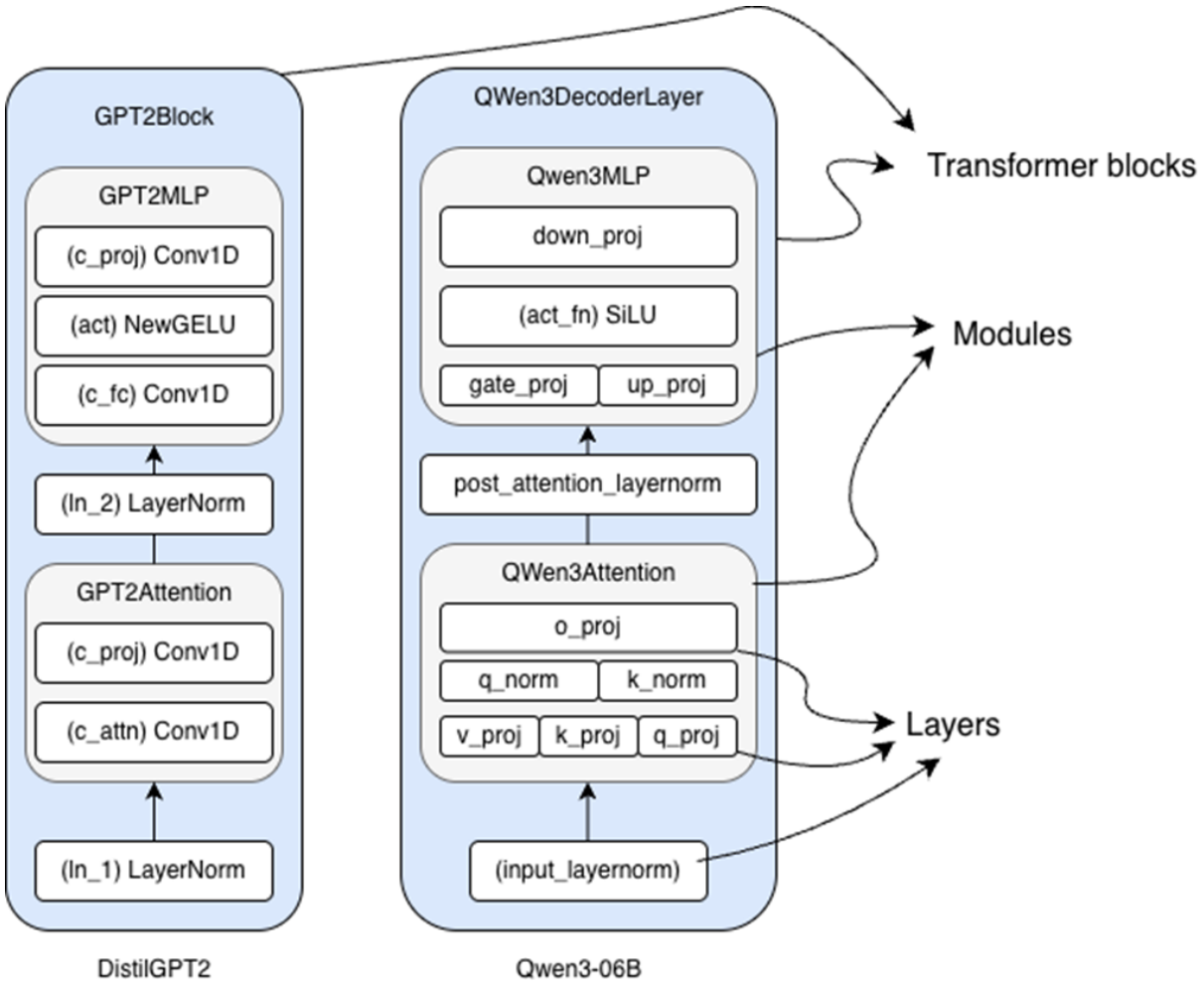


Figure 4.1 Structure of the transformer blocks of the DistilGPT2 and Qwen3-06B models. Each block contains two functional modules: Attention and MLP. In turn, each module is composed of individual computational layers such as linear projections, normalizations, and activation functions.

When you're depth pruning, you're removing an entire block. Along with the block, you're discarding a complete information transformation process that has been present throughout the entire training of the model. This is a very aggressive pruning method, but it's precisely this aggressiveness that gives the model the advantages we'll see next.

Removing a block alters the "shape" of the model. Language models can be deep and narrow, prioritizing a higher

number of sequential transformations, or wider, with a large processing capacity in fewer steps.

By removing blocks we're altering the vertical dimension of the model, transforming a deeper architecture into one with fewer blocks. Generally, models with greater initial depth, that is with more transformer blocks, tend to handle this technique better, since the functional redundancy between their numerous blocks allows the removal of some of them to have lesser impact.

4.1.2 The benefits: memory consumption and speed

We've already said that depth pruning can be an aggressive technique for the model. This aggressiveness is the key to its effectiveness. Surprisingly, the main benefit isn't in reducing calculations, but in reducing data movement in memory, as this is the real bottleneck in current LLM inference processes.

To understand this, we need to be aware that GPUs, which are not only expensive but also increasingly scarce, are often underutilized during inference. They rarely use more than 50-70% of their mathematical processing capacity because they spend most of their time waiting for data to arrive.

Depth pruning directly attacks this loading problem known as *memory-bound*. By eliminating complete Transformer blocks, you don't just reduce parameters: you eliminate entire sequences of memory access operations. Every block you remove means one less round of loading data from VRAM (the dedicated memory of the GPU), processing it, and writing the results back.

But we don't just improve by reducing the number of times data must be loaded from memory. Memory consumption is directly affected by the reduction of model parameters. In the Qwen3-06B model, the total set of transformer blocks occupies 58% of the model size, and each of its blocks occupies 2.1%. Every time we're eliminating a block we reduce the memory needed to load the model by 2.1%.

Even more crucial is the reduction that occurs in the KV-Cache, which stores the conversation's context. LLMs generate text token-by-token through a process called *autoregressive generation,* where each new token depends on all the previous ones. During this process, each layer must keep the attention states (Key and Value tensors) of all previously generated tokens in memory: this is the KV-Cache. This cache grows linearly with the sequence length and can exceed the size of the model's own parameters. By removing complete blocks, you also remove their attention layers, and with them their contribution to this cache growth. If you remove 25% of the layers, you reduce memory growth by 25% for each additional token you generate.

This reduction in memory requirements leads us directly to the gain we'll notice most: a considerable increase in inference speed. Fewer memory accesses mean less waiting time for the GPU. Therefore, we not only reduce the number of operations to perform by removing a block, but we also eliminate memory access bottlenecks that slow down the entire process.

However, remember that this gain in efficiency comes at a price of the eliminated knowledge. Just as the gain occurs from the number of blocks removed regardless of which ones they are, the loss of knowledge is linked to which blocks are removed (figure 4.2). The better you select which

blocks are redundant, the less knowledge will be lost and the simpler the information recovery process will be.

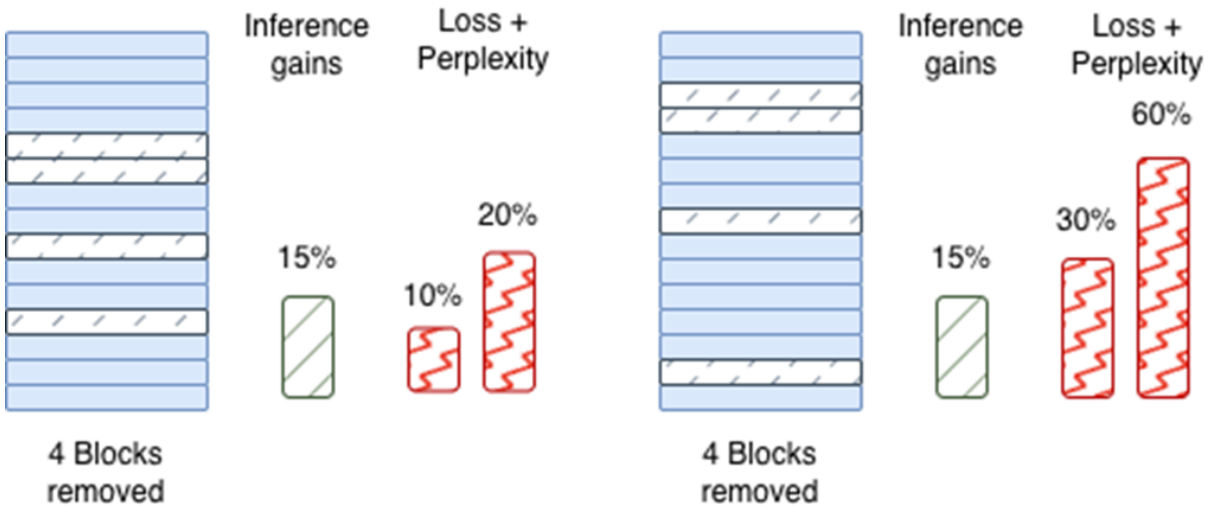


Figure 4.2 Impact of block selection on depth pruning. The gain in inference (+15%) is identical as we are removing four blocks in both scenarios. The performance cost varies drastically depending on which ones are removed.

The choice of the strategy for selecting the model blocks to remove is fundamental to optimizing depth pruning. A good choice minimizes damage and simplifies the recovery phase, saving us GPU time and allowing the model to recover more functionality with a lighter information recovery process.

We can employ different strategies for making this decision. We'll start with the most direct and simple approaches, known as *static* or "data-free." Then, we'll need a more sophisticated and data-driven method to achieve a pruning that suits our needs.

4.2 Static block selection

Static approaches are those based on analyzing the model's structure and initial weights. They don't analyze its behavior using real data. They stand out for their simplicity, zero computational cost for the selection process, and surprising

effectiveness. They are a solid choice when building general-purpose models or when task-specific data is not available.

4.2.1 Removing first, last, middle blocks

Block position-based approaches are very simple. As shown in the following listing, you only need a few lines of code. It's the system you used in chapter 2 where you removed the last two blocks of a model. This time we are going to remove four blocks from the middle of the model. The only difference is how to create the the list of layers to keep, removing the ones from the end was very simple you only had to create a smaller list, now you have to calculate which layers are the ones in the middle and create two lists of layers to then merge them and assign them to the new model.

NOTE

All the code in this chapter is available in the notebook CH04_NB01_lmeval.ipynb. The listings shown here contain the core implementation; the notebook also includes supporting utility functions.

This listing removes the four middle blocks. Alternatively, you can remove the first blocks or last blocks.

Listing 4.1 Remove the middle blocks

```
from copy import deepcopy

num_to_remove = 4
num_layers = len(model.model.layers)

mid_point = num_layers // 2
start_remove_index = mid_point - (num_to_remove // 2)
end_remove_index = start_remove_index + num_to_remove

print(f"Number of layers: {num_layers}")
print(f"Layers to remove {num_to_remove}.")
print(f"removing from {start_remove_index} to {end_remove_index -
1}")

pruned_model = deepcopy(model)

layers_before = pruned_model.model.layers[:start_remove_index]
layers_after = pruned_model.model.layers[end_remove_index:]

pruned_model.model.layers = layers_before + layers_after

print(f"Layers in pruned model: {len(pruned_model.model.layers)}")
print(f"Removed layers: {num_layers - len(pruned_model.model.layer
s)}")
```

What it gives us back:

```
Number of layers: 28
Layers to remove 4.
removing from 12 to 15
Layers in pruned model: 24
Removed layers: 4
```

The strategy of removing the middle blocks, implemented in listing 4.1, is based on the idea that these layers often contains redundancy, while the first ones learn more fundamental representations, and the last ones perform task-specific refinements. In a 28-layer model, removing four middle blocks preserves 85.7% of the depth while keeping both foundational and specialized transformations

intact. Alternatively, you can remove the last blocks, which perform more specialized transformations, or the first blocks, though this is less common.

These approaches have been improved over time through a preservation heuristic, which indicates that the first blocks and the last ones should be preserved. A commonly accepted approach, formalized in "Shortened LLaMA" (Kim et al., 2024) and discussed in depth in section 4.4, is to preserve the first four and the last two blocks, although it can vary depending on the size and number of blocks in the model.

Although this approach is more robust than simply removing the last or first blocks, it's still a static rule. Its main weakness is that it assumes a similar distribution across all models and for all tasks, an assumption that doesn't always hold true.

4.2.2 Removing by weight

A more sophisticated approach is one that uses the magnitude of the weights to infer the importance of each block. It’s assumed that blocks with smaller weights modify the data less, and therefore, their contribution to the model's final result is smaller. An aggregate metric is calculated for each block, and those with the lowest score are removed. In the following listing, we can see the code that calculates the weight of a model's blocks and shows us the four that theoretically contribute the least.

Listing 4.2 Remove the blocks with lower magnitude.

```
def calculate_layer_magnitude(layer):
    total_magnitude = 0
    for param in layer.parameters():
        total_magnitude += torch.norm(param).item()
    return total_magnitude

layer_magnitudes = []
for i, layer in enumerate(model.model.layers): #A
    magnitude = calculate_layer_magnitude(layer)
    layer_magnitudes.append((i, magnitude))

layer_magnitudes.sort(key=lambda x: x[1])  #B
layers_to_remove = [idx for idx, _ in layer_magnitudes[:4]]  #C

print(layers_to_remove)
```

#A Calculate magnitude for each layer.
#B Sort by magnitude.
#C Remove blocks with lower magnitudes.

For Qwen3-0.6B, the model used in this chapter's notebook, the blocks selected by this method are: [8, 2, 7, 6]. A different model will produce different results.

Note that one of the blocks selected for removal falls within those protected by the protection heuristic that marked the first four blocks and the last two as essential. The decision to apply the protection heuristic is ours, although if static methods are used, the most common thing is to keep it.

4.2.3 Static vs. weights

Static methods share clear limitations; the biggest weakness is their ignorance of the context. A block with small weights might have a low magnitude simply because the training data did not activate certain patterns, but that same block could be critical for a specific task. Furthermore, architectural assumptions like "the last blocks are

dispensable" don't have to be true in modern models or those that have gone through multiple fine-tuning stages. The very existence of the heuristic for preserving the last two blocks already indicates this problem exists.

These different static approaches, along with the data-driven approach we will explore next, can be clearly visualized in Figure 4.3, which compares the block removal patterns of each strategy.

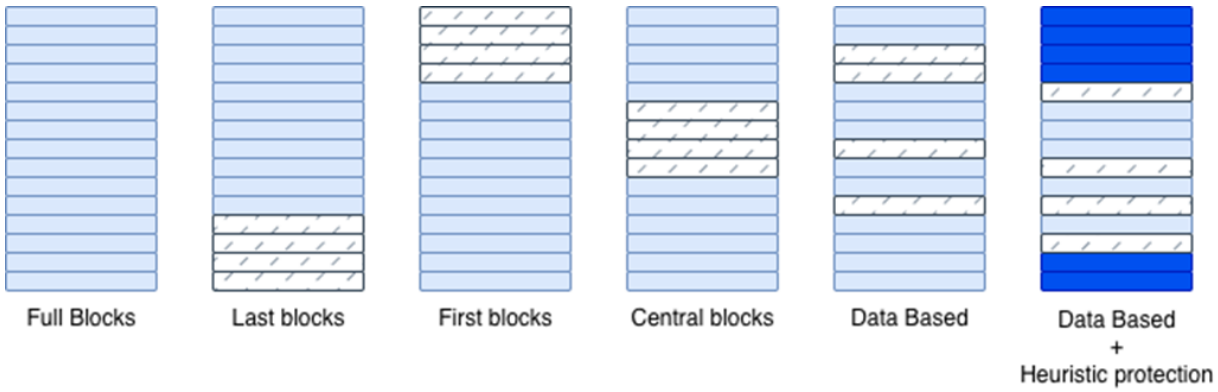


Figure 4.3 Comparison of block selection strategies for depth pruning. From left to right: The full base model. Static strategies that remove blocks based on their position: the first, the last, or the middle ones. A data-driven strategy, where the removed blocks are scattered according to their actual contribution to the task. The data-driven strategy combined with a protection heuristic, which preserves the initial and final blocks.

The problem gets worse because the function of the blocks is ignored, two blocks with similar magnitudes can have radically different roles. In short, general rules about layer importance can be completely wrong for your particular domain, removing exactly the capabilities you need to preserve.

These static approaches work well when building general-purpose models or when you lack task-specific data. For domain-specialized optimization, data-driven methods let you identify which blocks truly matter for your use case.

4.3 Data-Driven block selection

The limitations of static methods lead us to the conclusion that we need to observe the model in operation. The fundamental idea is that a block's importance isn't an absolute property of the model, but rather relative to the data and the task it has to solve.

To show this, we'll calibrate the analysis with WinoGrande (https://huggingface.co/datasets/allenai/winogrande), a coreference resolution benchmark where the model must identify which pronoun refers to which entity in a sentence, allowing you to identify which layers contribute least to this specific task.

To be able to analyze the dynamic behavior of the model, we'll need to "spy on" how each block transforms the information. For this we'll use PyTorch hooks, a mechanism that lets you attach a function to a layer so it executes automatically.

With hooks, you can get the data that enters each block and see how it has changed after the block's execution. You can compare the difference between the input and output of each of the blocks that make up the model. This gives you a metric that indicates which ones modify the information the most, and which ones keep it almost intact.

The key is identifying which blocks are "passive" for your specific context. Some blocks barely transform the information they receive; the representation that comes out is almost identical to what went in. If a block doesn't provide any significant transformations for your data or task, removing it will have minimal impact.

You're going to delve into dynamic analysis of an LLM's behavior. This milestone is important; you move beyond basing your decisions on the LLM's structure and start analyzing what happens in each step that is executed inside it and how it modifies the information.

You will work with Qwen3-0.6B, which is a very deep and narrow model. Before analyzing its behavior, let’s familiarize ourselves with its structure in the following listing.

Listing 4.3 Qwen3-0.6B structure

```
Qwen3ForCausalLM(
  (model): Qwen3Model(
    (embed_tokens): Embedding(151936, 1024)
    (layers): ModuleList( #A
      (0-27): 28 x Qwen3DecoderLayer( #B
        (self_attn): Qwen3Attention(
          (q_proj): Linear(in_features=1024, out_features=2048, bias
=False)
          (k_proj): Linear(in_features=1024, out_features=1024, bias
=False)
          (v_proj): Linear(in_features=1024, out_features=1024, bias
=False)
          (o_proj): Linear(in_features=2048, out_features=1024, bias
=False)
          (q_norm): Qwen3RMSNorm((128,), eps=1e-06)
          (k_norm): Qwen3RMSNorm((128,), eps=1e-06)
        )
        (mlp): Qwen3MLP(
          (gate_proj): Linear(in_features=1024, out_features=3072,
                                              ➥bias=False)
          (up_proj): Linear(in_features=1024, out_features=3072,
                                              ➥bias=False)
          (down_proj): Linear(in_features=3072, out_features=1024,
                                              ➥bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): Qwen3RMSNorm((1024,), eps=1e-06)
        (post_attention_layernorm): Qwen3RMSNorm((1024,), eps=1e-06)
      )
    )
    (norm): Qwen3RMSNorm((1024,), eps=1e-06)
    (rotary_emb): Qwen3RotaryEmbedding()
  )
  (lm_head): Linear(in_features=1024, out_features=151936, bias=Fals
e)
)
```

#A The 28 transformer blocks are inside model.layers.
#B Each Qwen3DecoderLayer is a complete transformer block with its attention module and its MLP module.

To discover which blocks are important for a specific task, you need three elements:

- A way to "spy on" the model's internal activations (PyTorch hooks).
- A metric to measure how much each block transforms the information (cosine similarity).
- Real data from your task to do the analysis.

You'll implement the entire process step by step, from the basic concepts to the complete analysis that will allow you to make informed decisions about which blocks to remove. By the end, you'll have a data-driven selection to set against the static methods from section 4.2, on the same benchmarks.

Let's start with PyTorch hooks.

4.3.1 Using PyTorch hooks

A *hook* is a mechanism that PyTorch offers that lets us attach a function to any module in a model, whether it's a transformer block or one of the layers that form it. This function will run automatically every time data passes through the block, so we can access it without having to modify the model's code or the libraries that run it.

The function that we're going to define to use as a hook must have a predefined form; in this case, it's going to receive three parameters:

- module: It's a reference to the module itself where we've attached the hook (for example `Qwen3DecoderLayer`, the transformer block of the Qwen3-0.6B model).
- input: It's a tuple containing the tensors that were passed as input to the module's `forward` method.

- output: It's the tensor (or tuple of tensors) that the `forward` method has returned.

This form of the hook works especially well for the use we want to give it, comparing the input with the output to assess the importance of the modifications that occur during the execution of the spied-on module.

To illustrate how it works, we're going to create a hook that shows us the shape of the tensors passing through the first block. Let's start with the definition of the function that will act as the hook:

```
def print_shape_hook(module, input, output):
    input_tensor = input[0] #A
    print(f"Module class: {module.__class__.__name__}")
    print(f"Module id: {id(module)}")
    print(f"Input shape:  {input_tensor.shape}")
    print(f"Output shape: {output.shape}")
```

#A input is a tuple, we take the first element (the hidden states).

This function is extremely simple. It prints available information, which enables us to see that the hook is executing.

The next step is to register this function in the module we want to "spy" on:

```
first_layer = model.model.layers[0]
hook_handle = first_layer.register_forward_hook(print_shape_hook)
```

First we get the module where we want to assign the hook. The transformer blocks are in the `model->layers` path shown in listing 4.3.

Then we register it with the `register_forward_hook` function, which returns a handle to the hook so we can remove it

when it's no longer needed. Now we need to test the hook's execution, for which we'll run a query on the model:

```
test_text = "He sat on the river bank to fish"
inputs = tokenizer(test_text, return_tensors="pt").to(device)
with torch.no_grad():
    outputs = model(**inputs)
```

The hook will print:

```
Module class: Qwen3DecoderLayer
Module id: 139523145412304
Input shape:  torch.Size([1, 8, 1024])
Output shape: torch.Size([1, 8, 1024])
```

The value of `Module class` confirms that we are hooked into a `Qwen3DecoderLayer`, one of the model's Transformer blocks. The `Module id` is a unique identifier for the object in memory.

The shape `[1, 8, 1024]` tells us that we are processing 1 sequence (batch size), with 8 tokens (the length of our tokenized sentence), and each token is represented by a 1024-dimension vector (the model's embedding size).

Now we must remove the hook, with `hook_handle.remove()`, which is a good practice if we want to keep working with the model we have in memory but we no longer need to analyze its behavior.

Hooks don't just let you observe; you can also modify the data. For example, you could multiply the activations by a factor, add noise, or even return zeros. Right now, we'll focus on using them to capture the input and output activations of each block, which is exactly what we need to calculate our importance metric.

4.3.2 Understanding cosine similarity

Now that we know how to capture the input and output activations of each block, we need a metric that tells us how much the information has changed inside the block we are measuring.

The information inside the model travels in a multi-dimensional vector; in the case of Qwen3-0.6B, 1024 dimensions. To compare two vectors of this dimensionality, we could calculate a simple distance between them, like the Euclidean distance. However, this metric is sensitive to both the magnitude (the "length" of the vector) and its direction. The problem is that the magnitude can change without the meaning changing, while the direction is what really encodes the semantics. Cosine similarity measures the angle between two vectors, which captures their direction and orientation.

Think of the vectors as arrows that start from the same origin (figure 4.4).

- If two arrows point in exactly the same direction, the angle between them is 0° and their cosine similarity is 1. They are identical in orientation.
- If they are perpendicular (pointing in totally unrelated directions), the angle is 90° and their similarity is 0.
- If they point in opposite directions, the angle is 180° and their similarity is -1.

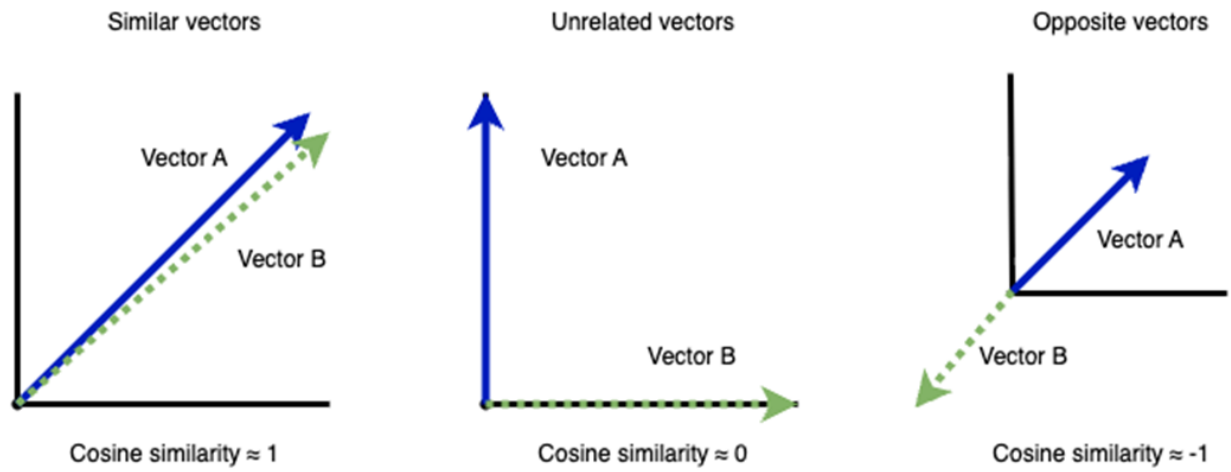


Figure 4.4 Three pairs of vectors from left to right: Vectors with similar meaning point in similar directions, unrelated vectors in different directions, vectors with opposite meanings in opposite directions.

In our case, if a block barely transforms the information, the input vector and output vector will point in very similar directions, giving us a similarity close to 1.0. If the block does a significant transformation, the vectors will point in more different directions, reducing the similarity.

Let's see an example of how it works. We will create the vectors for three sentences using the `sentence_transformers` library, and then we'll calculate the cosine similarity between them. Two of the sentences are semantically very similar, although with different lengths, while the third one is about a completely different topic.

Let's start by defining the sentences and creating the corresponding vectors:

```
from sentence_transformers import SentenceTransformer
import torch
import torch.nn.functional as F

modelst = SentenceTransformer('all-MiniLM-L6-v2') #A

sentences = [
    "The cat naps on the sofa.",
    "The feline is peacefully slumbering on the couch.",
    "The bus stops at the corner."
]

embeddings = modelst.encode(sentences, convert_to_tensor=True)
print(f"Embedding shape: {embeddings.shape}")
```

#A Load standard sentence embedding model.

We've selected `all-MiniLM-L6-v2` from the many pretrained embedding models available through the *sentence-transformers* library, because it offers an excellent compromise between size and performance, making it ideal for examples. However, the calculation method we're using is universal. The following steps to calculate cosine similarity using PyTorch's functional module will work with the vectors produced by *any* embedding model.

The output will show:

```
Embedding shape: torch.Size([3, 384])
```

Each of our three sentences has been converted into a 384-dimensional vector. Now we can calculate the cosine similarity between these vectors:

```
embeddings_normalized = F.normalize(embeddings, p=2, dim=1) #A

# Compute similarity matrix (cosine = dot product of normalized vect
ors)
similarity_matrix = torch.mm(embeddings_normalized,
                                            embeddings_normalized.T)
 #B

print(f"Sentence 1 vs 2: {similarity_matrix[0][1]:.4f}")
print(f"Sentence 1 vs 3: {similarity_matrix[0][2]:.4f}")
```

#A Normalize embeddings
#B Compute similarity matrix

Which returns:

```
Sentence 1 vs Sentence 2: 0.6106
Sentence 1 vs Sentence 3: 0.0563
```

The result is as expected: sentence one and two have semantically similar meanings although they have a different length, while sentence one and three have almost no similarity as indicated by their cosine similarity with a value close to 0.

The calculation is done in two steps. First, normalize the vectors. Second, calculate their dot product. The *dot product* is a simple operation that multiplies the corresponding elements of two vectors and sums all the results to get a single number. For example, for the normalized vectors [0.6, 0.8] and [1, 0.0], the dot product would be (0.6*1) + (0.8*0) = 0.6. This final number is directly its cosine similarity. A value between -1 and 1.

In the code we use a matrix multiplication (`torch.mm`) to calculate all combinations at once efficiently.

The next step is to use hooks to capture the activations of each block of the Qwen3 model, to decide which blocks contribute less to the calibration task.

4.3.3 Analyzing block contributions with WinoGrande

Now you've learned the two fundamental ingredients: capturing the model's activations using Pytorch hooks and measuring the cosine similarity between different vectors.

We’re going to put it together into a complete pipeline (figure 4.5) to identify which blocks of Qwen3-0.6B are most important when the model has to resolve coreference in WinoGrande-style sentences.

1. Assign hooks to each of the 28 blocks of QWen3-0.6B.
2. Iterate over the WinoGrande data, capturing the activations in each block.
3. Calculate the cosine similarity to know how much each block has transformed the information.
4. Aggregate the results to get a final importance score for each block.

With these steps, you'll have the importance scores you need to select which blocks to remove. The goal is to obtain a new model to put head-to-head against the models obtained with the static methods from section 4.2. Let's start by formatting the data.

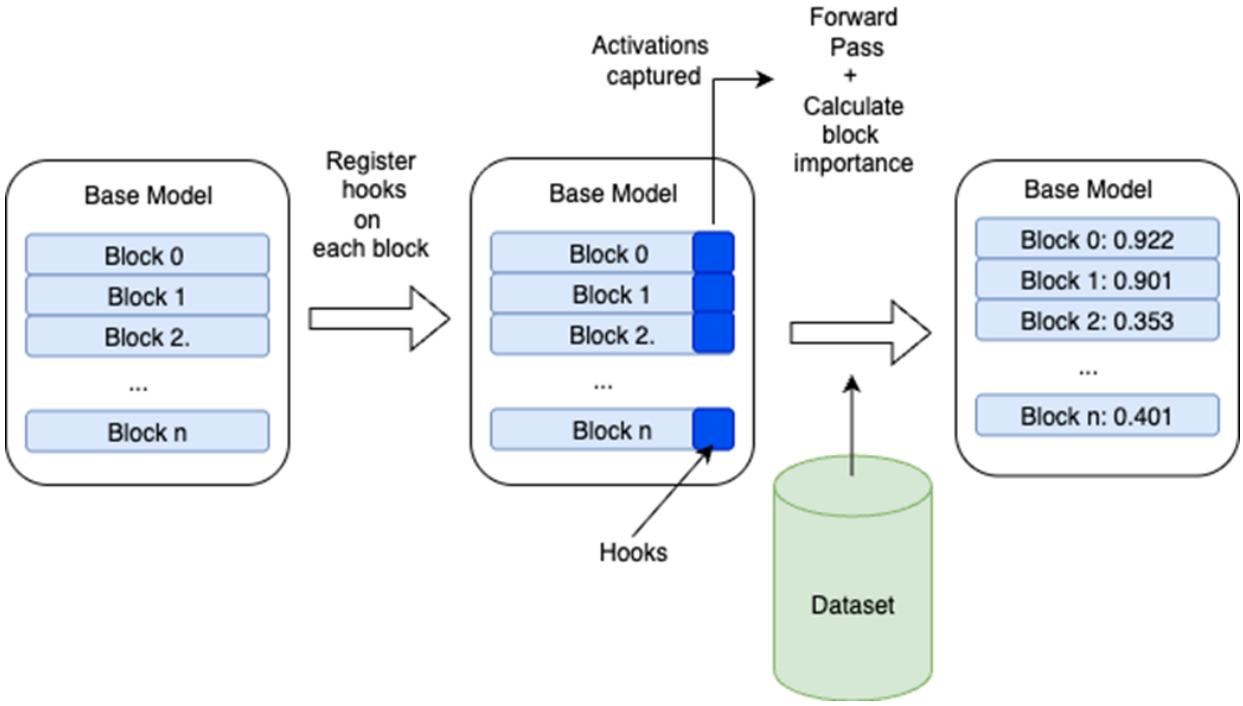


Figure 4.5 In the block evaluation pipeline, hooks are registered, a forward pass of the model is executed using the dataset, and the activations are captured to calculate the block importance. The result is a list containing the block identifier with the assigned importance.

FORMATTING THE DATASET

As introduced earlier, we'll calibrate the analysis with WinoGrande, a coreference resolution benchmark where the model must identify which pronoun refers to which entity in a sentence. The following listing converts each WinoGrande example into a single block of text that will actually flow through the model's blocks.

Listing 4.4 Formatting WinoGrande for calibration.

```
raw_winogrande = load_dataset('winogrande', 'winogrande_xl', split=
f'train[:{CALIBRATION_SAMPLES}]', trust_remote_code=True)

def format_winogrande(example):
    option1 = example['option1']
    option2 = example['option2']
    answer = example['answer']  # '1' or '2'
    answer_text = option1 if answer == '1' else option2

    example['text'] = (
        f"Sentence: {example['sentence']}\n"
        f"Option 1: {option1}\n"
        f"Option 2: {option2}\n"
        f"Answer: {answer}. {answer_text}"
    )
    return example

datawinogrande = raw_winogrande.map(format_winogrande) #A
```

#A Concatenates sentence, options, and answer into a single text block.

```
print(datawinogrande[0]['text'])
```

Which returns:

```
Sentence: Ian volunteered to eat Dennis's menudo after already havin
g a
bowl because _ despised eating intestine.
Option 1: Ian
Option 2: Dennis
Answer: 2. Dennis
```

Each example presents a sentence with a blank placeholder, two candidate entities to fill it, and the correct answer spelled out. This format mirrors how WinoGrande is presented as a benchmark task, with the answer written directly into the text instead of left for the model to predict.

ASSIGN HOOKS

The `setup_layer_hooks` function from the following listing is responsible for creating the hook and assigning it to each of the model's blocks.

Listing 4.5 Configuring hooks to capture activations.

```
def setup_layer_hooks(model):

    num_layers = len(model.model.layers)
    layer_inputs = {} #A
    layer_outputs = {} #A
    hooks = []

   def create_hook(layer_idx): #B
        def hook(module, input, output): #C
            input_tensor = input[0] if isinstance(input, tuple) else
input #D
            layer_inputs[layer_idx] =
➥input_tensor.detach() #E

            output_tensor = output[0] if isinstance(output, tuple) e
lse output #D
            layer_outputs[layer_idx] =
➥output_tensor.detach() #E
        return hook

    for i, layer in enumerate(model.model.layers):
        hooks.append(
            layer.register_forward_hook(create_hook(i))
        ) #F

    return hooks, layer_inputs, layer_outputs, num_layers
```

#A Empty dictionaries to store captured activations from each layer
#B Factory function that creates and returns a new hook function for a specific `layer_idx`.
#C The internal hook function that PyTorch will execute during forward pass
#D PyTorch sometimes passes arguments as tuples; we extract the tensor to ensure compatibility across different model architectures
#E Using .detach() we save the activations without gradients to free up GPU memory
#F We register a hook per layer that captures both input and output simultaneously

This function follows the same hook pattern you saw in section 4.3.1, with some refinements.

First we use a factory function: `create_hook`. This function isn't the hook itself; it's in charge of creating and returning the hook. But since we need to access a different block in each function, instead of having a different function for each block, we use the little trick of the factory function, as the created hook remembers the value of `layer_idx` it was called with. For example if we call `create_hook` twice, once with a value of 4 and once with a value of 8, it will return exactly the same code, but with the `layer_idx` variable replaced by the value it received as a parameter. In one, we will find `layer_inputs[4]` and in the other `layer_inputs[8]`.

The use of `.detach()` is also important. Creates a new tensor that shares the same underlying data but is detached from the model's calculation history (the computational graph). This prevents gradient accumulation and frees the GPU memory that would otherwise be allocated for storing gradients.

With these hooks registered, our function will now capture activations every time the model processes input.

CALCULATE THE COSINE SIMILARITY

Now we need a function that compares the activations captured by the hooks and calculates how much each block has transformed the information, returning a block importance score.

This function is `calculate_cosine_importance` shown in the following listing. Its goal is to calculate the average cosine similarity between the input and output vectors for the entire batch and convert it into an "importance" metric of 1 -

similarity. A high similarity (close to 1) means the block has barely changed the information, so its importance will be low (close to 0).

Listing 4.6 Calculating block importance.

```
def calculate_cosine_importance(input_tensor, output_tensor, attenti
on_mask, layer_idx, is_first_batch=False):
    if input_tensor.numel() == 0 or output_tensor.numel() == 0:
        return 0.0 #A

    similarities = F.cosine_similarity(input_tensor,
        output_tensor,
        dim=-1) #B

    attention_mask = attention_mask.to(similarities.device) #C
    valid_similarities = similarities * attention_mask
    num_valid_tokens = attention_mask.sum()

    if num_valid_tokens == 0: #D
        if is_first_batch:
            print(f"Warning: layer {layer_idx} contained only paddin
g tokens in this batch.")
        return 0.0

    mean_similarity = valid_similarities.sum() / num_valid_tokens #1

    if not torch.isfinite(mean_similarity):
        if is_first_batch:
            print(f"Warning: layer {layer_idx} generated non-finite
similarity values.")
        return 0.0

    return 1.0 - mean_similarity.item() #E
```

#A Empty tensors return zero importance.
#B Cosine similarity per token, along the hidden dimension: shape [batch, seq_len].
#C Zero out padding tokens so they don't count toward the average.
#D Guard against batches that are pure padding.
#E Convert similarity to importance.

The importance metric is a simple inversion of similarity: a 99% similarity (0.99), which indicates very little change, becomes an importance of 0.01. On the contrary, a similarity of 60% (0.60), which indicates a substantial transformation, becomes an importance of 0.40.

The `attention_mask` argument is what makes the average in #1 trustworthy. Padding tokens carry no real content, but the model still produces an output for them, usually nearly identical to the input. Including them would dilute the score with artificial near-perfect similarities unrelated to how the block treats the actual passage and question, dividing only by `num_valid_tokens` keeps the average focused on real tokens.

With the hook system and the measurement function ready, we are only missing the piece that orchestrates the whole process: the main function that iterates over the dataset.

ORCHESTRATING THE FULL PROCESS

The function orchestrating the whole process is `calculate_layer_importance_cosine()`. It coordinates the registration of hooks, the iteration over the data, the importance calculation, and the aggregation of final results:

Listing 4.7 Complete block importance analysis pipeline.

```
def calculate_layer_importance_cosine(model, dataloader, device):
    hooks, layer_inputs, layer_outputs, num_layers
➥= setup_layer_hooks(model) #A
    layer_importance_scores = {i: [] for i in range(num_layers)}

    with torch.no_grad(): #B
        for batch_idx, batch in enumerate(tqdm(dataloader,
                                               desc="Processing batche
s")):
            inputs = {k: v.to(device)
for k, v in batch.items()} #C

            model(**inputs)  #D

            for layer_idx in range(num_layers):
                if layer_idx not in layer_inputs or layer_idx
➥not in layer_outputs:
                    raise RuntimeError(f"Layer {layer_idx} Hook fail
ed.")
                block_importance = calculate_cosine_importance(
                    layer_inputs[layer_idx],
                    layer_outputs[layer_idx],
                    inputs['attention_mask'],
                    layer_idx,
                    is_first_batch=(batch_idx == 0)) #E
                layer_importance_scores[layer_idx].append(block_impo
rtance)
            # Clear storage for next batch
            layer_inputs.clear()
            layer_outputs.clear()

    [hook.remove() for hook in hooks] #F

    final_scores = {}
    for layer_idx, scores in layer_importance_scores.items():
        valid_scores = [s for s in scores if np.isfinite(s)]
        if not valid_scores:
            raise RuntimeError(f"Layer {layer_idx} not captured. Hoo
k failed.")
        else:
```

```
            final_scores[layer_idx] = np.mean(valid_scores)
    return final_scores
```

#A Set up hooks and storage dictionaries.
#B Disable gradient tracking to save memory.
#C Move all batch tensors to the specified device (GPU in the sample notebook) for processing.
#D Trigger the hooks with the model's forward pass.
#E Calculate importance for each block by calling calculate_cosine_importance()
#F Remove hooks once capture is done.

The function orchestrates the complete workflow: hooks capture activations during the forward pass, cosine similarity measures transformation at each block, and scores are aggregated across batches to produce the final ranking.

NOTE

It's important to mention that this code works well for experimenting since the hooks return the tensors and you can use them to calculate different metrics. To use it in production, you could move the importance calculation to the hook, which would save us from keeping the tensors in this function, and we'd only need to keep the scores to aggregate them at the end.

Now you're ready to run the analysis:

```
winogrande_importance = calculate_layer_importance_cosine(model,
    dataloaderwinogrande,
    device)
print("\nWinoGrande importance:")
print_sorted_importance(winogrande_importance)
```

With this, we'll get a dictionary containing the block identifier and its importance score.

An example of the data returned in `winogrande_importance`:

```
WinoGrande importance:
Layer  0: 0.947226
Layer  1: 0.129674
Layer  2: 0.129453
...
Layer 13: 0.044656
Layer 23: 0.031227
Layer 24: 0.029947
Layer 26: 0.025137
Layer 25: 0.025061
```

Next, we'll analyze these results in detail to decide which blocks are the best candidates for removal. You'll find the complete set of scores in the chapter's notebook.

4.3.4 Choosing the blocks to discard

Now we have all the necessary information to remove the least critical blocks: those that will minimally impact our model's performance on our specific data.

The decision is not straightforward. Although we have the importance data for each block according to the change they produce on the information, it’s important to consider this data as a guide and not an absolute truth.

The importance scores tell you which blocks transform the information more or less, but they don't capture all the factors that influence the model's final performance. For example, some blocks with low scores might be performing subtle but critical transformations that only manifest in specific cases of the dataset.

Preserving the first and last blocks of the model is always good practice and part of the protection heuristic. A second common safeguard, avoiding the removal of adjacent blocks, stands on weaker ground. The intuition is that removing two consecutive blocks takes out a longer uninterrupted section

of the model and might do more damage, but it is not as widely established as protecting the first and last blocks. We treat it here as an option to study rather than a fixed rule, and you will see how a protected selection compares against an unprotected one when we evaluate the models.

Figure 4.6 compares the static magnitude method from section 4.2 and the data-driven cosine similarity calibrated on Winogrande. Both are normalized for comparison. The contrast is sharp. Block 0 dominates the cosine ranking with a score far above any other block, consistent with its role as the first encoder of raw token representations. The last blocks, however, tell a different story depending on which lens you use: magnitude ranks them among the most important in the model, while cosine shows they barely transform information when processing Winogrande prompts.

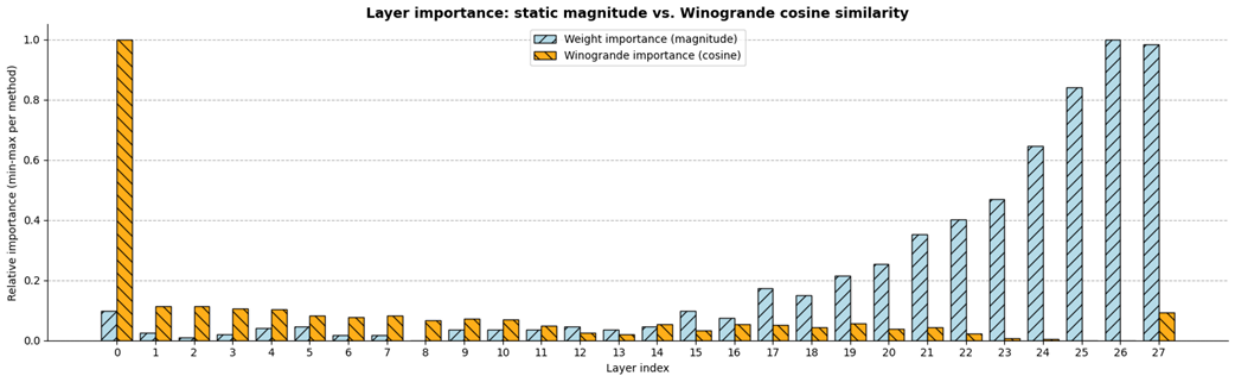


Figure 4.6 Layer importance across all 28 blocks of Qwen3-0.6B: static magnitude (weight-based) versus data-driven cosine similarity calibrated on Winogrande. Both metrics are normalized to the [0,1] range. Block 0 dominates the cosine ranking by a wide margin.

In Table 4.1 we find the six most important blocks and the six least important ones. As we can see, the block protection heuristic is corroborated by the data, especially for the first blocks, while for the last ones it holds for the last block (27) but not so much for the second-to-last (26).

Table 4.1 Block importance scores

Highest contribution	Score	Lowest contribution	Score
Transformer block 0	0.947	Bock 25	0.025
Transformer block 1	0.130	Bock 26	0.025
Transformer block 2	0.129	Bock 24	0.030
Transformer block 3	0.122	Bock 23	0.031
Transformer block 4	0.120	Bock 13	0.045
Transformer block 27	0.111	Bock 22	0.046

These results vary depending on the calibration dataset used, within reasonable margins: Running the same analysis on WikiText (available in CH04_NB02_wiki.ipynb) confirms the convergence at the top: the six most important blocks are exactly the same in both datasets. At the bottom, when selecting four blocks for removal, WinoGrande puts block 26 in second position, making it a removal candidate, while in WikiText block 26 drops out of the bottom positions and doesn't appear as a layer to remove, being replaced by block 13, a block from the middle part of the model.

Table 4.2 Block selection varies with calibration data

	WinoGrande calibration	WikiText calibration
Blocks removed	[25, 26, 24, 23]	[25, 24, 13, 23]
Key difference	Cluster in final blocks	Preserves more of the final blocks, includes one from the middle

The six most important blocks are identical in both cases. The difference appears at the bottom: WinoGrande removes the second-to-last block and clusters all removals in the final zone, while WikiText preserves a bigger final block and reaches into the middle instead. Same method, different data, different model.

Looking at the bottom of Table 4.1, all four removal candidates cluster in the same zone: blocks 23, 24, 25, and 26. Left unconstrained, the algorithm would select exactly those four. The standard heuristic, protecting the last one or two blocks, only partially addresses this: it saves block 26, but blocks 23, 24, and 25 remain candidates and they form a consecutive group at the tail of the model.

This is where adjacent protection becomes relevant. By preventing the selection of consecutive blocks, we force the algorithm to look beyond that cluster and distribute the removals across the model, producing a different block selection. Whether that produces better results is something we will examine when we evaluate the models.

The `select_layers_to_prune` function in Listing 4.8 lets you control both safeguards independently, so you can compare the two strategies directly.

Listing 4.8 Select blocks to remove.

```
def select_layers_to_prune(importance_scores,
                           num_layers=2,
                           heuristic_protection=True,
                           adjacent_protection=True):

    sorted_layers = sorted(importance_scores.items(), key=lambda x:
x[1])

    selected = []
    for layer, score in sorted_layers:
        if heuristic_protection and layer in [0, 1, 2, 3, 26, 27]:
            continue

        if adjacent_protection and any(abs(layer - l) == 1 for l in
selected):
            continue

        selected.append(layer)

        if len(selected) >= num_layers:
            break

    return selected
```

This simple function sorts the dictionary in reverse order of importance and can apply the protection heuristics that prevent deleting the first and last blocks or consecutive blocks.

We are going to use it to obtain two selections we will add to our comparison, one without heuristic or adjacent protection and one with both active:

```
winogrande_layers_2_remove = select_layers_to_prune(
    winogrande_importance,
    num_layers=4,
    heuristic_protection=False, adjacent_protection=False)

winogrande_protected_layers_2_remove = select_layers_to_prune(
    winogrande_importance,
    num_layers=4,
    heuristic_protection=True, adjacent_protection=True)
```

Which returns [25, 26, 24, 23] for the unprotected selection and [25, 23, 13, 15] for the protected one.

With this data we can use `prune_model` from OptiPFair library, which encapsulates a similar pruning logic you implemented in Listing 4.2, to obtain two different models:

```
from optipfair import prune_model

winogrande_model = prune_model(
    model=deepcopy(model),
    pruning_type="DEPTH",
    layer_indices=winogrande_layers_2_remove,
    show_progress=True,
)

winogrande_model_protected = prune_model(
    model=deepcopy(model),
    pruning_type="DEPTH",
    layer_indices=winogrande_protected_layers_2_remove,
    show_progress=True,
)
```

Both models have the same number of blocks removed, but the selection strategy differs. Now let's see whether that difference matters when we put them against the static methods on the benchmarks.

4.3.5 Analysis of the benchmarks

In the notebook we've prepared a fairly complete set of models to compare different results. We've created five models with static pruning, one guided by magnitude and two data-guided using the WinoGrande dataset. We've evaluated all eight models, plus the base model, on different standard benchmarks, including WinoGrande and LambadaOpenAI.

Table 4.3 Benchmark results

Model Layers removed	WinoGrande	WinoGrande retention	Lambada	Lambada retention
Base	0.566	100%	0.404	100%
Mid. blocks. [12, 13, 14, 15]	0.544	96.1%	0.257	63.7%
Mid. blocks adjacent protection [11, 13, 15, 17]	0.502	88.7%	0.140	34.6%
Last blocks [24, 25, 26, 27]	0.522	92.2%	0.0363	9%
Last blocks protected [23, 24, 25, 26]	0.545	96.4%	0.0413	10.2%
Last blocks protected 2 [22, 23, 24, 25]	0.548	96.8%	0.224	55.5%
Magnitude [8,2,7,6]	0.501	88.6%	0.0128	3.2%
Data guided [25, 26, 24, 23]	0.545	96.4%	0.0413	10.2%
Data guided protected [25, 23, 13, 15]	0.550	97.2%	0.2954	73.1%

Before diving into the results, here is a brief description of each model:

- **Mid. blocks** removes the four central blocks [12–15], the positional strategy from section 4.2.
- **Mid. blocks adjacent protection** also removes central blocks but skips consecutive ones, producing a scattered selection [11, 13, 15, 17].

- **Last blocks** removes the four final blocks [24–27].
- **Last blocks protected** shifts the window one position inward, preserving block 27 [23–26].
- **Last blocks protected 2** shifts it two positions inward, preserving blocks 26 and 27 [22–25].
- **Magnitude** uses the blocks selected in section 4.2.2 based on their weight magnitude [8, 2, 7, 6].
- **Data guided** uses Winogrande cosine importance with no protection, selecting the four least important blocks [25, 26, 24, 23].
- **Data guided protected** applies the same method but protects the last two blocks and avoids consecutive removals, producing [25, 23, 13, 15].

WINOGRANDE PERFORMANCE

We can observe how WinoGrande is a fairly resistant benchmark, with its fluctuations but maintaining good performance compared to the base model. This is due, in part, to the fact that the performance is close to chance: in WinoGrande you have to choose between two options, and a model that simply guesses would score close to 50%. Even so, we can observe a clear advantage for models that preserve the last layers. Worth highlighting is the good performance of the static version that removes the middle layers, especially when compared to the drop of the model where middle layers are removed while avoiding contiguous blocks.

The dynamic analysis has pushed us toward removing final blocks, but respecting the last one, which never emerged as a removal candidate. The result improved further when we applied both protection heuristics.

LAMBADA: THE STRESS TEST

WinoGrande's binary nature sets a floor close to 50%, compressing the range of observable degradation. In contrast, Lambada has no such floor. It measures the model's ability to predict a final word that depends on long-range context, and a model that loses that capability falls close to zero.

In the table we can see how for Lambada the models that remove final blocks collapse: last blocks (9%), last blocks protected (10.2%), and data guided unprotected (10.2%) all fall to roughly the same floor. Models that avoid the final zone fare significantly better: data guided protected leads at 73.1%, followed by mid blocks at 63.7%. A behavior that contrasts with what was observed with WinoGrande where the preservation of the last blocks maintained performance better.

The static middle block results add another layer to the picture. Mid blocks, which removes four consecutive central blocks, achieves 63.7% Lambada retention, the second best result after data guided protected. But mid blocks adjacent protection, which applies the same logic while avoiding consecutive removals, drops to 34.6%. Spreading the removal across non-consecutive central positions turned out to be more damaging than taking out a clean consecutive chunk.

The best overall result is obtained applying both protection heuristics to the data guided model, distributing the removal between the late and middle zones. That distribution seems to preserve long-range dependency capability, with the final blocks retaining connectivity, and the middle blocks absorbing the structural cost. It is worth noting that data guided unprotected selected exactly the same blocks as last blocks protected, just in a different order. On a 28-block

model, the cosine signal was not strong enough to escape the obvious cluster.

Analyzing wikitext dataset we have obtained some different numbers (CH04_NB02_wiki.ipynb), the same method without protection selects [25, 24, 13, 23] instead of [25, 26, 24, 23], with block 13, from the middle of the model, replaces block 26. A combination very similar to the one that has worked best with Lambada and WinoGrande. A different dataset produces a different signal, and that signal is already finding solutions that pure position cannot. In larger models with more blocks and more opportunity for specialization between them, this effect becomes more pronounced, and that is where data-driven methods take their real value over static heuristics.

BENEFITS IN PERFORMANCE

After having analyzed the impact of pruning on knowledge, we still need to measure the benefit in performance. To do this, we will evaluate three key inference metrics (see table 4.4):

- **Inference time**: The total time to generate a fixed-length response.
- **Time to first token (TTFT)**: The time it takes the model to generate the first token. It is a crucial metric for the user's perception of speed in interactive applications.
- **Throughput**: The number of tokens generated per second, a direct indicator of computational efficiency.

Table 4.4 Inference performance benchmarks

Performance	Inference Time	TTFT	Throughput
Base model	2.834s ± 0.041s	64.49ms ± 1.0ms	17.65 tokens/s ± 0.26
Data guided	2.429s ± 0.009s (+14.3%)	56.30ms ± 4.05ms (+12.7%)	20.59 tokens/s ± 0.08 (+16.7%)
Data guided protected	2.426s ± 0.010s (+14.4%)	54.40ms ± 1.78ms (+15.6%)	20.61 tokens/s ± 0.09 (+16.8%)

The table data was obtained on a T4 GPU in the Google Colab environment, using GPU warmup and 10 records from each dataset. If you repeat the experiment, the numbers will vary slightly, but not the trend. You can see a clear improvement in a range between 12% and 17% across all metrics. Both pruned models show virtually identical inference performance, as expected: what determines the speed gain is the number of blocks removed, not which ones. It's important to note that we're using a very small model that doesn't cause significant stress on the GPU. Improvements are usually more visible in larger models that move much more data in memory.

4.4 From paper to practice

The two papers that have been the foundation of the chapter are "Shortened LLaMA: Depth Pruning for Large Language Models with Comparison of Retraining Methods" by Kim et al. (2024) and "ShortGPT: Layers in Large Language Models are More Redundant Than You Expect" by Men et al. (2024).

The first paper shows that depth pruning is more effective than width pruning to accelerate inference in small batch scenarios. Their analysis confirms our premise: removing entire blocks not only reduces parameters, but the costly

cycle of reading and writing to memory for each discarded block is eliminated, which results in a direct improvement in latency.

Their results show significant speed gains. For example, when pruning 35% of a LLaMA-7B model, their depth pruning method reduced latency by 33% (from 2.4s to 1.6s) and increased throughput by 49% (from 53.7 to 80.1 tokens/s), while width pruning methods actually slowed down inference. This validates depth pruning as the right strategy if one of the main goals is improving inference performance.

The main difference we found between this paper and the approach followed in the chapter is the method for choosing which blocks to remove. In the paper, they mainly use two metrics: Taylor and Perplexity. To calculate the importance of different blocks using perplexity, they remove them from the model one by one and measure how much the model's perplexity worsens compared to the original model with all its blocks. A block that, when removed, shoots up the perplexity is critical; one that barely affects it is expendable. In practice, this means that to analyze a 28-block model, we'd need to run inference on our calibration dataset 28 times, once for each block we temporarily remove.

The other proposed metric, the Taylor approximation, estimates a block's importance by measuring how much the model's error (the loss) would increase if that block were removed. It is computationally more efficient than calculating perplexity, as it only requires one forward/backward cycle. However, the need to calculate gradients adds computational and implementation overhead compared to our approach, which only needs a pure forward inference pass and saves both processing and memory by not requiring gradient calculation.

This leads us to look for a metric that is commonly recognized and at the same time lighter to compute, like cosine similarity. The second paper, “ShortGPT,” introduces the metric they call "Block Influence" (BI). This is where our work directly connects with the research: the BI metric is calculated as 1 — the cosine similarity between the block's input and output. It’s a closely related metric you implemented in the `calculate_cosine_importance` function.

The results from ”ShortGPT” validate why this metric is so powerful. In their comparison (Table 2 of the paper), when pruning 27.1% of Llama2-7B, their method retained 86.3% of the original model's average performance (obtained by averaging 14 different benchmarks). Worth noting that this figure is obtained without knowledge recovery, directly after the pruning process.

In comparison, other structured pruning methods like LLMPruner or SliceGPT barely reached 72.8% and 68.7% retention, respectively. This is the quantitative proof that a simple metric, based on a single forward pass, can outperform much more complex approaches, which require gradient information or multiple iterations.

The paper also shows that this metric is a reliable indicator of block importance and that it outperforms static methods. Specifically, the paper's data demonstrates this correlation: for Llama2-7B, intermediate blocks with low BI scores (around 0.05) barely increase perplexity when removed, while initial blocks with high BI scores (>0.4) shoot it up. What you implemented in section 4.3 isn't an ad-hoc solution: it's a metric backed by formal research and validated on models up to 70B parameters.

In the chapter we've also seen a protection heuristic that preserved the first and last blocks of the model. The two papers approach this fact differently; in "Shortened Llama"

we find the explicit heuristic: they protect the first four and last two blocks of the model from pruning. While in "ShortGPT" they trust that this protection is provided because the higher importance of these blocks will make them get a high BI and therefore they'll be protected by the calculation process itself.

A crucial point that both papers address is the need for a knowledge recovery process, which we already have planned in our rearchitecting pipeline to mitigate the performance drop. In "Shortened LLaMA" they explore two main paths: Low-Rank Adaptation (LoRA) and Continued Pretraining (CPT), which is very similar to full training. Their conclusion is clear: while LoRA is sufficient for moderate pruning, CPT is better for recovering performance in models with aggressive pruning (above 40%). For its part, "ShortGPT" opts for a post-training strategy that consists of replacing the removed blocks with lightweight MLP layers and retraining the modified model, also demonstrating significant performance recovery.

In this book we'll use Knowledge Distillation for recovery rather than LoRA, following the approach present in other papers like NVIDIA's Minitron and Mistral's Ministral 3, where the goal is recovering general capabilities across the full distribution, not just adapting to a specific task. We'll cover this in detail in chapter 6.

Throughout the next chapters, our main tool for recovery will be Knowledge Distillation. We'll reserve LoRA not as a generic recovery technique, but as a tool for domain specialization, as you'll see in chapter 7.

The adopted solution in this chapter is mixed, and although it's supported by different studies, we didn't find one that fits 100% with the development carried out. This is a very

normal fact; we take from the studies what best suits our needs.

Many times we are not looking for a new technique. We are looking for a validation of our idea. In this case, the "ShortGPT" paper presents experiments carried out on 70B parameter models that are not simple or cheap to replicate, and that validate the techniques you have used in the block selection of our small 0.6B model. Thanks to the paper, you know that this same technique scales well.

4.5 Hands-on lab

Now that you've learned how to analyze block contributions and perform data-driven pruning, it's time to experiment. These exercises will help you develop intuition about when different selection strategies work, how aggressive you can be with pruning, and how to optimize the process for production use.

- Compare static versus data-driven block selection:
  - What to do: The magnitude model is already created in section 4.2.2 with blocks [8, 2, 7, 6]. Compare its benchmark results against `winogrande_model` and `winogrande_model_protected` from the chapter.
  - Questions to answer: How does magnitude-based selection compare to data-driven selection on WinoGrande and Lambada retention? Does the static magnitude approach preserve any advantage over the data-guided methods, or does it fall behind consistently across all benchmarks?
- Disable heuristic protection:
  - What to do: The chapter already shows the result of disabling both protections simultaneously on WinoGrande, producing [25, 26, 24, 23]. Now

experiment disabling them one at a time: first only `heuristic_protection=False`, then only `adjacent_protection=False`. Evaluate each resulting model on WinoGrande and Lambada.
    - Questions to answer: Which protection has more impact on block selection? Does disabling only one of them produce a result closer to the fully protected or fully unprotected version? Can you find a combination that outperforms winogrande_model_protected?
- Progressive degradation analysis:
    - What to do: Create a systematic experiment that removes 2, 3, 4, 5, and 6 blocks from the model, always using data-driven selection with WinoGrande importance scores and both protections active. Evaluate each configuration on WinoGrande and Lambada retention and visualize the degradation curve for both benchmarks.
    - Questions to answer: Is the degradation linear, or does it accelerate at a certain point? Can you identify a "breaking point" where the model loses critical capabilities?
- Optimize the hook for production:
    - What to do: Refactor the analysis pipeline to make it more memory-efficient. Modify the hook function so that it calculates the importance score directly inside the hook, instead of storing the large input and output tensors. The main function will then only need to retrieve the final scalar scores.
    - Questions to answer: Does the computation time change significantly? What's the trade-off between memory efficiency and debugging capability (since you can no longer inspect the raw activations after the forward pass)?

- Scale to a larger model:
  - What to do: Repeat the core experiments from this chapter using a larger model like Qwen/Qwen3-1.7B. Calibrate the importance analysis with Winogrande and compare the block importance patterns against those obtained with Qwen3-0.6B. Evaluate the pruned models on WinoGrande and Lambada retention.
  - Questions to answer: Do the importance patterns differ between the two models? Are the final blocks still the least important for WinoGrande, or does a larger model show more distributed importance? When you prune the same percentage of blocks, does the larger model maintain better Lambada retention? Does the data-driven method find more distinctive solutions in a larger model, escaping the obvious clusters that appeared in the 28-block model?

NOTE

Analyzing a larger LLM will take significantly longer than Qwen3-0.6B due to its size. Consider reducing `RECOVERY_SAMPLES` to 500 or using a smaller batch size if you encounter memory issues. The insights about scalability are valuable even with fewer samples. If you're using Colab's free tier (T4 GPU with ~15GB VRAM), you will not be able to keep multiple model copies in memory simultaneously, but you can calculate the metrics in separate notebook runs, save them and finally aggregate them to obtain the charts.

4.6 Summary

- Depth pruning eliminates entire Transformer blocks from a model. While aggressive, this approach targets the real bottleneck in LLM inference: data movement between memory and compute units, not just computational operations.
- The primary benefit of depth pruning is reducing memory-bound operations. Each removed block eliminates a complete round of loading data from VRAM, processing it, and writing results back, directly improving inference latency and throughput.
- Static block selection strategies (position-based or magnitude-based) are simple and computationally free but ignore how the model actually processes your specific data. The common heuristic of preserving the first four and last two blocks provides some safety but remains a general rule.
- PyTorch hooks allow you to intercept and capture the input and output activations of any module during inference without modifying the model's code. Use `register_forward_hook()` to attach a function that receives the module, input, and output tensors.
- Block importance depends on the calibration data and task. In this small model, data-driven methods converge toward the final blocks, but the calibration data shapes the selection at the margins, enough to produce measurable differences in task performance and an effect that grows with model size.
- Research papers "Shortened LLaMA" (Kim et al., 2024) and "ShortGPT" (Men et al., 2024) validate depth pruning as superior to width pruning for latency reduction and confirm that cosine-based importance metrics are effective across models up to 70B parameters.

5 Shaping model architectures via width pruning

This chapter covers

- Understanding width pruning trade-offs
- Creating specialized task models
- Building models using your own data
- Measuring speed, energy, and reasoning

The most direct structural pruning technique is depth pruning, which is especially useful if your main concern is inference performance. It's a very aggressive technique that removes a complete Transformer block.

A more refined technique is to target specific neurons in a Multi-Layer Perceptron (MLP) module with a Gated Linear Unit (GLU) structure, the gating mechanism we explored in chapter 3. It's a technique that can be used with precision, not just to improve performance, but, as you'll see, to change the model's personality. The ideal candidate for this type of optimization are wide models. In this chapter, we'll use the Llama-3.2-1B model with a x4 expansion in its MLP layer.

We'll focus mostly on the advantages this technique can bring in terms of performance, but its uses go beyond that. To measure the general impact on performance, capabilities, and consumption, we'll introduce a more complete

benchmarking system (whose technical details are in the appendices) that we'll use throughout the rest of the book.

We'll approach this technique from two complementary angles:

1. **Static selection**: First, you'll use a strategy based on weight magnitude range. Assuming that neurons with smaller weight ranges contribute less to the network's expressiveness. An assumption we'll put to the test in section 5.4, where you'll prune the most important neurons instead of the least important ones and measure the result.
2. **Data-driven selection**: Next, you'll use PyTorch hooks to spy on the internal activations of each neuron in one of the layers of the GLU structure and thus adjust the MLP block's behavior to your data's interests.

5.1 Selecting static neurons

Before designing a neuron selection system, it’s helpful to understand how the GLU structure works in the MLP module. Figure 5.1 shows the Llama-3.2-1B model and its MLP layers that we’ll be targeting for width pruning.

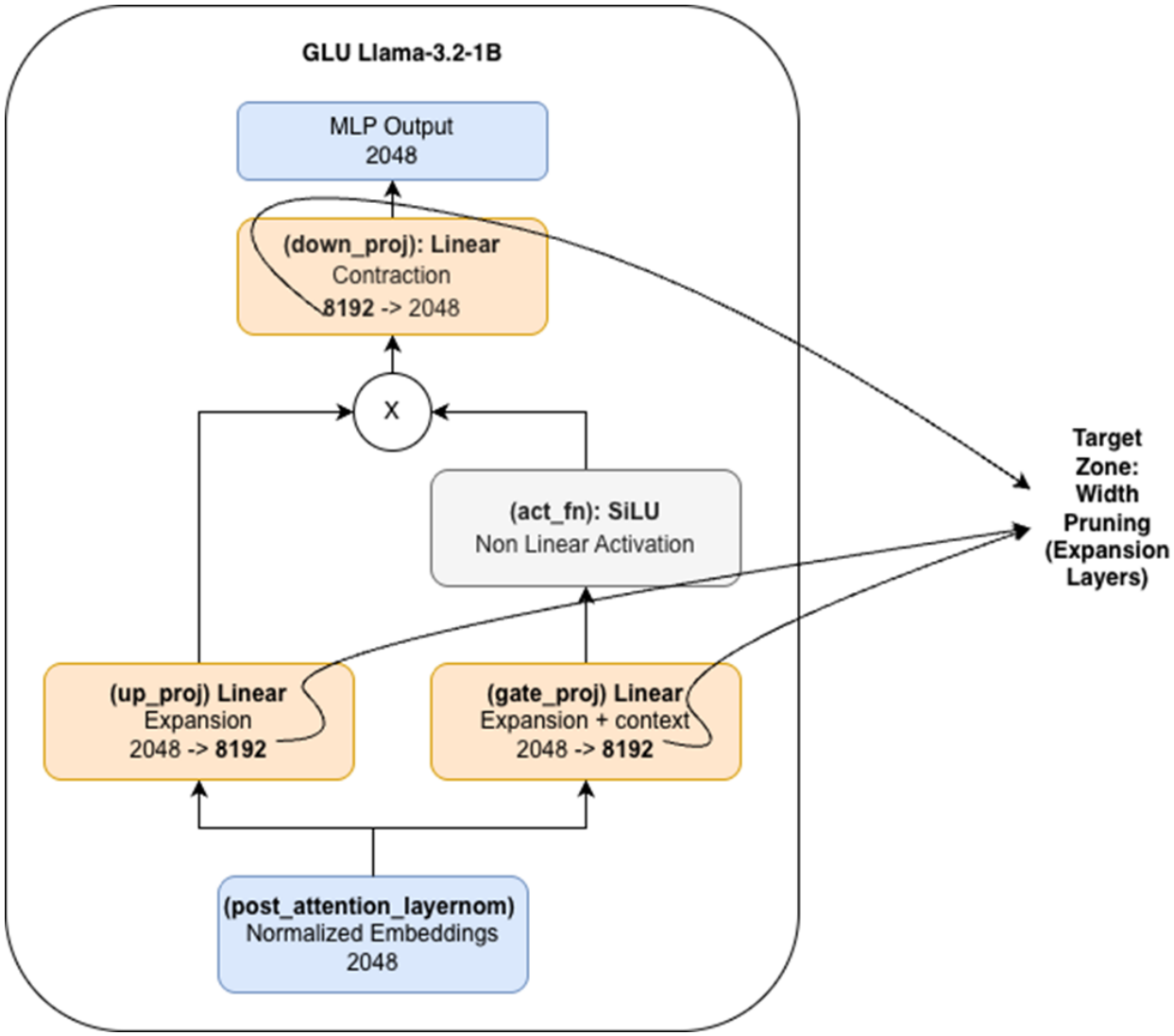


Figure 5.1 The complete GLU structure. Information enters with dimension hidden_size (2048 in Llama-3.2-1B) and expands to intermediate_size (8192) via gate_proj and up_proj. The gate_proj output, modulated by SiLU, is element-wise multiplied with up_proj's output. The result is contracted back to hidden_size by down_proj.

NOTE

SiLU (Sigmoid Linear Unit), also known as Swish, is the activation function used in modern GLU architectures. Like the GELU we encountered in Chapter 3, it acts as a "switch with regulator" that controls which neuron signals pass through and with what intensity. This non-linear transformation is what allows the network to learn complex patterns beyond simple linear relationships.

The size we're going to modify will be the expansion one, meaning the `intermediate_size`. This way we won't break the connection between modules. The attention layer of the next transformer block will still receive a tensor of size 2048, just as it expects. It's internal surgery to the MLP module.

The trick is in how we decide which neurons to remove. This time you're going to use a peak-to-peak magnitude range based method, similar to calculating the magnitude of a block's weights to know which ones you could eliminate (similar to the technique in chapter 4). But this time the calculation is more complicated.

Neurons don't work in isolation. Remember that `gate_proj` and `up_proj` operate as a coordinated pair. The output of `gate_proj` modulates the output of `up_proj` before the information reaches `down_proj`. If we remove a neuron from one projection, we must remove its counterpart in the other; otherwise, we break the gating synchronization.

The importance metric must consider both weights simultaneously: a neuron can have modest magnitudes in `gate_proj` but high ones in `up_proj`, or vice versa. We need a measure that captures the combined contribution.

$$\text{importance\_score} = (\max(W\_\text{gate}) + |\min(W\_\text{gate})|) + (\max(W\_\text{up}) + |\min(W\_\text{up})|)$$

The total distance between the most positive and most negative weight of that neuron is measured, since a neuron with a very high range has a very large capacity to generate significant activations.

By adding the ranges of both projections, we get the total transformation potential of the pair. The pairs with the lowest potential are our best candidates for pruning.

With this quick refresher on how GLU works, we can now start optimizing the llama-3.2-1B model.

NOTE

The code to follow this section is available in the notebook CH05_NB01_width_pruning.ipynb from the book's repository (https://github.com/peremartra/Rearchitecting-LLMs/blob/main/CH05/CH05_NB01_width_pruning.ipynb). For usability reasons, we won't reproduce all the code in the chapter, but we will see all the necessary steps to understand and perform the pruning process.

5.1.1 Identifying neurons for removal

Now that we understand the logic behind the magnitude metric, let's implement it in code. The first thing we need is a function that calculates the importance scores for each neuron pair:

```
def compute_neuron_pair_importance(gate_weight, up_weight):

    gate_max_abs = torch.max(gate_weight, dim=1).values + ➥
torch.abs(torch.min(gate_weight, dim=1).values)
    up_max_abs = torch.max(up_weight, dim=1).values + ➥
torch.abs(torch.min(up_weight, dim=1).values)

    importance_scores = gate_max_abs + up_max_abs

    return importance_scores
```

This function receives the weight matrices from both projections: `gate_weight` and `up_weight`. Each has the shape [`intermediate_size`, `hidden_size`] — for Llama-3.2-1B, [8192, 2048]. That is, 8192 rows (neurons), each with 2048 weights corresponding to the 2048 input dimensions (`hidden_size`) from the previous block.

In PyTorch, linear layers store weights with shape (`out_features`, `in_features`). Since both `gate_proj` and `up_proj` are defined as `Linear(in_features=2048, out_features=8192, bias=False)`, their weight matrices have shape [8192, 2048]:

- **Rows (8192):** Each row represents one of the neurons in the intermediate layer (the expansion we want to prune).
- **Columns (2048):** Represent the input weights that neuron receives from the hidden state.

For each of the 8192 rows (neurons), the code identifies the highest positive weight and the lowest (most negative) weight, converts the negative weight to its absolute value, and sums both to obtain the neuron's importance score.

An example:

- Neuron X: weights between [-0.2, 0.1] → range = 0.1 + 0.2 = 0.3

- Neuron Y: weights between [-0.5, 0.4] → range = 0.4 + 0.5 = 0.9
- Neuron Z: weights between [0.0, 0.4] → range = 0.4 + 0.0 = 0.4

Neuron X has the smallest range, or lowest importance score, which means it would be the candidate to be removed.

The function returns a vector of 8192 importance scores. This vector provides the necessary information to identify the most expendable neurons in the model.

5.1.2 Reconstructing the MLP module

Once we have the importance scores, the next step is to modify the model with the new intermediate size. The `prune_neuron_pairs` function encapsulates the entire reduction process: it calculates the new size, selects the indices to keep, and transfers the weights to new smaller Linear layers, which we'll end up adding to the model.

Listing 5.1 Create new MLP layers

```
def prune_neuron_pairs(mlp, prune_percent):

    gate_weight = mlp.gate_proj.weight.data.float() #1
    up_weight = mlp.up_proj.weight.data.float()

    importance_scores = compute_neuron_pair_importance(gate_weight,
                        up_weight) #2
    original_intermediate_size = gate_weight.size(0)

    num_neuron_pairs_to_prune = min(int(prune_percent *
➥original_intermediate_size),original_intermediate_size - 1) #3

    k = original_intermediate_size - num_neuron_pairs_to_prune #4

    _, indices_to_keep = torch.topk(importance_scores,
                k,
                largest=True,
                sorted=True) #5

    indices_to_keep = indices_to_keep.sort().values #6

    new_gate_proj = nn.Linear(
                mlp.gate_proj.in_features,
                k,
                bias=False).to(device) #7
    new_up_proj = nn.Linear(
                mlp.up_proj.in_features,
                k,
                bias=False).to(device)
    new_down_proj = nn.Linear(
                k,
                mlp.down_proj.out_features,
                bias=False).to(device)

    new_gate_proj.weight.data =
➥mlp.gate_proj.weight.data[indices_to_keep, :] #8
    new_up_proj.weight.data =
➥mlp.up_proj.weight.data[indices_to_keep, :]
    new_down_proj.weight.data =
➥mlp.down_proj.weight.data[:, indices_to_keep] #9
```

```
    return new_gate_proj, new_up_proj, new_down_proj, k
```

The function begins by extracting the weight matrices #1 from both `gate_proj` and `up_proj` layers, each with shape [8192, 2048], and converting them to `float32`. This conversion minimizes potential rounding issues, which is particularly important because models are typically loaded with lower precision formats such as `float16` or `bfloat16`. However, this may increase memory usage, which could be problematic for larger models. In this case, we've prioritized precision over memory consumption optimization.

With the weights extracted, the function calculates an importance score for each of the 8192 neuron pairs using the `compute_neuron_pair_importance` function #2. This returns a vector where higher scores indicate neurons that contribute more to the model's transformations. Next, the code calculates how many neurons to remove, ensuring we never prune all neurons by using `min()` to cap the removal #3. The new intermediate size `k` is then determined by subtracting the number of pruned neurons from the original size #4. For example, with 40% pruning, k = 8192 - 3277 = 4915, reducing the expansion ratio from 4x to approximately 2.4x.

The function then uses `torch.topk` to identify the k neurons with the highest importance scores #5. This operation is crucial for selecting which neuron pairs to retain. However, `torch.topk` returns indices sorted by value (from highest to lowest importance), which would scramble the arrangement of weights in the layer. Therefore, reordering after the `torch.topk()` call is essential. By applying `.sort()` `#6`, the neurons are recovered according to their original sequence, which keeps the internal structure of the weights as similar as possible to the original—important during the Knowledge Distillation process that we'll explore in the next chapter.

With the indices of neurons to keep identified and properly sorted, the function creates new Linear layers #7 with reduced dimensions: `gate_proj` and `up_proj` expand from 2048→k, while `down_proj` contracts from k→2048. This pattern preserves the connections with other modules in the Transformer block. Finally, the function copies the weights: #8 for `gate_proj` and `up_proj`, it selects only the rows (neurons) corresponding to `indices_to_keep`, keeping all input connections intact. For `down_proj`, it selects only the columns that correspond to `indices_to_keep #9`, because `down_proj` receives as input what `gate_proj` and `up_proj` produce as output.

The result is an MLP module that has reduced its expansion by the indicated percentage, retaining only the neurons that show the greatest transformation potential. In the next section, we will explore how to scale this operation across all Transformer blocks in the model.

Listing 5.2 Orchestrating pruning across all blocks

```
def update_model(model, prune_percent):

    new_intermediate_size = None
    for idx, layer in enumerate(model.model.layers): #A

        mlp = layer.mlp #1

        new_gate_proj, new_up_proj, new_down_proj, new_size = prune_
neuron_pairs(mlp, prune_percent) #2

        mlp.gate_proj = new_gate_proj #3
        mlp.up_proj = new_up_proj #3
        mlp.down_proj = new_down_proj #3
        new_intermediate_size = new_size

    model.config.intermediate_size = new_intermediate_size #4

    return model
```

#A Loop through each model layer.

The function is simple, as the complexity is encapsulated in `prune_neuron_pairs` from listing 5.1. For each layer, we access the MLP component #1. Since each layer is a `LlamaDecoderLayer`, it contains multiple components like Attention, MLP, and Layer norms; we target the MLP component specifically by accessing `layer.mlp`.

We then call `prune_neuron_pairs` with the MLP module and the pruning percentage #2, receiving the pruned layers and the new intermediate size. Next, we replace the original layers with the pruned versions #3: the new `gate_proj`, `up_proj`, and `down_proj` layers are assigned back to the MLP module, effectively updating the model's architecture in place.

The final step, updating `model.config.intermediate_size` #4, is critical. This update ensures the model's configuration reflects the architectural change, maintaining consistency for

any downstream code, like the Transformers library, that inspects model structure. Without this line, any code that inspected the model's architecture would still see the old dimensions, which could cause errors when trying to load the model.

Now we'll apply this function to our Llama-3.2-1B model with 40% pruning. This reduction will bring the expansion from 4x to approximately 2.4x, which will translate into a decrease of more than 26% in the model's total parameter count.

Choosing the pruning percentage is typically a pragmatic decision based on your resource constraints, intentions, and acceptable performance trade-offs rather than a mathematical rule. Models with higher expansion ratios (Llama-3.2-1B) generally tolerate more aggressive pruning than compact models (Gemma3-270m), because they have more "redundancy" to spare. Common practice is to start conservatively (10-20%) and incrementally increase if performance remains acceptable. We use 40% on LLaMA-3.2-1B because its 4x expansion provides enough margin for significant reduction while keeping degradation manageable, making the effects clearly visible for learning purposes.

```
prune_percent = 0.4
model_pruned = update_model(copy.deepcopy(model), prune_percent)
```

Note that we use `copy.deepcopy(model)` to create a complete copy before modifying it. This way we keep the original model intact so we can compare results later.

We can verify the impact of this reduction by counting the parameters:

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters())

original_param_count = count_parameters(model)
pruned_param_count = count_parameters(model_pruned)
reduction_in_params = original_param_count - pruned_param_count
percentage_savings = (reduction_in_params / original_param_count) *
100

print(f"Pruned model parameters: {pruned_param_count}")
print(f"Reduction in parameters: {reduction_in_params}")
print(f"Percentage of weight savings: {percentage_savings:.2f}%")
```

Which returns:

```
Pruned model parameters: 913770496
Reduction in parameters: 322043904
Percentage of weight savings: 26.06%
```

We've eliminated approximately 322 million parameters from the model. To put this into structural context, each complete Transformer block in Llama-3.2-1B contains roughly 60 million parameters. This reduction is equivalent to removing the content of five complete blocks, but with a crucial difference: all 16 blocks remain intact, preserving the model's depth.

NOTE

This pruning technique is architecture-agnostic and works with all modern GLU variants (SwiGLU, GeGLU, ReGLU) because they share the same three-layer structure. The activation function choice (SiLU, GELU, ReLU) doesn't affect pruning. Successfully tested on LLaMA, Gemma, and Qwen families.

The benefits differ from depth pruning, as we retain the attention layers in their entirety. These layers are responsible for most of the model’s memory consumption, primarily due to the KV cache they maintain. In the next section, we’ll analyze the model’s performance and behavioral changes after this modification.

5.1.3 Measuring the impact: benchmarks and trade-off

When evaluating width pruning, you need a more comprehensive benchmark system because the model's capabilities don't degrade uniformly. We might find that we're eliminating some capabilities while maintaining others. This is because during the training process, neurons specialize in different areas.

But before taking a look at the benchmarks, let's test that the model remains functional.

PM

Paris is the capital of

Paris is the capital of France. It is also known as Paris, the city of the France, is located in the center of this country. Paris is a capital city, it is known also as the Paris city. This city is one of Europe's most famous cities.

The response cannot be considered high-quality and error-free, but the model maintains basic structural integrity and remains on-topic. It’s important to note that this test was

conducted after a 26% reduction in the model's weight and without any fine-tuning or recovery process.

Now that we know the model still works, let's quantify exactly where the damage occurred. We need to run a comprehensive benchmark suite across multiple dimensions: reasoning capabilities, general knowledge retention, and language fluency. To keep the focus on pruning rather than evaluation mechanics, we've encapsulated this workflow in the `model_evaluation` helper function, which handles the standardized `lm_eval` harness integration. The technical details of this evaluation system are covered in Appendix B.

NOTE

The notebook runs three benchmarks (`truthfulqa_mc2`, `lambada_openai`, and `lambada_standard`) using `BENCHMARK_LIMIT = None` by default, which reproduces the full results shown in this chapter. If you're on a free Colab T4 and want faster execution, set `BENCHMARK_LIMIT = 100`. The complete results across all pruning configurations and the full benchmark suite are available at https://github.com/peremartra/llama-glu-expansion-pruning. The evaluation logic is encapsulated in the `model_evaluation` helper function, part of `utils.py`, a support file containing various helper functions used throughout the book, that will be covered in different appendices.

The evaluation system uses the `lm_eval` library (https://github.com/EleutherAI/lm-evaluation-harness), a true standard in LLM evaluation. It lets you run a standardized set of benchmarks that the research and engineering community uses to compare models. The

benchmarks the library allows you to run cover different types of tasks.

Reasoning and knowledge benchmarks:

- **MMLU (Massive Multitask Language Understanding)**: Evaluates general knowledge and problem-solving ability across 57 academic tasks in multiple-choice format, ranging from mathematics to history. A high MMLU score indicates the model possesses broad encyclopedic knowledge and analytical capability across various domains
- **GSM8K (Grade School Math 8K)**: Measures mathematical reasoning ability through grade-school level arithmetic problems requiring multiple steps. It's one of the best indicators of the model's sequential logical reasoning capability (Chain-of-Thought).
- **ARC-C**: Science questions at grade-school level that require reasoning beyond simple memorization. It focuses on the most difficult questions in the set.
- **HellaSwag**: Evaluates common sense. The model is given the beginning of an everyday situation and must choose from different options the most logical ending. It checks if the model understands what would normally happen next in the real world.

Understanding and language coherence benchmarks:

- **WikiText Perplexity & Lambada Perplexity**: Measure the model's fundamental fluency. Perplexity indicates how "uncertain" the model is by the words that appear in a text. Lower perplexity values mean the model predicts language naturally and coherently. Lambada is especially demanding because it requires capturing long-range dependencies in text.

- **BoolQ**: Yes/no questions based on text passages, measuring basic reading comprehension.
- **PIQA (Physical Interaction QA)**: Evaluates reasoning about real-world physical interactions through practical common-sense questions (for example, which tools to use for a household task).

Instruction following and truthfulness benchmarks:

- **IFEval (Instruction Following Evaluation)**: Measures how precisely the model executes specific instructions (format, length, constraints) without deviating. It's critical for applications where fidelity to the request is more important than creativity.
- **MUSR (Multistep Soft Reasoning)**: Evaluates the model's ability to solve problems that require multiple steps of logical reasoning under complex constraints and long narratives.
- **TruthfulQA**: Measures the truthfulness of the model's responses to questions designed to elicit incorrect but common answers (misconceptions). It evaluates whether the model can avoid reproducing common myths, superstitions, or frequent errors that are prevalent in internet training data.
- **WinoGrande**: Evaluates common-sense reasoning by resolving pronominal ambiguities: determining who a pronoun refers to when the context is ambiguous.

The evaluation is simple to run. In the notebook, a list of benchmarks is defined and the `model_evaluation` function is called, which encapsulates all the `lm_eval` logic:

```
BENCHMARK_LIMIT = None

BENCHMARKS_PRUNED = [
    {"name": "truthfulqa_mc2", "num_fewshot": 0},
    {"name": "lambada_openai", "num_fewshot": 0},
    {"name": "lambada_standard", "num_fewshot": 0},
]

results_pruned = model_evaluation(model_pruned,
                                  tokenizer,
                                  BENCHMARKS_PRUNED,
                                  limit=BENCHMARK_LIMIT,
                                  batch_size=4)
```

The `BENCHMARK_LIMIT` parameter controls the number of evaluated samples. It defaults to `None` to reproduce the full results shown in this chapter, but can be set to `100` for faster execution.

Let's look at the result of the base model and the one you created with pruning across different benchmarks, for which we'll use three different bar charts. We'll start by examining perplexity metrics, which directly measure the model's fundamental language modeling capability. Figure 5.2 shows the comparison across WikiText and Lambada datasets.

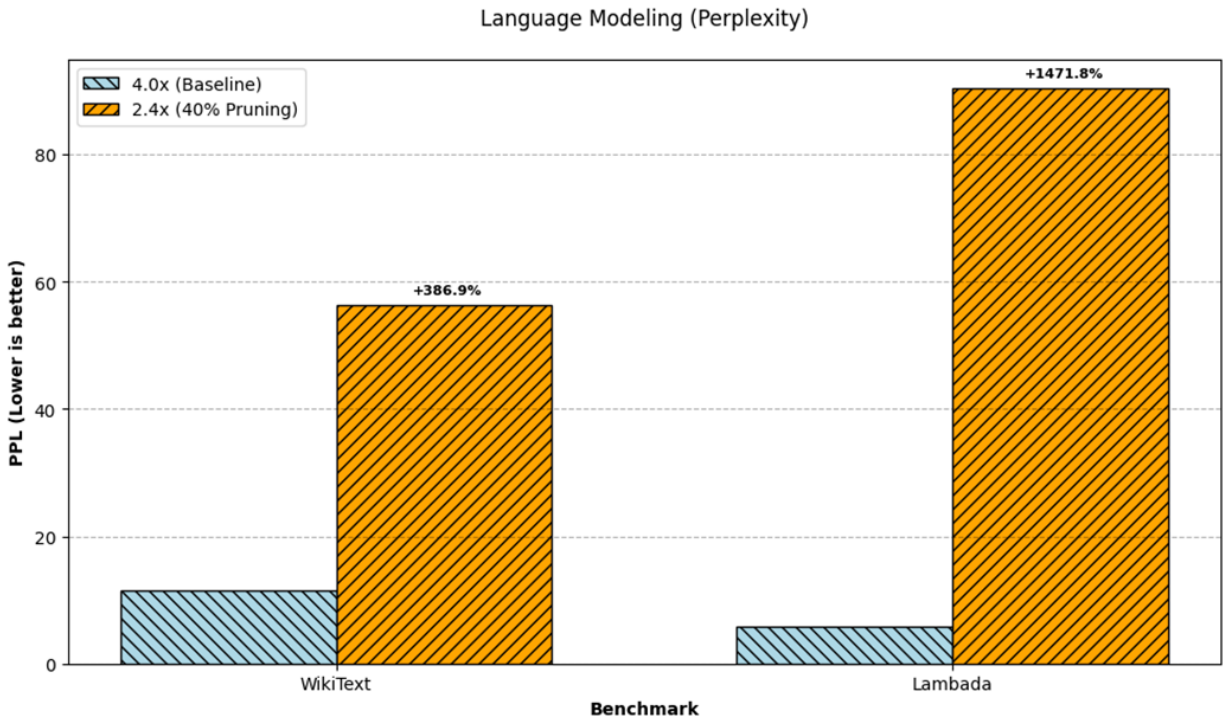


Figure 5.2 Impact of width pruning on perplexity (PPL). Since this is a metric where "lower is better," the bars on the right show severe degradation: removing 40% of the expansion neurons causes perplexity to spike, indicating that the model has lost much of its ability to predict text sequences fluently.

The WikiText and Lambada results confirm what you've seen in the empirical text generation test: the model generates less coherent text, without becoming a useless model, but far from what the base model achieved. The model is no longer a good generalist writer.

Beyond perplexity, we need to examine how pruning affects the model's higher-level cognitive abilities. Figure 5.3 shows the impact on reasoning and knowledge benchmarks.

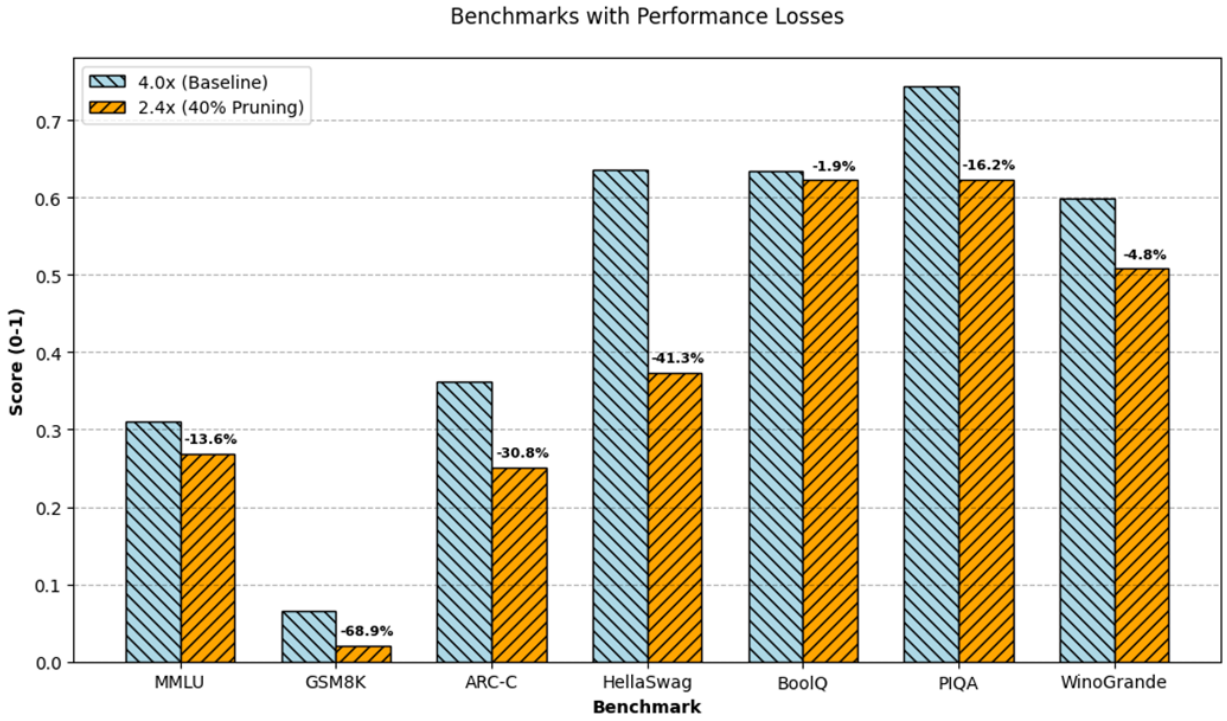


Figure 5.3 Degradation across reasoning and knowledge benchmarks. Mathematical reasoning (GSM8K -68.9%) and common sense understanding (HellaSwag -41.3%) suffer the most severe losses, while reading comprehension tasks (BoolQ -1.9%, WinoGrande -4.8%) remain relatively stable.

In this graph, you can start to observe an asymmetric impact on the model's capabilities. Tasks requiring reasoning (chain-of-thoughts) suffer more drastic drops. Linguistic comprehension tasks show more robustness.

- **GSM8K (-68.9%):** This is the most affected metric. This benchmark evaluates mathematical problem-solving through logical steps (chain-of-thought). The magnitude of the drop indicates that pruning has compromised the model's capability in multi-step reasoning processes.
- **HellaSwag (-41.3%), ARC-C (-30.8%), PiQA (-16.2%):** Just like in GSM8K, the critical factor here is reasoning: the model loses effectiveness when inferring plausible continuations (HellaSwag) or when deducing complex scientific answers (ARC-C).

- **MMLU (-13.6%):** Primarily evaluates declarative knowledge (facts), while other benchmarks like GSM8K demand procedural reasoning (logical steps). The smaller drop suggests that pruning affects reasoning pathways more than stored factual associations.
- **WinoGrande (-4.8%):** Focuses on resolving linguistic ambiguities (coreferences). Its stability indicates that local grammatical and semantic structures are much more resistant to pruning than prediction or causal reasoning mechanisms.
- **BoolQ (-1.9%):** Measures fundamental reading comprehension. Its resistance confirms that the basic "mechanics" of language processing remain intact; the model understands what it reads, even though it fails when performing complex reasoning about it.

Up to this point, the results are within expectations: by reducing size without a recovery process, the model loses fundamental reasoning and knowledge capabilities. However, a counterintuitive phenomenon emerges. While reasoning metrics collapse, another set of benchmarks (figure 5.4) behaves in the opposite way. This asymmetric impact reveals that different capabilities respond very differently to pruning, suggesting that width pruning can be more than a blunt optimization tool.

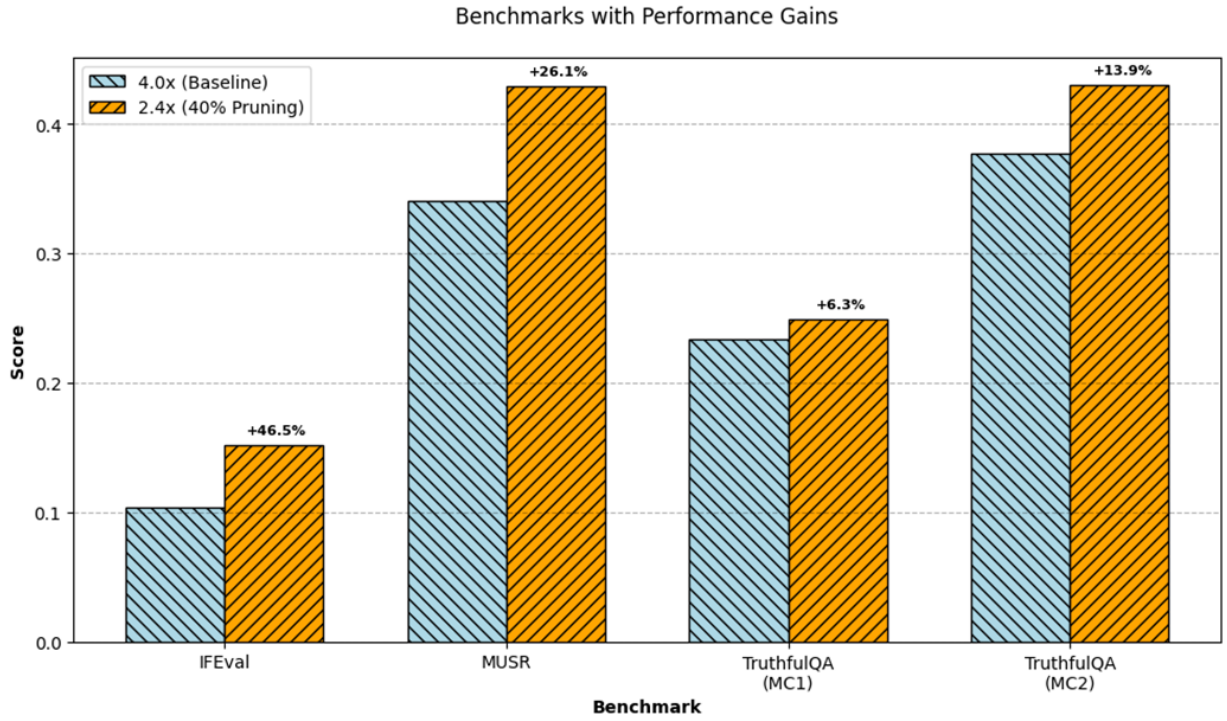


Figure 5.4 Unexpected performance gains in instruction-following and truthfulness benchmarks. The pruned model dramatically improves at following precise instructions (IFEval +46.5%) and multi-step reasoning under constraints (MUSR +26.1%), while also becoming more truthful and less prone to generating plausible but incorrect answers (TruthfulQA +13.9% on MC2).

The benchmarks that improve have a common characteristic: they all prioritize fidelity and precision over creativity or contextual elaboration.

- **IFEval (+46.5%)**: This improvement is critical for API integrations and agentic systems. IFEval measures the model's ability to execute instructions with specific constraints (for example, "respond in JSON format without preambles"). The original model, with its wide MLP modules, tended to "enrich" responses by adding unsolicited context.
- **MUSR (+26.1%)**: Being a reasoning benchmark, its improvement contrasts with GSM8K's collapse. The difference is that it requires extracting information from

long texts under strict rules and not mathematical reasoning, so it benefits from the increase in IFEval.

- **TruthfulQA (+13.9% in MC2)**: This benchmark penalizes plausible-sounding but incorrect answers. TThe pruned model scores measurably higher, though the reasons are likely multiple: reduced expressive capacity, lower output entropy, or a genuine reduction in the tendency to elaborate beyond what the model reliably knows. What the data shows clearly is that the pruned model is less likely to produce confident-sounding falsehoods, regardless of the underlying mechanism.

The general pattern is clear: we've obtained a model that has lost sophistication but has gained precision. It's less capable of reasoning in complex ways, but produces more constrained and focused responses. This makes it potentially well-suited for scenarios where precision matters more than creativity:

- **Structured data extraction (ETL):** When converting thousands of unstructured PDF invoices into JSON format, a "creative" model might try to correct a typo in an address or summarize a line item. A disciplined model will extract the data exactly as it appears, ensuring the downstream accounting system doesn't receive "hallucinated" corrections.
- **Agentic tool calling:** If an LLM is acting as an agent to trigger a bank transfer via API, it must provide the `account_id` and amount in a strict schema. A model that adds a "friendly preamble" or "explains its reasoning" in the middle of an API call will crash the system.
- **Personally identifiable information (PII):** In healthcare or legal contexts, you might instruct a model to "Replace all names with [REDACTED]." A creative model might accidentally leave a name because it felt it was "important for context." A disciplined model follows

the constraint to the letter, mitigating privacy breach risks.
- **Deterministic classification:** For high-volume content moderation, you need a model that outputs a single label (for example, SPAM or NOT_SPAM). Pruned models are less likely to "explain" their decision, which saves on output token costs and prevents the "drift" that occurs when a model starts debating its own labels.

This isn't simply a "worse" model. It's a model with a different capabilities profile.

To create this model you've only used weight magnitude, a completely data-free strategy that doesn't require knowing anything about the model's final use. This approach has a fundamental limitation: it treats all neurons equally, without considering which ones are actually important for your specific task. In the next section, we'll go a step further: we'll adapt the model to a concrete dataset using the input activations to the `down_proj` layer, capturing which neurons actually activate when the model processes the type of data we care about.

5.2 Data-driven neuron selection

In this section, you'll incorporate an additional level of sophistication to the neuron selection process. To do so, you'll redefine both `compute_neuron_pair_importance` and `prune_neuron_pairs` functions, with extended signatures that incorporate dynamic information captured during model execution. The reconstruction logic stays identical, selecting top-k neurons and creating new layers. What changes is how we decide which neurons are worth keeping.

You've already seen how the model's personality changes as you remove neurons; this change, although not intentionally

sought, emerged as a consequence of removing those neurons that were considered less structurally important.

We'll use a calibration dataset to decide which neurons to keep, similar to what you did in chapter 4 with layer selection, but this time at the level of individual neurons. In this case the approach is hybrid: the calculation of an `importance_score` is maintained, but it takes into account both weights and activations.

NOTE

The datasets are the same ones you used in chapter 4, so you won't find their preparation here. All the code is available in the notebook that accompanies the chapter: CH05_NB02_data_sms_wiki.ipynb (https://github.com/peremartra/Rearchitecting-LLMs/blob/main/CH05/CH05_NB02_data_sms_wiki.ipynb).

HYBRID IMPORTANCE THROUGH WEIGHTS AND ACTIVATIONS

The hybrid calculation combines two signals: the weight rank metric you used in section 5.1 (static component) and the activation norms captured from real data (dynamic component). Using activations alone would work well with large generic datasets, as laboratories like Mistral do, but with domain-specific calibration data the coverage is limited. Neurons that are structurally important but simply didn't fire during calibration would be incorrectly pruned, potentially destroying capabilities unrelated to the target domain. By multiplying both signals, we preserve neurons that have significant structural influence and are actually used by the

model on our data. This lets us specialize with small datasets and modest compute.

Because of how the GLU architecture works, in the previous solution we only took into account the weights of the `up_proj` and `gate_proj` layers, then we simply removed the corresponding neurons in the `down_proj` layer. This time we're going to also consider the weights of the neurons in `down_proj` and we'll also capture the activations that occur at its input.

Let's look at the circuit in figure 5.5.

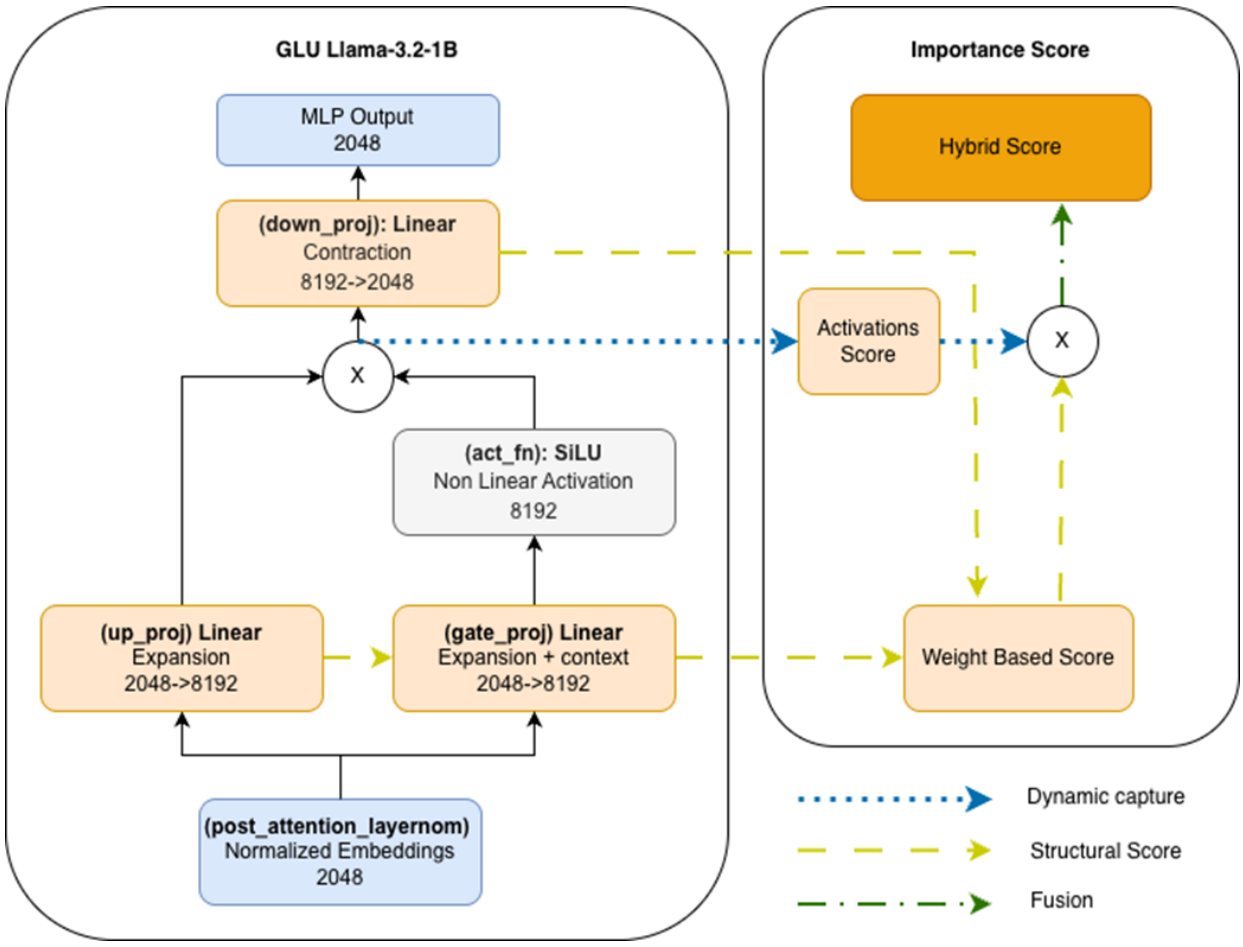


Figure 5.5 Hybrid importance calculation in GLU architecture. The static component combines weight ranges from gate_proj, up_proj, and down_proj. The dynamic component captures activation norms at the input of down_proj, where gated information flows. Both components multiply to produce the final importance score.

In figure 5.5, you can observe the three key components:

- **Dynamic capture**: The dotted line that comes right after the central multiplication (the circle in the left square with the "X"). We intercept the signal at the input of `down_proj`. We do this because `down_proj` mixes the signals from thousands of neurons and contracts the information; that is, in the case of the original Llama-3.2-1B, it goes from 8192 neurons to 2048. If we measured at its output, we'd lose traceability. By measuring at the input, we capture the exact contribution of each individual neuron after being filtered by the gate (`gate_proj`) and the activation function (`SiLU`).
- **Structural score (weight-based score)**: We calculate the magnitude range of the weights, but this time we include the three layers involved: `up_proj`, `gate_proj`, and `down_proj`. To maintain coherence with the previous section, we use the same range metric (Max + |Min|) that you already know, summing the normalized contributions of the three matrices.
- **Fusion (the multiplier)**: The key to the algorithm is the multiplication of the static and dynamic components. For a neuron to survive, it must have a solid structure (significant weights) and be actively used by the model (high activations). If either factor is low, the Hybrid Score will drop, marking the neuron as a candidate for elimination. For example: if a neuron has large weights but activation close to zero, its score will be low and it'll be a candidate for pruning.

Let's see the implementation in the following listing:

Listing 5.3 Implementation of the hybrid importance calculation

```
def compute_neuron_pair_importance(gate_weight, up_weight, down_weig
ht, X_d_norm):

    gate_weight = gate_weight.float() #A
    up_weight = up_weight.float() #A
    down_weight = down_weight.float() #A

    X_d_norm = X_d_norm.float().to(gate_weight.device)

    # gate_proj and up_proj: Neurons are rows (dim=1)
    gate_score = torch.max(gate_weight, dim=1).values +
➥torch.abs(torch.min(gate_weight, dim=1).values) #1
    up_score = torch.max(up_weight, dim=1).values +
➥torch.abs(torch.min(up_weight, dim=1).values) #1
    down_score = torch.max(down_weight, dim=0).values +
 ➥torch.abs(torch.min(down_weight, dim=0).values) #1

    gate_norm = gate_score / (gate_score.max() + 1e-8) #2
    up_norm = up_score / (up_score.max() + 1e-8) #2
    down_norm = down_score / (down_score.max() + 1e-8) #2

    structural_score = down_norm + gate_norm + up_norm #3

    importance_scores = structural_score * X_d_norm #4

    return importance_scores
```

#A Convert to float32 for numerical stability.

The function receives the weight matrices from the three GLU projections and the accumulated activation norms (`X_d_norm`) that you'll capture via PyTorch hooks. The result is a tensor with an importance score per neuron.

The function begins by calculating the static component using a range metric (Max + |Min|) for each projection #1. This captures the spread of weights for each neuron. Note the dimensional difference: for `gate_proj` and `up_proj`, neurons are rows (dim=1), while for `down_proj`, neurons are columns (dim=0). This reflects the architectural fact that `down_proj`

receives as input what the other two projections produce as output.

Next, the function normalizes each component to the [0, 1] range #2 so they contribute equally when summed. This normalization is a key difference from the code you've seen before. In modern LLMs, weight scales can differ by several orders of magnitude between expansion and contraction layers. Without this normalization to the [0, 1] range, a layer with numerically large weights would completely dominate the `structural_score`, making the contribution of the other two matrices irrelevant. The small `1e-8` added to each denominator is a standard numerical stability trick to prevent division by zero in the unlikely case that all weights in a layer are exactly zero.

The three normalized components are then summed #3 to produce the total structural capacity score, which represents the neuron's importance based purely on its weight characteristics. Finally, this structural score is multiplied by the activation norms #4 to produce the final hybrid importance. This final multiplication is the key to the hybrid approach: a neuron will only receive a high score if it combines significant structural capacity (large weight ranges) with real activity using the calibration data.

This means two neurons with identical weights, which would have had the same `importance_score` with the static approach, can now have a totally different score. Figure 5.6 illustrates this with concrete examples.

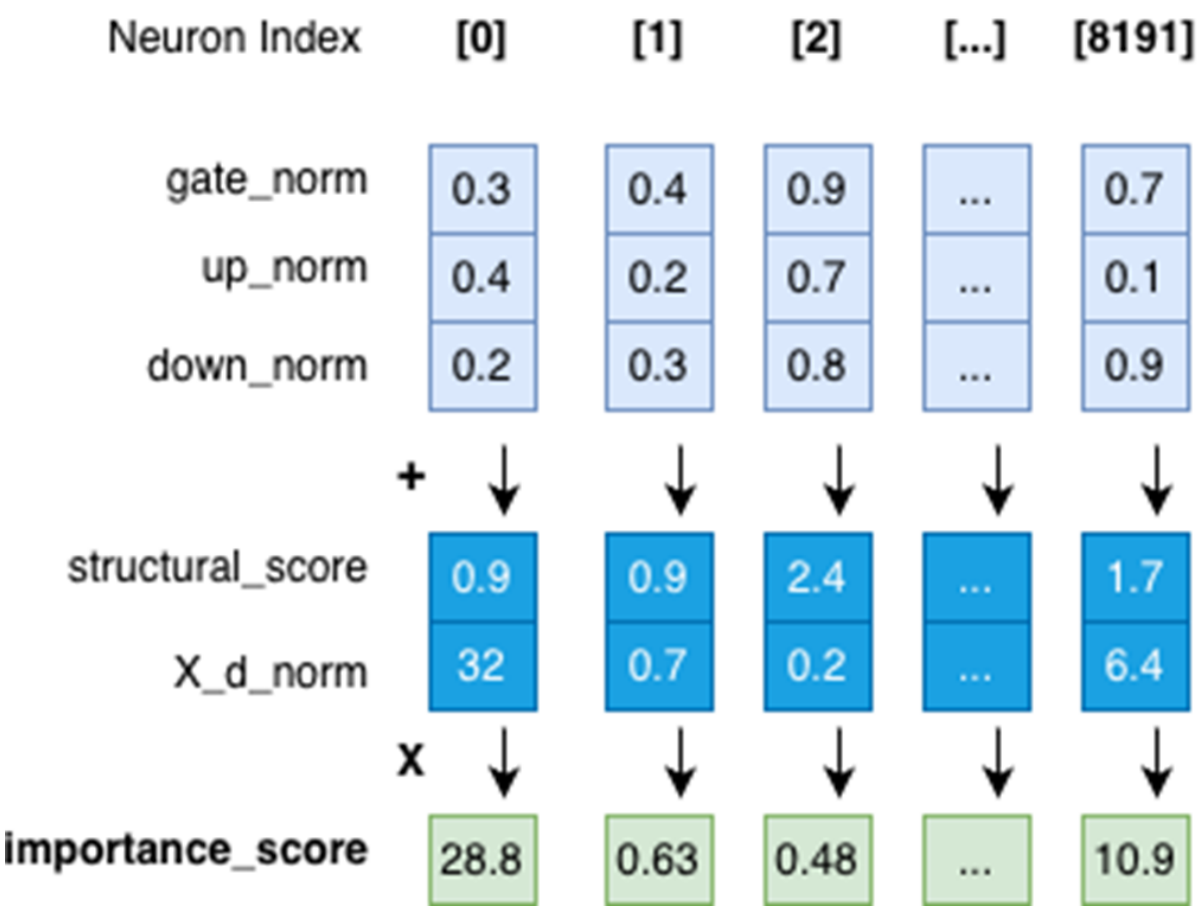


Figure 5.6 Hybrid importance combines structural capacity with activation patterns. Neuron [0] survives with moderate weights (0.9) but high activation (32). Neurons [1] and [2] are removed despite having identical or higher structural scores due to low activations (0.7 and 0.2).

- **Neuron [0]:** Moderate weights, but high activations on your dataset → High hybrid score → is kept.
- **Neuron [1]:** Identical weights, but activations close to zero → Low hybrid score → Removed
- **Neuron [2]:** High weights, with activations close to zero → Low hybrid score → Removed

The next step is to capture those `X_d_norm` activations during the processing of your calibration dataset. To do this, you'll use PyTorch hooks that will intercept the activations at the

input of `down_proj`, exactly the point marked by the dotted “Dynamic capture” arrow in Figure 5.5.

5.2.1 Extracting the dynamic importance component

As you saw, the `compute_neuron_pair_importance` function from listing 5.3 receives the `X_d_norm` parameter, which contains the dynamic component of the hybrid score. In this section we'll see how and when it's calculated.

The fundamental technique is the same as using PyTorch hooks to capture activations and decide which Transformer blocks were most important for different datasets. We’re registering automatic callbacks that intercept the data flow during the forward pass, but at a much finer level of granularity.

The capture, as you saw in figure 5.5, happens right before the input to `down_proj`; that is, after the gating mechanism. At this point, the signal has already passed through the "gate" (`gate_proj`) and the activation function (SiLU). If the gating mechanism decided to silence a specific neuron, you'll see it here.

If we measured at the output of `down_proj`, the information would have already been compressed back to the `hidden_size` (from 8192 to 2048 in Llama-3.2-1B), making it impossible to trace which intermediate neuron is responsible for which signal.

One of the main challenges is memory consumption. We can't store all captured activations in GPU memory because we would overflow it. Accumulating activations on CPU rather than GPU is a best practice regardless of your hardware capacity. This process happens offline during

model preparation, so the slight time overhead is negligible, but the memory savings translate directly into reduced cloud costs in production environments. The solution is to perform the calculation in the same hook, without returning the activations, and pass the result to the CPU to accumulate it. Figure 5.7 shows this process.

Let's look at the code:

Listing 5.4 Configuring hooks to capture MLP activation norms

```
_accumulated_act_norms = {} #1

def setup_mlp_hooks_for_importance(model, device):
    global _accumulated_act_norms
    _accumulated_act_norms.clear() #2
    gc.collect() #2
    torch.cuda.empty_cache() #2
    handles = []

    for idx, layer in enumerate(model.model.layers):
        intermediate_size = layer.mlp.down_proj.in_features
        _accumulated_act_norms[idx] = torch.zeros(
            intermediate_size,
            dtype=torch.float32,
            device='cpu' #3
        )

    def make_hook(layer_idx): #4
        def hook(module, input, output): #5

            X_d = input[0].detach() #6

            act_norms_L2 = torch.norm(
                X_d.to(torch.float32),
                p=2,
                dim=(0, 1)  #7
            )
            _accumulated_act_norms[layer_idx] +=
 ➥act_norms_L2.cpu() #8

       return hook

    for idx, layer in enumerate(model.model.layers):
        handle = layer.mlp.down_proj.register_forward_hook(
            make_hook(idx)
        ) #9
        handles.append(handle)

    print(f"✓ Registered {len(handles)} hooks on down_proj")

    return handles
```

```
def get_activation_norms():
    return {
        layer_idx: norms.clone()
        for layer_idx, norms in _accumulated_act_norms.items()
    }
```

The function accomplishes three main objectives: initialize storage, create the hooks, and register them in each `down_proj` layer of the model.

It begins by establishing a global dictionary #1 to store accumulated activation norms across all layers. At the start of each calibration run, this storage is cleared, and GPU memory is freed via garbage collection and cache clearing #2 to prevent out-of-memory errors during the calibration process.

For each layer, the function initializes CPU storage #3 with zeros matching the intermediate size. CPU placement prevents GPU memory overflow when processing large calibration datasets, as activation statistics are small enough to accumulate on CPU without performance penalties.

The hook creation uses a factory function pattern #4, because each layer needs to maintain its own `layer_idx`, and the factory function creates a closure that captures this value. The internal hook function #5 is what PyTorch executes automatically during the forward pass, intercepting X_d at the `down_proj` input.

Inside the hook, the function extracts the input tensor #6 from the tuple and detaches it from the computational graph to prevent gradient accumulation and free GPU memory. One difference from the function in chapter 4 is that we don't return the raw activations so that another function can calculate the difference between them. Instead, we convert the activations to their L2 norm, or Euclidean norm #7. This

formula tells us the intensity of the activations and is formed by taking the square root of the sum of squares of all values (for us, a simple call to `torch.norm(...)`). The norm is calculated across batch (dim=0) and sequence (dim=1) dimensions, reducing the [B, S, I] tensor to [I] per-neuron importance scores.

These norms are then accumulated on CPU #8. The `+=` operator sums across batches, building aggregate importance scores for the entire calibration dataset. Finally, the function registers the forward hook #9 on the `down_proj` layer for each MLP in the model. The hook will capture X_d every time data flows through this layer during calibration.

WHY DO WE USE L2?

To understand why we use L2, let's look at what happens with other metrics using a neuron that outputs the values [+5, -5, +5, -5]:

- **Simple sum**: 5 + (-5) + 5 + (-5) = 0. The values cancel out. The neuron appears inactive when it's clearly working.
- **L1 norm (sum of absolute values)**: |5| + |-5| + |5| + |-5| = 20. Captures total activity without cancellations. But treats all values equally.
- **L2 norm**: sqrt(5² + (-5)² + 5² + (-5)²) = sqrt(100) = 10. Captures activity and, by squaring, penalizes larger values more.

A neuron with activations [10, 0, 0, 0] has L1 = 10 and L2 = 10, while [5, 5, 0, 0] has L1 = 10 but L2 ≈ 7.07. L2 rewards the first neuron, the one with an activation spike.

The L2 norm helps us preserve "specialist" neurons that have high peaks, while it penalizes neurons that have constant but low background noise.

The `get_activation_norms()` function retrieves the stored scores before calling `compute_neuron_pair_importance()`. It returns a clean copy using `.clone()`, creating independent tensors for hybrid importance calculations, ensuring the original tensor data remains unaltered.

With the capture system complete, you're ready to execute the end-to-end process. In the next section, you'll orchestrate the full workflow: you'll feed real data (WikiText vs. SMS) into the system and observe how the model

autonomously determines which neurons are redundant, based on the calibration dataset.

5.2.2 Reconstructing the MLP with hybrid scores

We already have two of the key functions: `compute_neuron_pair_importance` from listing 5.3 combined weights and activations, and the hook system created with `setup_mlp_hooks_for_importance` in listing 5.4. Now you're going to create a function that prunes the model by creating layers that keep the neurons with the highest `importance_score`.

To do this, we need to extend the `prune_neuron_pairs` function from listing 5.1. The general structure remains: calculate importance, select the k most important neurons, create new layers, and transfer the weights. The critical difference is that now the importance calculation requires an additional parameter: the activation norms `X_d_norm` that you've captured with the hook system. The intention is to change the selection criterion from "large neurons" to "large and active neurons."

Listing 5.5 Pruning neuron pairs with hybrid importance

```
def prune_neuron_pairs(mlp, prune_percent, X_d_norm, layer_idx):
    gate_weight = mlp.gate_proj.weight.data #1
    up_weight = mlp.up_proj.weight.data #1
    down_weight = mlp.down_proj.weight.data #1

    original_intermediate_size = gate_weight.size(0)

    importance_scores = compute_neuron_pair_importance(
        gate_weight=gate_weight,
        up_weight=up_weight,
        down_weight=down_weight,
        X_d_norm=X_d_norm #2
    )

    num_to_prune = min(
        int(prune_percent * original_intermediate_size),
        original_intermediate_size - 1 #3
    )
    k = original_intermediate_size - num_to_prune #D #4

    if k <= 0:
        raise ValueError(f"Invalid k={k} for layer {layer_idx}")

    _, indices_to_keep = torch.topk(
        importance_scores,
        k,
        largest=True,
        sorted=True #5
    )

    indices_to_keep = indices_to_keep.sort().values #6

    new_gate_proj = nn.Linear(
        mlp.gate_proj.in_features,
        k,
        bias=False
    ).to(device) #7

    new_up_proj = nn.Linear(
        mlp.up_proj.in_features,
        k,
        bias=False
```

```
    ).to(device) #7

    new_down_proj = nn.Linear(
        k,
        mlp.down_proj.out_features,
        bias=False
    ).to(device) #8

    new_gate_proj.weight.data = gate_weight[indices_to_keep, :] #9
    new_up_proj.weight.data = up_weight[indices_to_keep, :]
    new_down_proj.weight.data = down_weight[:, indices_to_keep] #10

    return new_gate_proj, new_up_proj, new_down_proj, k
```

This function is nearly identical to the static version from section 5.1.2, with one critical difference: it now receives and propagates `X_d_norm` to `compute_neuron_pair_importance`.

The function begins by extracting weight matrices #1 from the three MLP projection layers. Unlike the static version, we don't convert to float32 here because that conversion is already encapsulated in `compute_neuron_pair_importance` (listing 5.3). These weights are then passed to the importance calculation function along with `X_d_norm` #2, which contains the accumulated L2 norms captured by hooks during calibration, representing the dynamic component of importance. The scores that function returns already incorporate both structural information (weights) and the model's actual behavior during calibration (activations).

The rest of the process stays the same: the function includes a safety check #3 to ensure we never prune all neurons, then calculates the new intermediate size `k` #4 by subtracting the number of neurons to remove from the original size. It selects the k neurons with the highest hybrid importance scores #5 using `torch.topk`, then sorts the indices #6 in ascending order to maintain the original neuron

arrangement in the weight matrices, keeping the structure organized.

The function creates new layers with reduced dimensions: first the expansion layers #7 (`gate_proj` and `up_proj`) where the input dimension remains at `hidden_size` to maintain compatibility with the attention module, but the output dimension shrinks to k neurons. Then the contraction layer #8 (`down_proj`) where the input dimension shrinks to k neurons, but the output dimension remains at `hidden_size` to maintain compatibility with the next block's attention module. Finally, it copies weights for the kept neurons: for `gate_proj` and `up_proj` #9, where neurons are rows in these projections, we index the first dimension. For `down_proj` #10, because neurons are columns here, we index the second dimension.

The result is a model that has been physically rearchitected to prioritize the transformations that actually matter for your calibration domain. If you calibrate with WikiText, you'll preserve neurons specialized in processing formal encyclopedic text. If you calibrate with SMS, you'll preserve neurons optimized for colloquial language and short messages.

Similarly, you could calibrate with the dataset from a reasoning benchmark like ARC-Challenge to specifically preserve the neurons that activate during scientific reasoning tasks. Unlike the static pruning from section 5.1, where you saw accidental improvements in certain benchmarks, the data-driven approach allows you to intentionally design which model capabilities you want to protect during pruning.

With this adapted function, you're ready to run the end-to-end process: capture activations through the calibration loop

(which we'll see in the next section) and apply pruning to create domain-specialized models (in section 5.2.5).

5.2.3 Running the calibration loop

In this section, you'll load your calibration data in batches, move it to the GPU, and run a standard forward pass. During each forward pass, the hooks fire automatically, intercept `x_d`, calculate its L2 norm, and accumulate the result in the CPU. The pipeline is shown in figure 5.7. All this happens transparently, without you having to do anything more than call the model.

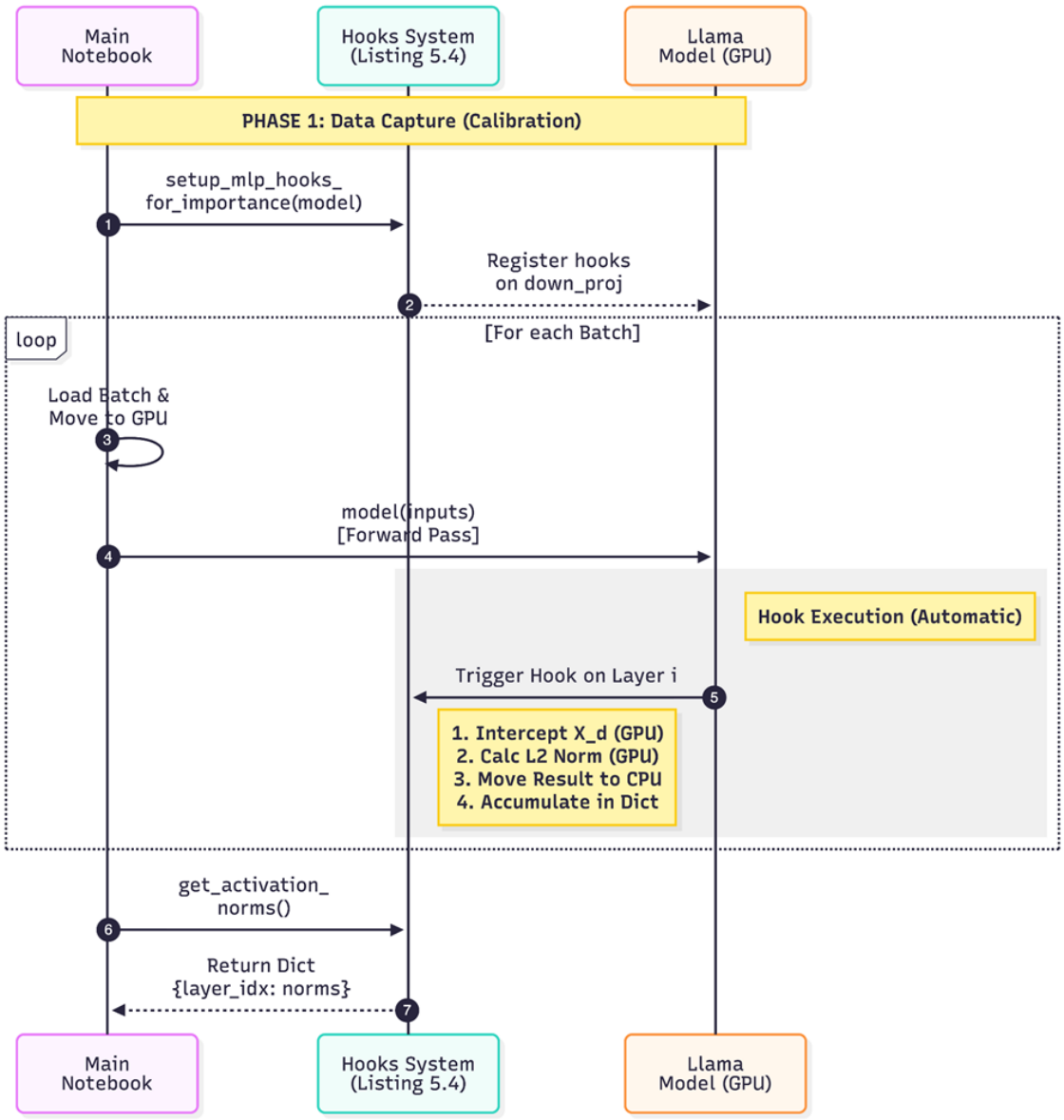


Figure 5.7 Activation capture pipeline. The hooks registered in down_proj (1-2) are automatically triggered during the forward pass (3-4), intercept X_d, calculate L2 norms and accumulate them in CPU (5), finally returning an importance dictionary per layer (6-7).

The dataloaders you'll use here are the same ones you prepared in chapter 4: WikiText-2 for formal encyclopedic text and SMS Spam for informal conversational messages.

The following listing shows the calibration loop for the WikiText dataset.

Listing 5.6 Calibration loop

```
handles_wiki = setup_mlp_hooks_for_importance(model, device) #1
model.eval() #2
with torch.no_grad(): #2
    for batch_idx, batch in enumerate(➥
tqdm(dataloaderwiki, desc="Wiki Calibration")): #3
        inputs = {
            'input_ids': batch['input_ids'].to(device),
            'attention_mask': batch['attention_mask'].to(device)
        } #4
        outputs = model(**inputs) #5

        if (batch_idx + 1) % 10 == 0:
            torch.cuda.empty_cache() #6

for handle in handles_wiki:
    handle.remove() #7

activation_norms_wiki = get_activation_norms() #8
```

The loop orchestrates the functions you've created previously, responsible for creating the hooks, calculating the L2 norms, and saving them.

It begins by registering hooks #1 on all MLP `down_proj` layers. From this moment, every forward pass will trigger automatic activation capture. The model is then set to evaluation mode and gradient computation is disabled #2. Evaluation mode disables dropout and ensures deterministic behavior during calibration, while disabling gradients is essential because we're not training, just observing activations, so gradients would waste memory and time.

The loop then iterates through calibration batches #3, processing each one sequentially. For each batch, the input tensors are moved to GPU #4. This is necessary because the model and hooks are already on GPU, so inputs must match device placement. The forward pass #5 is where the magic

happens: while the model computes outputs, the registered hooks trigger automatically, capturing X_d (activations) at every `down_proj` input, calculating L2 norms, and accumulating them on CPU.

Periodically, the cache is cleared #6 to prevent gradual VRAM accumulation from intermediate tensors that PyTorch doesn't immediately release. This happens every 10 steps because memory saving is prioritized when working with limited GPUs like the T4, available in Google Colab's free tier. It's a defensive decision that can impact processing speed. On GPUs with more available memory, it can be removed or executed every 50 steps.

After processing all batches, the hooks are unregistered #7 to prevent them from firing during subsequent model operations. Finally, the call to `get_activation_norms()` #8 retrieves the accumulated norms dictionary. This contains one tensor per layer with importance scores for all neurons.

To obtain the norms for the model specialized in SMS, you just need to repeat this same block replacing the dataloader. At the end of this phase, you'll have two L2 norm dictionaries: `activation_norms_wiki` and `activation_norms_sms`, which you'll use to run the guided pruning process and obtain two different models.

5.2.4 Creating domain-specialized models

You now have two dictionaries, `activation_norms_wiki` and `activation_norms_sms`, which were extracted from the same base model using the same code but with different datasets. These dictionaries tell us which neurons activate most intensely for each dataset.

When you prune the model with `activation_norms_wiki`, you'll eliminate neurons that were barely activated with encyclopedic text. When you prune it with `activation_norms_sms`, you'll eliminate different neurons: those with low activation patterns during SMS processing.

Both models will have the same parameter reduction (the same pruning percentage), the same external shape (Llama-3.2-1B with narrower MLP), but different neural connections under the hood.

The first step is to adapt the `update_model` function you used in static pruning so it accepts and propagates the activation norms dictionary.

Listing 5.7 Update model for data-driven pruning

```
def update_model(model, prune_percent, activation_norms):
    new_intermediate_size = None
    pruning_stats = []

    for idx, layer in enumerate(model.model.layers): #A
        mlp = layer.mlp

        if idx not in activation_norms: #1
            raise KeyError(f"No activation norms for layer {idx}")

        X_d_norm = activation_norms[idx] #2
        original_size = mlp.gate_proj.out_features

        new_gate_proj, new_up_proj, new_down_proj, new_size = prune_
neuron_pairs(
            mlp=mlp,
            prune_percent=prune_percent,
            X_d_norm=X_d_norm, #3
            layer_idx=idx
        )

        mlp.gate_proj = new_gate_proj
        mlp.up_proj = new_up_proj
        mlp.down_proj = new_down_proj

        if new_intermediate_size is None:
            new_intermediate_size = new_size

    model.config.intermediate_size = new_intermediate_size #4

    return model
```

#A We iterate over all the model's layers.

The code in Listing 5.7 orchestrates multiple components. To avoid getting lost in the syntax, Figure 5.8 illustrates the system's logical architecture: how we intercept an internal signal and convert it into an importance metric, which ends up defining the new composition of the model's layers.

For each layer, the function first performs a safety check #1 to ensure we have activation data for every layer. Missing data would cause silent failures downstream, so this validation is essential. It then extracts the activation norm tensor #2 for this specific layer. This tensor contains importance scores for all 8192 neurons, accumulated during the calibration loop from listing 5.6.

These activation norms are passed to `prune_neuron_pairs` #3, where they will be multiplied with the structural score in `compute_neuron_pair_importance` (listing 5.3), combining the dynamic component (actual neuron activity) with the static component (weight magnitudes) to produce the final hybrid importance scores. After the pruning operations complete and the new layers are assigned back to the MLP module, the function modifies the model configuration file #4 to avoid compatibility issues with libraries, like Transformers, that use it.

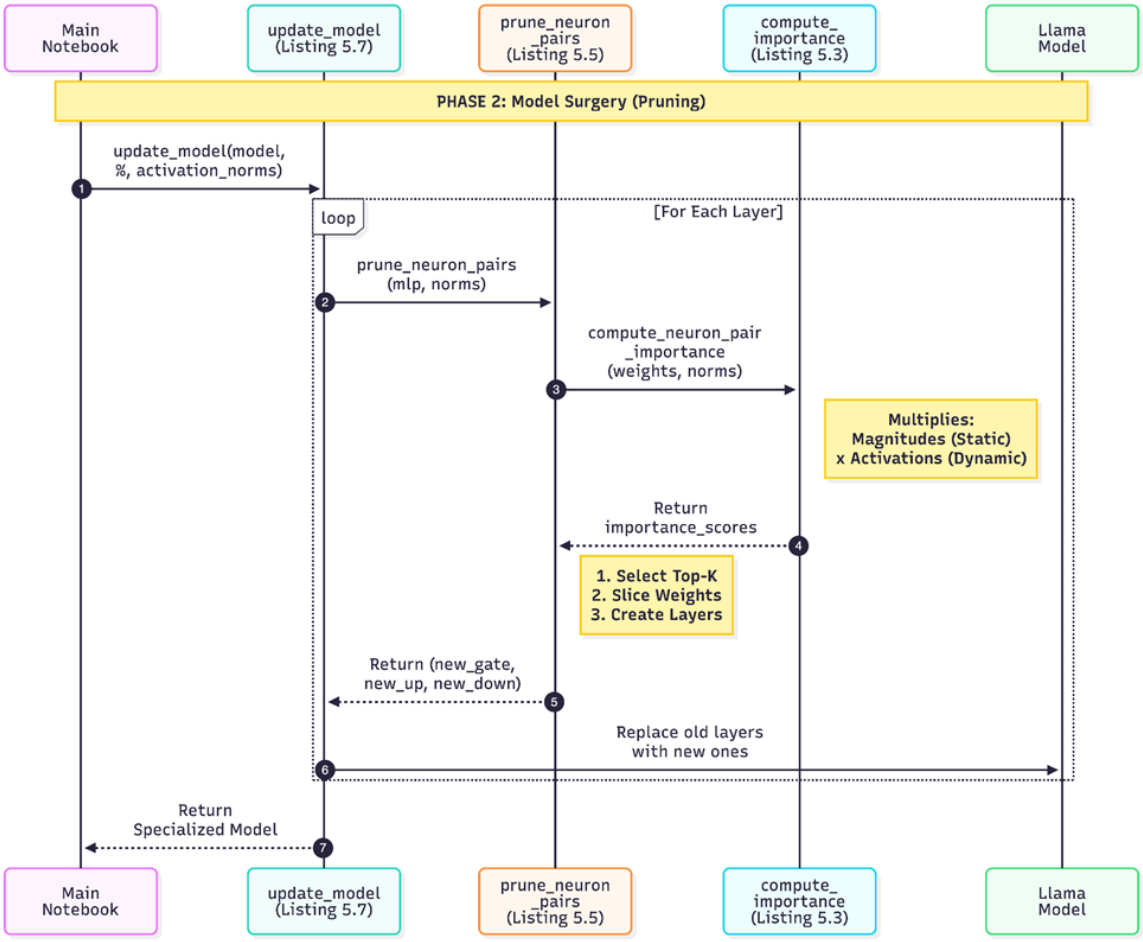


Figure 5.8 Model surgery using hybrid importance. The update_model function (Listing 5.7) iterates over each layer (1-2), calls prune_neuron_pairs (3) that calculates hybrid scores by multiplying structural magnitudes by activation norms (4), selects the top-K neurons and creates new layers (5), replaces the original layers (6), and returns the specialized model (7).

The function is now ready to create the models. We work with 20% pruning instead of the 40% used in section 5.1. A more conservative reduction lets us isolate the effect of the calibration data on neuron selection, without the added noise of a heavily degraded model.

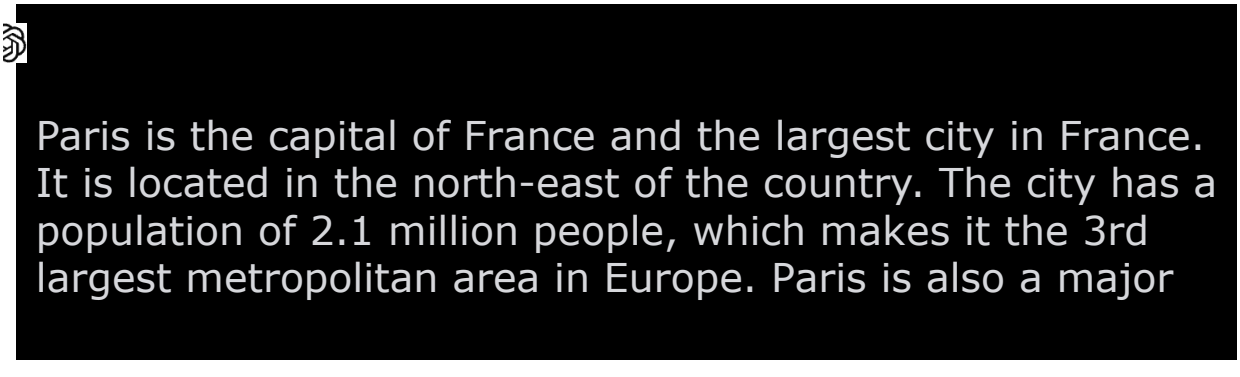
```
PRUNE_PERCENT=20
wiki_model = update_model(copy.deepcopy(model),
                          PRUNE_PERCENT,
                          activation_norms_wiki)

sms_model = update_model(copy.deepcopy(model),
                          PRUNE_PERCENT,
                          activation_norms_sms)
```

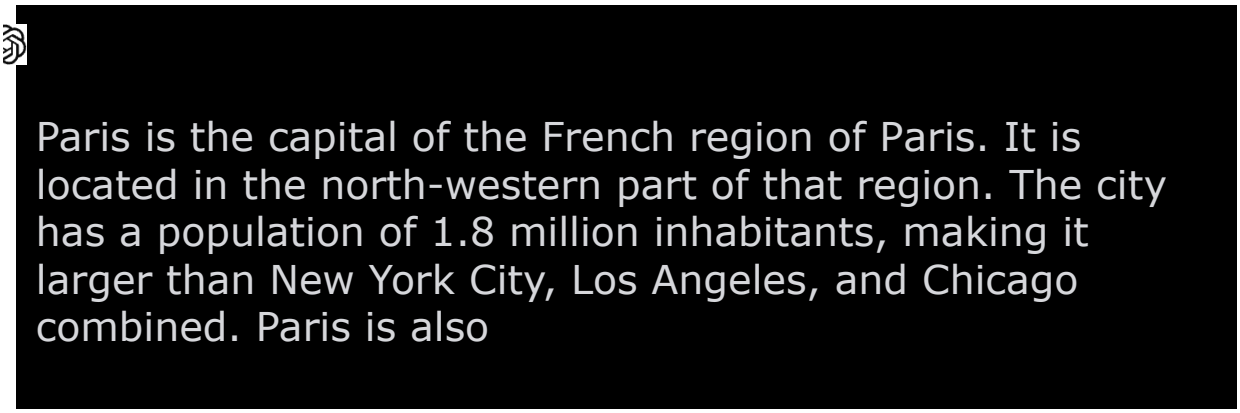
Each MLP module will go from an expansion of 8192 neurons to 6554, eliminating the 1638 neurons that contributed the least, based on their structural magnitude and activations.

The two models share the same architecture:

```
Original model:     1,235,814,400 parameters
wiki_model:  1,074,792,448 parameters (13.03% reduction)
sms_model:   1,074,792,448 parameters (13.03% reduction)
```

But the internal weights of the MLP module are different, since the neurons that have been preserved are not the same. Let's see how both models complete the prompt: "Paris is the capital of”.





Both models continue generating coherent text, but with different styles. The model prepared with the SMS spam dataset introduces more factual inaccuracies, incorrect comparisons, and a more direct text structure.

`Wiki_model` has preserved the neurons that react to complex grammatical structures and extensive vocabulary, while `sms_model` has preserved the neurons that activate with colloquial language, abbreviations, and short phrases. We forced the model to physically specialize in the domain of its calibration data.

In the next section, you'll evaluate the three models (`base`, `wiki_model`, `sms_model`) on both datasets (WikiText and SMS).

5.2.5 Measuring the specialization effect

You've created two variants of the same base model (Llama-3.2-1B), both removing 20% of their expansion neurons. The difference between them is the selection criterion for which neurons remain.

- `wiki_model` preserves neurons that are active when facing formal and encyclopedic language.
- `sms_model` preserves those that react to short, colloquial messages.

Both models retain the most structurally important neurons, which are responsible for the model maintaining its basic capabilities.

Now, just like in Chapter 4, you're going to use the perplexity metric with the `evaluate_metrics` function from Listing 4.8, which measures how "surprised" the model is by the words that appear. A model specialized in SMS should have lower perplexity on SMS than on WikiText, and vice versa.

Let's run the evaluations:

```
metrics_base_wiki = evaluate_metrics(model, dataloaderwiki)
metrics_base_sms = evaluate_metrics(model, dataloadersms)

metrics_wiki_wiki = evaluate_metrics(wiki_model, dataloaderwiki)
metrics_wiki_sms = evaluate_metrics(wiki_model, dataloadersms)

metrics_sms_wiki = evaluate_metrics(sms_model, dataloaderwiki)
metrics_sms_sms = evaluate_metrics(sms_model, dataloadersms)
```

Let's look at the results in table 5.1. Additionally, I've included in the table a static-pruned model pruned to 20% but using only weight magnitudes, as a baseline to measure the real value of the data-driven approach. The results for this static model were obtained in the companion notebook CH05_NB03_bonus.ipynb, available in the book's repository.

Table 5.1 Perplexity results

Model	Parameters	Wiki Perplexity	SMS Perplexity
Base	1.24B	25.69	122.91
Wiki model	1.07B (~87%)	36.28	182.54
SMS model	1.07B (~87%)	48.65	165.54
Static 20% pruned	1.07B (~87%)	52	217

Each of the models works better with the dataset it was calibrated with, which shows that the calibration dataset directs the pruning toward preserving specific domain capabilities.

You can see that the model created with pruning guided exclusively by the static criterion performs the worst on any of the datasets. This is normal because, even though we used different datasets, both models created via the dynamic method have preserved neurons that participate in text generation. If we had calibrated for a very different task (for example, binary yes/no classification), we would have preserved different neurons and that model's perplexity could have been worse (higher) than the static one, even though its performance on the binary task would be better. The advantage of the data-driven method is that it acts as a guide to shape the model's behavior.

The on-domain specialization has worked correctly—each model is better on the dataset it was calibrated with, which shows that pruning guidance can be very fine-grained, because you can not only differentiate by neurons that

generate text, but they also differ depending on the text they need to generate or understand.

It's crucial to observe that, although specialization works, the pruned models show higher perplexity (worse) than the original base model. By removing 20% of the parameters, we're reducing the model's total capacity to represent knowledge.

However, the value of data-driven pruning isn't to surpass the original model on its own terms, but to minimize damage in the domain we care about. This way, the subsequent recovery process (like fine-tuning or distillation that we'll see later) will be faster, cheaper, and achieve better final numbers on your specific task. The complete pipeline isn't just pruning; it's pruning plus recovery.

5.2.6 Inference performance and energy efficiency

The capability benefits you just saw depend critically on the chosen calibration dataset. However, the improvements in computational performance and energy consumption depend only on the percentage of pruning applied, not on which specific neurons you removed.

NOTE

In this section we're only going to discuss the results of evaluating the models. The evaluation system is explained in different appendices. The inference performance system is in Appendix D and the energy efficiency one is in Appendix C.

As a measure to check the inference improvement, we used the number of tokens generated per second. The values we're going to see correspond to a T4 GPU in the Google Colab environment, with short text generation (maximum 50 tokens) in batches of 4.

Table 5.2 shows the text generation speed (tokens per second) when processing batches of prompts.

Table 5.2 Inference performance

Model	Tokens / sec (WikiText)	Tokens / sec (SMS)	Gains
Base	18.31	18.26	
Wiki model	21.90	21.93	+19.6% / +20.1%
SMS model	21.83	21.89	+19.2% / +19.9%

A consistent speed increase of almost 20% is observed. Note that the improvement is identical for both pruned models, regardless of the calibration dataset used. This makes sense: although the internal weights are different, the resulting physical architecture (the matrix dimensions) is the same in both cases.

Speed isn't the only cost metric. Energy consumption is a critical factor in large-scale deployments. Using the `codecarbon` library (https://github.com/mlco2/codecarbon) during generation, we've measured the energy consumed per generated token in Table 5.3.

Table 5.3 Energy efficiency comparison

Model	Energy	Joules / Token	Efficiency	Tokens Generated
Base	1908.42	1.322		1444
Wiki model	2544.24	1.236	6.5%	2059
SMS model	2473.31	1.239	6.3%	1997

Remember when using a shared environment like Google Colab, the data can vary between different sessions, depending on the assigned GPU and the shared workload it has. Additionally, libraries like CodeCarbon calculate emissions based on publicly available data from major cloud providers, which doesn't always account for the full carbon footprint of specific data centers. For this reason, energy efficiency calculations for SLMs are significantly more accurate when performed on a private computer or a dedicated cluster where you have direct access to hardware sensors. You should treat the numbers in this chapter as a comparative baseline rather than absolute metrics.

From the results in the table, we can observe a reduction in the number of joules needed to generate a token. However, the pruned models generate more tokens than the base model. This behavior can be corrected with fine-tuning, but the cost per token will continue to be the same, since it's due to the expansion reduction, while the higher token generation is due to the behavioral change the model has undergone from having its weights modified.

This reduction in cost per token would allow us to deploy different specialized variants from the same base model,

with results that can even be superior to the base model and with lower GPU consumption.

The improvements from reducing the expansion ratio usually reach a limit, because the real performance bottleneck of the model is typically found in the memory movements caused by the attention module.

By reducing the expansion, we're reducing the necessary compute and the memory occupied by the model, but if the GPU isn't at its limit, it will yield only marginal performance improvements.

If we're only looking for an improvement in inference performance, width pruning will be really useful when the bottleneck is the GPU's compute capacity. Width pruning offers several additional advantages, which we'll see throughout the book, like adapting the model to a domain or correcting unwanted behaviors without needing a fine-tuning process.

ALIGNMENT WITH THE GPU ARCHITECTURE

When the bottleneck is actually GPU compute capacity (production scenarios with high load), there's an additional factor that can make a difference: the resulting `intermediate_size` should be divisible by 32, 64, 128, or 256.

Modern GPUs (like NVIDIA's Ampere or Hopper architectures) process matrices by dividing them into small blocks ("tiles"). Each Tensor Core, a specialized hardware unit, performs matrix multiplications on these tiles very efficiently. For example, in a single clock cycle, a Tensor Core can process a 16x16 block of elements (for FP16 precision numbers).

To take full advantage of Tensor Cores, matrix dimensions (like width or height) need to be multiples of certain values, typically powers of 2 like 32, 64, 128, or 256. If we choose an intermediate size, like 6554, the GPU will have to add padding (fill with zeros).

The OptiPFair library that comes with this book includes an `expansion_divisor` parameter that automatically adjusts the number of neurons to keep so the resulting `intermediate_size` is divisible by the specified value.

```
pruned_model, stats = prune_model(
    model=model,
    pruning_type="MLP_GLU",
    neuron_selection_method="PPM",
    pruning_percentage=20,
    expansion_divisor=64,  #A
    dataloader=dataloaderwiki,
    show_progress=True
)
```

You may find that, when running this model on a T4 GPU (common in Colab), you do not observe a performance improvement compared to the non-aligned model. This is expected, as the T4 is typically constrained by memory bandwidth, causing the Tensor Cores to idle while waiting for data.

Dimension alignment is particularly effective in compute-bound scenarios, such as training or high-traffic inference on A100/H100 GPUs, where maximizing the computational efficiency of the Tensor Cores is critical.

5.3 From paper to practice

The work you've done in this chapter has been heavily inspired by two papers. The first, "Fragile Knowledge, Robust Instruction-Following: The Width Pruning Dichotomy in Llama-3.2" (Martra, 2025), demonstrates that width pruning doesn't degrade model capabilities uniformly. The second, "CFSP: An Efficient Structured Pruning Framework for LLMs with Coarse-to-Fine Activation Information" (Wang et al., 2024), proposes using model activations as a guide to identify which neurons are actually important for your specific data.

From the first paper we've taken the identification of structural neurons through static magnitude-based calculation using the `up_proj` and `gate_proj` layers of the GLU structure.

For each neuron in `gate_proj` and `up_proj`, the range of its weights is calculated: the maximum value plus the absolute value of the minimum. Then these values are summed to get an importance score. Why does this seemingly simple metric work? Because it captures the "structural range" of each

neuron: its potential capacity to transform information. A neuron with a large weight range can generate broader transformations in the activations, while one with a small range contributes less to information transformation.

To validate the robustness of the method, the paper performs an exhaustive evaluation by systematically testing different pruning percentages (10%, 20%, 30%, 40%, 50%, and 60%) on the Llama-3.2-1B and Llama-3.2-3B models. Let's take the results from the 1B model with 40% pruning (reducing the expansion from 4.0x to 2.4x) as a representative example:

Capabilities that collapse (Knowledge-Intensive):

- MMLU: 0.3111 → 0.2689 (-13.6%)
- GSM8K: 0.0660 → 0.0205 (-68.9%)
- ARC-Challenge: 0.3626 → 0.2509 (-30.8%)
- HellaSwag: 0.6363 → 0.3737 (-41.3%)

The most obvious damage is seen in perplexity metrics: WikiText increases by +387% (from 11.57 to 56.33) and Lambada by an impressive +1,472% (from 5.75 to 90.38). These numbers indicate that the model's fundamental ability to predict text coherently has been severely compromised. The collapse in GSM8K is particularly severe, with a 68.9% loss of the original capability.

Capabilities that improve (Instruction-Following):

- IFEval: 0.1035 → 0.1516 (+46.5%)
- MUSR: 0.3399 → 0.4286 (+26.1%)
- TruthfulQA-MC2: 0.3772 → 0.4298 (+13.9%)
- TruthfulQA-MC1: 0.2338 → 0.2485 (+6.3%)

This pattern is not exclusive to the 1B model. In the 3B model shown in figure 5.9, the improvement in IFEval is even more pronounced: from 0.0943 to 0.1645, an increase of +74.4% with the same 40% pruning.

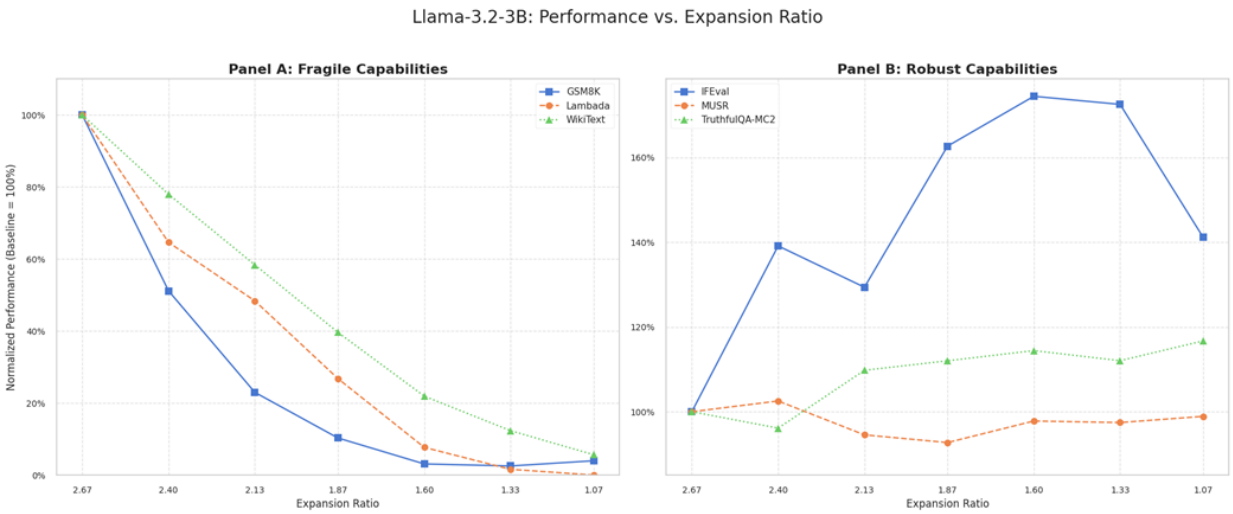


Figure 5.9 The dichotomy visualized in Llama-3.2-3B. Panel A shows the progressive collapse of fragile capabilities (knowledge and reasoning) as the expansion ratio decreases. Panel B reveals the paradox: robust capabilities (instruction-following and truthfulness) improve dramatically, with IFEval reaching a peak of 174% of baseline at 1.6x expansion (40% pruning). Performance is normalized to the base model (2.67x expansion = 100%).

The tables in the paper document how this behavior holds across the entire spectrum of pruning percentages. For example, in the 1B model we can see how IFEval improves progressively: 0.1035 (base) → 0.1423 (10%) → 0.1275 (20%) → 0.1811 (30%) → 0.1516 (40%) → 0.1534 (50%). The improvement is not perfectly linear, but the upward trend is clear even while reasoning capabilities are collapsing.

The second paper, CFSP, gave us the data-driven method. Although we haven't adapted their algorithm, we did take the idea of where the dynamic selection criterion should be applied.

CFSP first evaluates importance at the level of complete modules, then focuses on individual neurons within each module. Their formula for calculating fine-grained importance is complex and includes multiple terms that normalize the weights of `down_proj`, `up_proj`, and `gate_proj` in different ways, combining activation magnitude ratios with weight magnitude ratios.

Here's where the hybrid implementation (section 5.2) differs by taking a simpler approach. Instead of adopting all the complexity of CFSP's coarse-to-fine method, you've taken just the core idea: capturing activations at `down_proj` (`X_d_norm`). Then, instead of using its multi-term formula, you've extended the magnitude-based formula from the static method. You calculate the structural component by summing the normalized scores of `gate_proj`, `up_proj`, and `down_proj`, and then multiply it by the captured activations.

This simplification works for a fundamental reason: if a neuron has great structural potential but rarely activates with your data, the product will be close to zero and that neuron becomes a candidate for removal. It's a filter that combines structural robustness with the dynamic specificity of activations.

This is perhaps the most important lesson from this section: academic papers aren't bibles you need to follow to the letter. They're sources of fundamental ideas you can adapt to your context. We've adapted the formula from the first paper because experiments demonstrate its superiority over alternatives and because it exhaustively documents the dichotomy across multiple configurations. We've taken the activation intuition from the second paper because it makes practical sense: measuring what actually gets used with your data. And we've created something new: a simple hybrid method that maintains its effectiveness.

5.4 Hands-on lab

Width pruning is one of the most versatile and focused techniques for modifying a model. You can make massive reductions in its processing capacity or small adjustments to modify behaviors. In this section, you will find a series of exercises designed to expand your intuition when using this technique.

- Hardware alignment (Tensor Cores):
  - What to do: Modern GPUs process matrices much more efficiently when their dimensions are multiples of 64 or 128. Modify the code of the `prune_neuron_pairs` function (or use the `expansion_divisor` parameter in OptiPFair) to ensure that the resulting `intermediate_size` is divisible by 64, regardless of the chosen pruning percentage.
  - Questions to answer: How does the exact number of pruned neurons change to meet this restriction? If you run the inference benchmark, do you notice any improvement in tokens/second compared to an arbitrary size (for example, 6554) on your current hardware?
- Finding the breaking point (Stress Testing):
  - What to do: In the chapter, we've seen that 40% pruning degrades reasoning but improves instruction-following. Push this to the limit. Generate models with 60% and 75% pruning using the static method (weight magnitude). Evaluate them with `generate_text` using complex prompts.
  - Question to answer: At what percentage does the model stop generating grammatically correct sentences? Is the collapse gradual or sudden? Do

you observe the model becoming repetitive or just generating noise?

- Validation by inversion (Sanity Check):
  - What to do: Modify the selection logic to prune neurons with the highest importance score (the ones that are theoretically most important) keeping the weakest neurons.
  - Question to answer: Compare the perplexity of this "inverted" model against a correctly pruned one. If the magnitude metric is valid, the inverted model should be unusable. Is this the case?
- Implementing mixed-domain calibration:
  - What to do: Create a hybrid calibration approach by combining samples from two contrasting datasets (for example, 50% WikiText + 50% GSM8K, or 50% SMS + 50% TruthfulQA). Create a mixed-calibration model with 20% pruning and compare it against the purely Wiki and SMS models.
  - Questions to answer: Does the mixed-calibration model achieve a "middle ground" performance on both domains, or does it perform worse than specialized models on their respective domains? Is there a benefit to multi-domain calibration, or does specialization always win? What happens to unexpected capabilities like instruction-following?

5.5 Summary

- Our width pruning technique modifies LLMs by removing specific neurons from MLP expansion layers in GLU architectures, offering more fine-grained control than depth pruning's aggressive block elimination. This technique can reshape model behavior without destroying all capabilities.

- In GLU architectures, the `gate_proj` and `up_proj` layers work as a synchronized pair. Neurons must be pruned simultaneously in both projections based on a combined importance score.
- The reconstruction process creates new, smaller Linear layers while preserving connections with attention modules. By maintaining `hidden_size` dimensions at module boundaries, the surgery remains internal to the MLP block without breaking the Transformer architecture.
- Data-driven neuron selection combines structural importance (weight magnitudes) with dynamic importance (activation norms). This hybrid approach multiplies a neuron's structural score by its actual usage during calibration, ensuring only neurons that are both structurally significant and actively used survive.
- PyTorch hooks capture activations at `down_proj's` input, after the gating mechanism but before dimensionality reduction, using `register_forward_hook()`. The L2 norm measures activation intensity, rewarding specialist neurons with sharp peaks over those with constant low-level noise.
- Domain-specialized models emerge from identical pruning percentages applied with different calibration data. A WikiText-calibrated model preserves neurons for formal text processing, while an SMS-calibrated model retains neurons for conversational language, despite sharing the same architecture externally.
- The efficiency gains from width pruning are most visible in compute-bound scenarios.
- Energy efficiency gains result from architectural changes, not behavioral modifications. However, width pruning's benefits are limited by the attention module bottleneck.

- Aligning the resulting `intermediate_size` to be divisible by 64, 128, or 256 maximizes GPU Tensor Core utilization in compute-bound scenarios. This optimization matters most in production deployments with high traffic on modern GPUs (A100/H100).
- Research papers "Fragile Knowledge, Robust Instruction-Following" (Martra, 2025) and "CFSP" (Wang et al., 2024) validate the dichotomy between degrading and improving capabilities, and the effectiveness of activation-based importance metrics. The practical implementation adapts their core insights into a simpler hybrid approach that combines structural capacity with dynamic usage.

6 Knowledge recovery through distillation

This chapter covers

- Deciding what to prune before training
- Transferring knowledge to smaller models
- Recovering lost capabilities efficiently
- Applying proven recovery strategies

Structural pruning — both depth and width — tends to lead to a loss of model knowledge; the more you prune, the greater that loss. Recovering that knowledge is one of the most critical stages in the model rearchitecting pipeline: it's what determines whether the model can match, or even surpass, the base model.

That recovery depends on two things: which parts you've removed, and how you train to recover the lost knowledge. The first lesson I learned about this came before any training loop.

I remember the first day I compared two recovery experiments. Both started from models with the same number of Transformer blocks removed, but in one I had removed the blocks by importance and in the other I had removed the last blocks. The difference was striking: the model with importance-based pruning recovered its performance in significantly less training time. That moment changed how I saw optimization. Knowledge recovery doesn't start when we apply Knowledge Distillation; it starts when we decide which parts of the model to keep.

In Chapter 2, you saw the fundamentals of Knowledge Distillation: how a teacher model transfers its knowledge to a smaller student using only the model outputs (soft labels).

In this chapter, you're going to go further. You're going to walk through the complete recovery process, following a series of experiments available in the book's repository, which replicate the workflow I follow in a real optimization project, from the decision of which blocks to remove to deciding which advanced Knowledge Distillation techniques to apply.

This time you will use a system that, in addition to label alignment (logits), also uses the dataset (hard labels) and aligns the model's intermediate blocks. More powerful but also more complex, because it introduces a problem: How do we align the blocks of both models when the numbers don't match?

Let’s start with the decision that influences the whole process the most: which parts of the model do we remove?

6.1 Choosing what to prune

Knowledge Distillation training consumes a considerable amount of resources. Running distillation on a GPU for several hours or days, only to later discover that you chose the wrong blocks to remove, is frustrating and costly. That’s why, before investing GPU time in recovery, we need to identify which method loses the least critical knowledge and thus can recover faster and more effectively.

Depth pruning includes two families of strategies for selecting blocks: data-driven methods, which use importance analysis with calibration data, and static methods, which follow simple heuristics like removing the

last blocks. Depth pruning also includes the protection hypothesis, which suggests always preserving the final Transformer blocks because they contain the critical refinement for the model's output.

All these strategies have their own logic, and although we can find literature defending each of them, in practice it’s best to run experiments to identify which one works best for our goals. This choice determines not only the number of hours we'll have to dedicate to the model for it to recover its capacity, but also the recovery ceiling.

6.1.1 Experiment configuration

To identify which block selection strategy works best, we designed an experiment that compares the four most relevant approaches. The base model is google/gemma-3-270m (https://huggingface.co/google/gemma-3-270m), which has 18 Transformer blocks; we will remove four of these blocks. This represents approximately 22% of the model's depth; aggressive enough to produce measurable degradation, but within the range where knowledge recovery remains feasible on standard hardware.

NAVIGATING THIS CHAPTER'S NOTEBOOKS

This chapter includes multiple notebook types:

- Research Experiments (EXP01-04): These notebooks document the complete experimental process comparing four pruning strategies on A100 GPUs. All strategies are compared using 2K samples, while only the winner (EXP01) scales to 15K and 40K. Use these notebooks to understand the methodology. Results analyzed in tables 6.1-6.4.
- Your Starting Point (NB01): The winning strategy (EXP01) implemented for Google Colab's free T4 GPU. This is where you should begin hands-on work. Covers sections 6.2-6.3 implementation.
- Width pruning compatibility (NB02): Demonstrates width pruning recovery, runs on a T4 GPU. Optional, for section 6.7.
- Challenge Lab (NB03): Your mission to beat the baseline model. Uses full 40K dataset; requires better hardware than T4.

For an initial evaluation, we limit the training to 2,000 records from the Cosmopedia dataset. It’s sufficient to determine the optimal strategy and reduce the computational cost of full-scale training.

After identifying the winning strategy, we'll scale the experiment to larger datasets to see how far we can push the recovery.

The four strategies we're going to compare are:

- Experiment 1 (EXP01): Data-driven individual. We apply the importance analysis by cosine similarity that you learned in chapter 4. Identifying the 4 individual Transformer blocks that contribute the least to the transformation of the representations, regardless of whether they're consecutive or not.
- Experiment 2 (EXP02): Data-driven consecutive. We use the same importance analysis, but with an additional constraint: the four removed blocks must be consecutive.
- Experiment 3 (EXP03): Last blocks. We remove the final four blocks of the model (blocks 14-17). This is the simplest heuristic and the one we used in chapter 2. The hypothesis is that the final blocks refine specific output details, but the fundamental knowledge resides in earlier blocks.
- Experiment 4 (EXP04): Final block protection. We remove blocks 12-15, explicitly preserving the last two Transformer blocks. This strategy responds to the protection hypothesis you saw in chapter 4: the final blocks are critical for output quality and should always be preserved.

For the data-driven strategies (EXP01 and EXP02), we use cosine similarity analysis to measure each block's contribution to representation transformation (the same technique you learned in Chapter 4). The `optipfair.analyze_layer_importance()` function (listing 6.5, later in this chapter) calculates importance scores for each of the 18 blocks, allowing us to identify candidates for removal. EXP01 removes the four lowest-scoring blocks regardless of position, while EXP02 constrains removal to consecutive blocks. In contrast, EXP03 and EXP04 use simple heuristics: removing the final blocks or preserving specific layers based on hypothesis rather than measurement. Figure 6.1

illustrates how these four strategies differ in their block selection patterns.

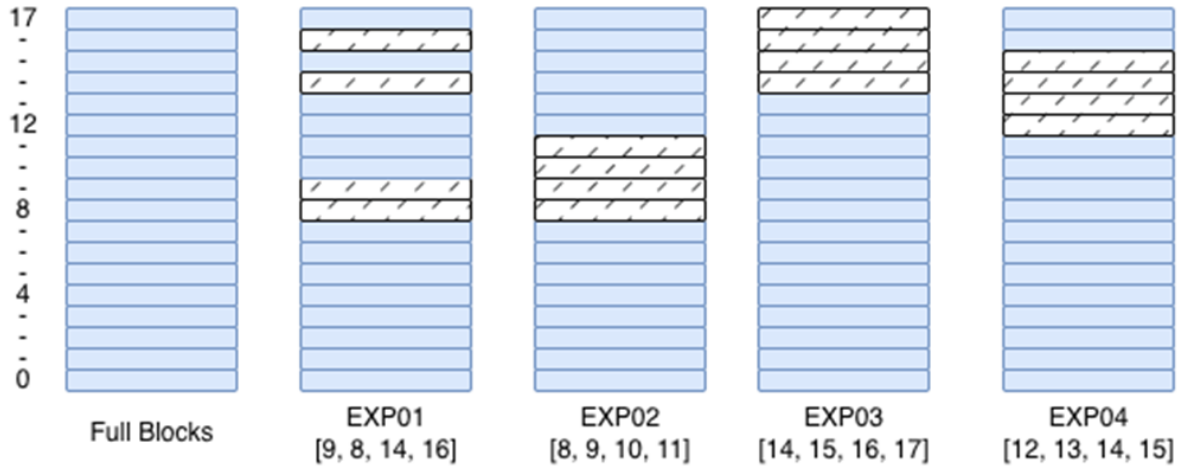


Figure 6.1 Comparison of layer removal strategies. All experiments remove 4 blocks (striped) from the original 18-block model. EXP01-02 use data-driven importance analysis (Listing 6.5), while EXP03-04 follow heuristic rules.

Each strategy produces a pruned model with 14 blocks, with the same structure but the blocks removed differ.

To apply the data-driven strategies (EXP01 and EXP02), we first need a dataset that allows us to calculate the importance of each Transformer block. The same dataset will also be used for Knowledge Distillation training. The choice of dataset is crucial because it determines what type of knowledge we prioritize during recovery.

Let's see how to prepare it.

6.1.2 Preparing the recovery dataset

The choice of dataset for Knowledge Distillation has a direct impact on recovery quality. In this experiment, we use Cosmopedia (https://huggingface.co/datasets/HuggingFaceTB/cosmopedi

a), a high-quality synthetic dataset generated by advanced language models, and curated by the Hugging Face team

Because it is generated by state-of-the-art LLMs and curated by the Hugging Face team, with a strict validation pipeline to filter raw LLM generation, all content maintains a high standard of coherence and grammatical correctness. There are no incomplete articles, spam, or noisy text that could confuse the distillation process.

It's organized into thematic subsets (stories, tutorials, educational content, web samples) that cover different writing styles and knowledge domains. This allows the model, if it’s our goal, to recover general capabilities instead of specializing in a single type of text.

In real projects, you can use the base model itself to generate your own synthetic dataset for distillation, or even if you use standard datasets like Cosmopedia or rawer datasets like SlimPajama, you can generate some samples with the base model to complete the dataset.

The following listing shows how we load Cosmopedia combining four different subsets to maximize the diversity of the recovery dataset.

Listing 6.1 Loading and mixing Cosmopedia subsets

```
from datasets import load_dataset, Dataset
from torch.utils.data import TensorDataset, DataLoader, random_split

MAX_LENGTH = 512
BATCH_SIZE = 4
RECOVERY_SAMPLES = 2000


dataset_name = "HuggingFaceTB/cosmopedia"
subsets = ["stories", "wikihow", "openstax", "web_samples_v1"] #A
samples_per_subset = int(RECOVERY_SAMPLES / 4) #B
num_samples = samples_per_subset * len(subsets)

all_samples = []
for subset in subsets:
    print(f"  Loading {subset}...")
    subset_data = load_dataset(dataset_name,
        subset, split="train",
        streaming=True) #C
    subset_samples = list(subset_data.take(samples_per_subset))
    all_samples.extend(subset_samples)
    print(f"    ✓ {len(subset_samples):,} samples from {subset}")

distillation_dataset = Dataset.from_dict({'text': [s['text'] for s i
n all_samples]})
```

#A Four subsets with different styles: narratives, instructions, educational, and web
#B Distributes samples evenly across the four subsets
#C Allows loading only the necessary samples without downloading the entire dataset

The combination of these four subsets aims to expose the model to a varied set of examples that allows it to recover general capabilities. With 500 samples from each subset (in the case of 2,000 total samples), we create a balanced dataset that exposes the student model to different linguistic patterns, which is usually enough for us to decide which strategy yields the best results.

Before tokenizing the dataset, we need to load the teacher model and configure it correctly for distillation.

Listing 6.2 Teacher model and tokenizer configuration

```
MODEL_NAME = "google/gemma-3-270m"
teacher_model = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    torch_dtype=torch.bfloat16 if torch.cuda.is_available() else tor
ch.float32, #A
    device_map="auto" if torch.cuda.is_available() else None
)

teacher_model.eval() #B
for param in teacher_model.parameters():
    param.requires_grad = False #C


tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
```

#A bfloat16 on the GPU reduces memory without significant precision loss
#B Disables dropout and batch normalization
#C Freezes all teacher parameters - we'll never train it

We freeze the teacher model, because we don't need to calculate gradients for its parameters, which saves GPU memory and speeds up the process.

With the model loaded, the next step is to convert our text into numerical tensors that PyTorch can process.

Listing 6.3 Efficient tokenization in batches

```
dataset_list = list(distillation_dataset)
texts = [item['text'] for item in dataset_list]

tokenized_data = []
batch_size = 100 #A
for i in tqdm(range(0, len(texts), batch_size), desc="Tokenizing"):
    batch_texts = texts[i:i+batch_size]
    batch_tokens = tokenizer(
        batch_texts,
        truncation=True,
        padding="max_length",
        max_length=MAX_LENGTH,
        return_tensors="pt"
    )
    tokenized_data.append(batch_tokens)

input_ids = torch.cat([batch['input_ids']
➥ for batch in tokenized_data], dim=0) #B
attention_mask = torch.cat([batch['attention_mask']
➥ for batch in tokenized_data], dim=0)

full_dataset = TensorDataset(input_ids, attention_mask) #C
```

#A Processes 100 texts at a time to balance speed and memory usage
#B Concatenates all batches into a single tensor
#C Allows for efficient indexing during training

Batch tokenization is more efficient than processing each text individually, especially with large datasets. In the experiment notebooks, I use a batch of 1000, because running on a A100 GPU provides more resources. The Hugging Face tokenizer is optimized to process multiple texts simultaneously.

The final step is splitting our dataset into training and validation sets. This separation is critical to get honest metrics of the recovery process. If we validate the model with the same training data, we could be measuring memorization capacity rather than abstraction.

Listing 6.4 Train/Val split and creation of DataLoaders

```
generator = torch.Generator().manual_seed(42) #A
train_size = int(0.9 * len(full_dataset))
val_size = len(full_dataset) - train_size

train_dataset, val_dataset = random_split(
    full_dataset,
    [train_size, val_size],
    generator=generator #B
)

train_dataloader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True #C
)

eval_dataloader_raw = DataLoader(
    val_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False #D
)
```

#A Fixed seed ensures identical split across all runs
#B Generator with seed guarantees reproducibility
#C Shuffling during training improves generalization
#D No shuffling in validation maintains original order for consistent evaluation

The 90/10 split between train and validation prevents a common problem in Knowledge Distillation: overfitting to the training data. If we evaluated perplexity on the same data we trained with, we'd get artificially optimistic results that wouldn't reflect the model's true capacity. The validation set gives us an honest measure of how effectively we've transferred knowledge. In later experiments, with a higher number of recovered records, the split is 80/20 to increase the number of records used in validation and get more reliable data.

With the dataset prepared and correctly split, we're ready to run the four experiments and analyze which pruning strategy produces the best candidate for recovery.

6.1.3 Selecting the best candidate

To compare the four pruning strategies, we need to ensure the model maintains the ability to generate understandable text and reasoning capacity. So we'll use perplexity (PPL) as our main fluency metric, combined with five standard benchmarks to measure specific capabilities: arc_easy, hellaswag, lambada_openai, piqa, and winogrande.

Perplexity is ideal for our goal because we're creating a generalist model. But it's fundamental to understand that metric selection should align with the final purpose of your model. If you were optimizing a model for tool calling in an agent system, the main metric should be accuracy in tool selection accuracy, not perplexity. In specialized projects, you'll often need to create custom benchmarks that capture exactly the capabilities your application requires.

We evaluate each strategy at two critical moments: immediately after pruning (without any recovery) and after applying basic Knowledge Distillation with 2,000 samples. This comparison shows us not just which strategy degrades less, but which has greater recovery potential.

Table 6.1 shows the state of the four models right after removing 4 Transformer blocks, before any recovery attempt.

Table 6.1 Initial degradation by pruning strategy (without recovery)

Experiment	Strategy	Blocks Removed	Perplexity(PPL)	Capabilities Retention
Teacher			13.38	100
EXP01	Data-driven	[9, 8, 14, 16]	126	78.6
EXP02	Data-Driven Consecutive	[8, 9, 10, 11]	490	65.8
EXP03	Last blocks	[14, 15, 16, 17]	264	66.6
EXP04	Last Blocks (preservation)	[12, 13, 14, 15]	464	67

The results indicate that EXP01, which removes the least important blocks identified through cosine similarity analysis, suffers the lowest degradation. With a perplexity of 126, it's approximately 9x that of the teacher (compared to the others ranging from 20x to 36x). More importantly, it retains 78.6% of reasoning capabilities (average of benchmarks: arc_easy, hellaswag, lambada_openai, piqa, and winogrande).

In contrast, EXP04 (preserving the two final blocks) and EXP02 (data-driven consecutive) show the biggest degradations, with perplexities above 460 and capability retentions below 68%. This result directly challenges the protection hypothesis: preserving final blocks doesn't guarantee better performance if the intermediate removed blocks are critical for information processing.

With these numbers we already have a clear candidate for the recovery process, but before making a final decision we'll perform a small recovery test by running a KD process with 2,000 records from the cosmopedia dataset for 5 epochs. A training can take between 3 and 10 minutes per model,

depending on the GPU processing capacity used. Table 6.2 shows the post-recovery results.

Table 6.2 Capabilities recovered with KD (2,000 samples)

Experiment	Strategy	Perplexity(PPL)	Capabilities retention	Recovery gain (pp)
Teacher	-	13.38	100	-
EXP01	Data-driven	18.9	86.7	+10.31
EXP02	Data-Driven Consecutive	21.2	86.4	+31.31
EXP03	Last blocks	21.4	82.4	+23.72
EXP04	Last Blocks (preservation)	21.3	86	+28.36

EXP01 maintains its advantage after recovery. With 86.7% retention in capability benchmarks and a perplexity of 18.9, it outperforms all other strategies in both dimensions.

The "Recovery gain" column reveals a relevant pattern: EXP01 shows the smallest recovery gain (+10.31 pp), yet it achieves the best final result. This is not a contradiction, it reflects that EXP01 started from a significantly stronger baseline after pruning, so there was simply less ground to recover. EXP02 (+31.31 pp) and EXP04 (+28.36 pp) show much larger recoveries, as expected given how severely degraded their initial bases were. But even with that greater recovery margin, they still can't reach EXP01's final capability retention.

Now that you've seen that the individual data-driven strategy produces the best results, let's look at how to create this model in code. The following listing shows the complete process: from importance analysis through pruning and preparing the Student for training.

Listing 6.5 Removing four less important layers

```
import optipfair as opf

student_model = deepcopy(teacher_model) #A
importance_scores = opf.analyze_layer_importance(student_model,
    train_dataloader,
    show_progress=True
) #B

LAYERS_TO_REMOVE = sorted(
    importance_scores.keys(),
    key=lambda x: importance_scores[x]
)[:4] #C

print (LAYERS_TO_REMOVE)
student_model = opf.prune_model_depth(
    model=student_model,
    layer_indices=LAYERS_TO_REMOVE,
    show_progress=True,
) #D

for param in student_model.parameters():
    param.requires_grad = True #E
```

#A Create an independent copy of the base model
#B Calculate the importance of each block using the calibration dataset
#C Select the 4 blocks with the lowest importance scores
#D Remove the selected blocks from the model
#E Unfreeze the parameters inherited from the frozen teacher

The `deepcopy` function creates a copy of the teacher, but it retains the `requires_grad=False` state that we set when loading the model (listing 6.2). Without it, PyTorch wouldn't calculate gradients during training and the student model couldn't learn. It's an easy mistake to make that results in silent training failure.

For the rest of this chapter, we'll continue working exclusively with this model. In the next section, we'll explore advanced Knowledge Distillation techniques using the model's intermediate hidden states, so we won't just learn

from the teacher model's output. But first we need to solve a technical challenge: how do we align the internal representations when the student has fewer blocks than the teacher?

6.2 Aligning Teacher and Student

You've just created a model with 14 Transformer blocks and want to align it with one that has 18. If we were doing classic distillation, based on hard labels (dataset) and soft labels (logits), the structural difference wouldn't matter, since you simply compare the final output of both models and calculate the loss. But our goal now isn't just for the student to reach the same conclusion as the teacher, but to reach it in the same way, which is especially valuable when the intermediate representations themselves are the final product, as in embedding models or RAG retrievers. That's why we'll use a technique known as Feature Alignment, which is based on aligning the result produced by each intermediate Transformer block.

This raises a practical problem: our Teacher has 18 Transformer blocks, while the pruned Student only has 14. We need to decide how they align, so three different strategies have been tested: uniform mapping, last-block mapping, and original indices mapping, whose results are shown in figure 6.2.

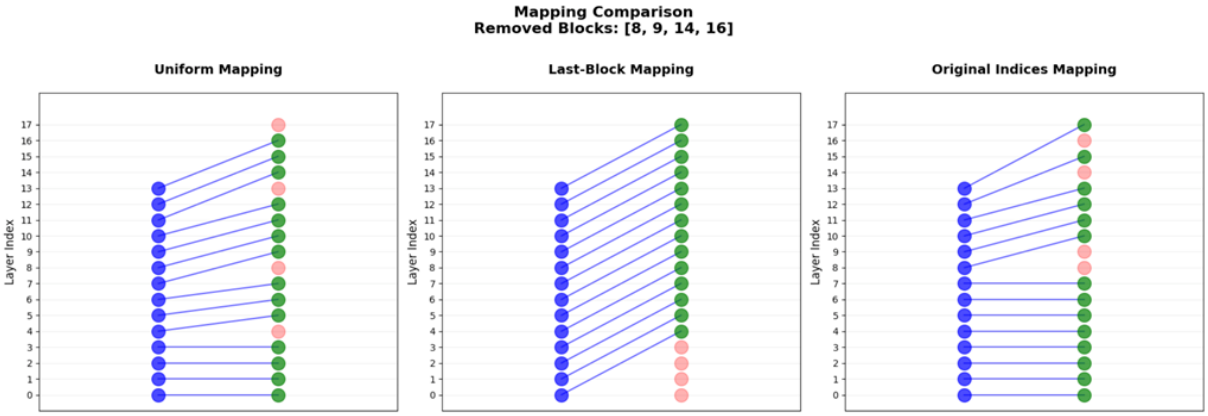


Figure 6.2 The three mapping strategies: Uniform mapping distributes proportionally, last mapping aligns with the deepest layers, and original mapping preserves the original architectural connections, leaving gaps where the removed layers were.

In uniform mapping, each Student block is mapped proportionally to its position in the Teacher:

```
n_student = len(student_model.model.layers)
n_teacher = len(teacher_model.model.layers)

uniform_map = []
for i in range(n_student):
    teacher_idx = int(i * n_teacher / n_student) #A
    uniform_map.append(teacher_idx)
```

#A Scales student position proportionally to teacher's depth

Block 7 of the Student (middle of the model) maps to block 9 of the Teacher (also roughly the middle). The Student has access to knowledge from all "regions" of the Teacher: early, middle, and late blocks.

The last mapping focuses exclusively on final reasoning. It aligns the top blocks of the student with those of the teacher:

```
offset = n_teacher - n_student #A
last_map = [i + offset for i in range(n_student)]
```

#A Calculates the shift needed to align student blocks with teacher's deepest blocks

This way, block 0 of the Student maps to block 4 of the Teacher, block 1 to 5, and so on until block 13 of the Student aligns with the last block of the Teacher. The hypothesis is that the final blocks contain the most refined knowledge and the most abstract representations.

The third alternative, original mapping, maintains the original alignment of the blocks:

```
def create_layer_map_original_indices(student_model,
    teacher_model,
    removed_layers):
    n_teacher = len(teacher_model.model.layers)

    original_indices = []

    for i in range(n_teacher):
        if i not in removed_layers: #A
            original_indices.append(i) #B

    layer_map = original_indices

    return layer_map #C

original_map = create_layer_map_original_indices(
    student_model,
    teacher_model,
    LAYERS_TO_REMOVE #B
)
```

#A Checks if the teacher block survived the pruning process
#B Keeps track of original positions for blocks that weren't removed
#C Returns mapping that preserves pre-pruning architectural relationships

Instead of creating a position-based mapping, it preserves the architectural relationships that existed before pruning. Each block of the Student maps to its original block in the Teacher model.

With these three mapping strategies implemented, we now have the tools to align asymmetric architectures. Now it's time to put them into practice: train the student using the teacher's knowledge and compare which of these strategies produces the best results.

6.3 Recovering the knowledge

We already have the pruned model with the best strategy (data-driven individual) and three different ways to map its 14 blocks to the teacher's 18. Now comes the crucial part: training the student to recover the lost knowledge.

In chapter 2, we trained the student using only logits. This allows the student to learn to imitate the teacher's uncertainty and second-best candidates.

The next step is to incorporate:

- The "absolute truth" of the dataset, using what's known as hard labels. With them, the model learns what to answer (the correct answer), but loses the nuances of reasoning.
- The hidden states (the intermediate result of each block) to learn, not just the final result, but the internal thought process.

Our learning process must synchronize these three levels simultaneously: hard labels, soft labels, and hidden states.

To achieve this, we need to build a composite function that combines the three elements with specific weights. Let's see how this function is built.

6.3.1 The compound loss function

The loss function you're going to build has to perform three different loss calculations and combine them by assigning a weight to each one.

Before getting into the implementation of each calculation, let's look at the general structure and the "signature" of our function to understand what data we need.

Listing 6.6 General structure of the compound loss function

```
def compute_compound_loss(
    student_logits,
    teacher_logits,
    student_hiddens,
    teacher_hiddens,
    labels,
    layer_map,
    alpha=0.4,
    beta=0.4,
    gamma=0.2,
    temperature=2.0
):

    loss_task = ...
    loss_logits = ...
    loss_hidden = ...

    total_loss= alpha * loss_task + beta * loss_logits + gamma * los
s_hidden

    loss_dict = {
         'total': total_loss.item(),
         'task': loss_task.item(),
         'logits': loss_logits.item(),
         'hidden': loss_hidden.item()
     }

     return total_loss, loss_dict
```

The function takes the complete forward pass outputs from both models. The `student_hiddens` and `teacher_hiddens` variables are tuples or lists containing the activation tensors from all model blocks. The `layer_map` argument specifies how to map the Student's layers with the teacher's layers.

The `alpha`, `beta`, and `gamma` parameters control the weight assigned to each learning signal: task loss, logit loss, and hidden loss respectively. The default values (0.4, 0.4, 0.2) assign equal weight to two complementary signals: learning

the correct next token directly from the dataset (*alpha*), and imitating the teacher's confidence distribution across the vocabulary (*beta*). Together they represent 80% of the total loss, with intermediate representation alignment as a secondary stabilizer (*gamma*). This balance, prioritizing functional behavior over internal state imitation, works as a robust baseline across generative transformer architectures, regardless of model size or family. As we will discuss in sections 6.3.1 and 6.5.1, this balance shifts only when the internal representations themselves are the final product, such as in embedding or classification models.

These three weights indicate how the optimizer’s attention should be distributed among the signals, so it's common for them to sum to 1.0. It's not mandatory but helps interpret the influence of each one and compare different experiments. If you sum more than 1.0, you'd amplify the gradients and could destabilize training; if you sum less, the gradients would be smaller and learning would slow down unnecessarily. The important thing is to remember that when you raise one of these parameters, you should lower another to compensate.

Before diving into the implementation of each loss component, let's visualize how these signals work together. Figure 6.3 illustrates the complete flow: the teacher and student both process the same input ("Paris is the capital of..."), generating not just final predictions but intermediate representations at each layer. The compound loss function captures three types of knowledge transfer simultaneously.

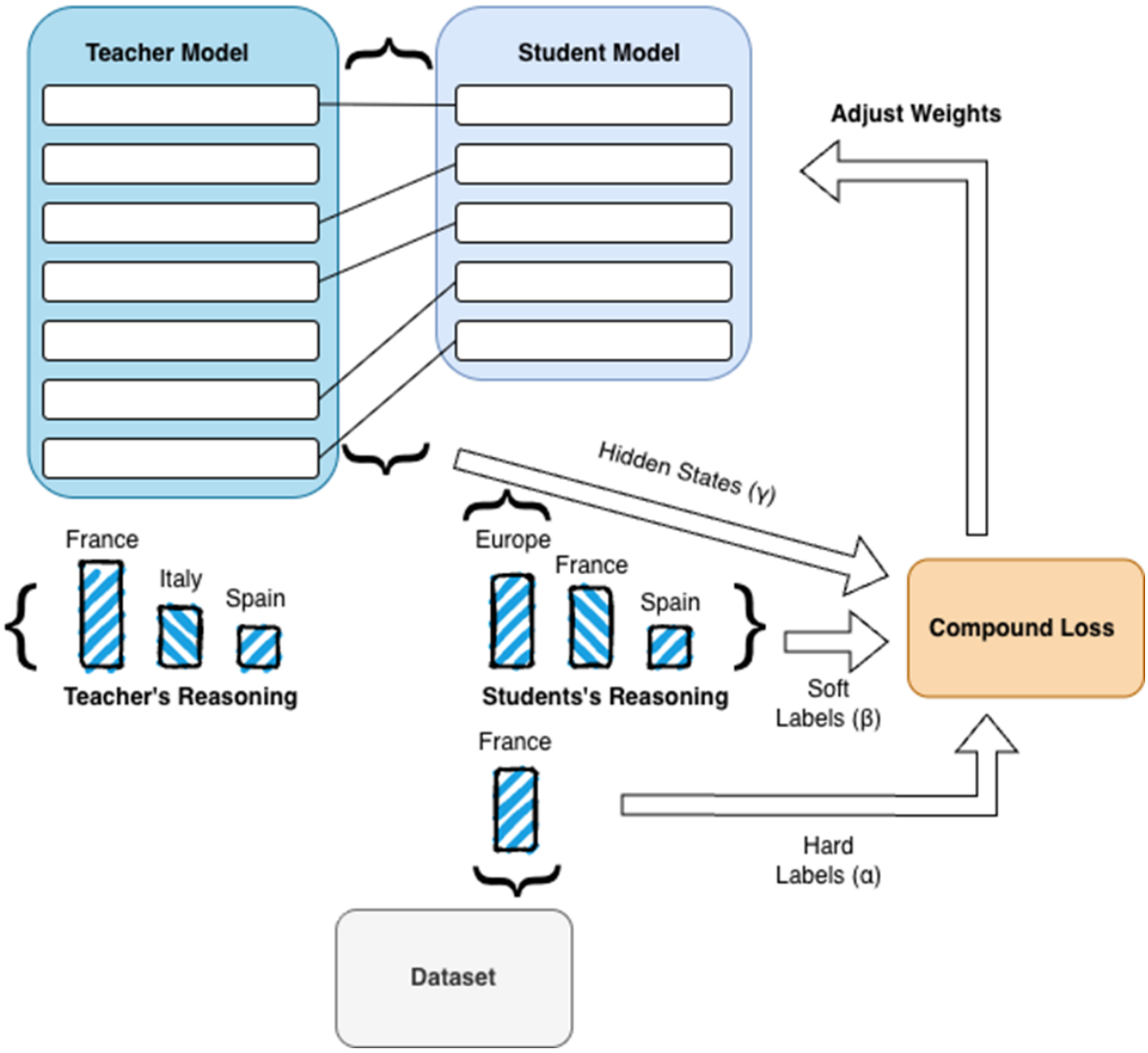


Figure 6.3 The compound loss combines three signals: Hard Labels from the dataset (the ground truth "France"), Soft Labels capturing the teacher's probability distribution across candidates, and Hidden States alignment between corresponding Transformer blocks. The weights α, β, γ control each component's influence on the total loss.

The temperature parameter softens the teacher's probability distributions. Figure 6.4 illustrates this effect with the prompt "Work hard play": at standard inference temperature (T=1.0), the teacher assigns an overwhelming 94.5% probability to "hard", with minimal consideration for alternatives like "well" (3.7%), "good" (0.3%), or "harder" (0.2%). Without temperature scaling, the student would only learn that "hard" is correct and completely ignore these

alternatives. With temperature=2.0, those probabilities get softened dramatically—"hard" drops to 23.9% while "well" rises to 4.7%, "good" to 1.3%, and "harder" to 1.2%. This shows the student that the teacher also considered "well" as a viable option and that "good" or "harder" aren't completely absurd answers in certain contexts.

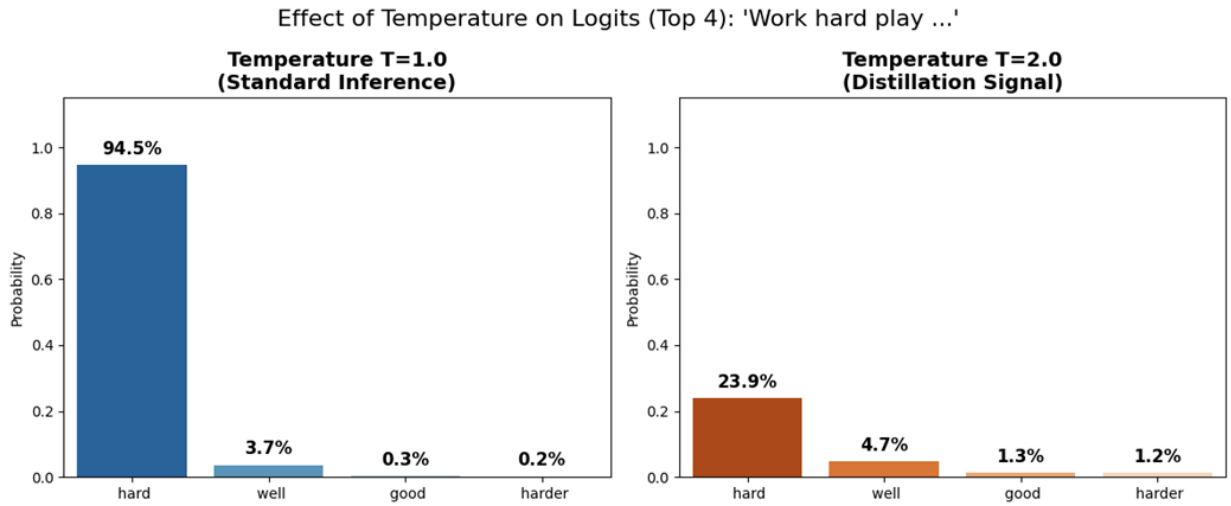


Figure 6.4 Temperature softens the teacher's probability distribution. At T=1.0, the model assigns 94.5% to "hard". At T=2.0, probabilities spread across alternatives (23.9%, 4.7%, 1.3%, 1.2%), exposing the teacher's reasoning process to the student.

CALCULATING TASK LOSS

We begin calculating the task loss (Cross-Entropy) using hard labels from the dataset. The goal is straightforward: ensure that the model, despite trying to imitate the teacher, retains the ability to accurately predict the next token from the training data.

Let's look at the `task_loss` calculation:

```
shift_logits = student_logits[..., :-1, :].contiguous() #A
shift_labels = labels[..., 1:].contiguous() #B
loss_task = F.cross_entropy(
    shift_logits.view(-1, shift_logits.size(-1)), #C
    shift_labels.view(-1),
    ignore_index=-100 #D
)
```

#A Discard last logit (no corresponding label)
#B Discard first label (no corresponding logit)
#C Reshape to (batch_size * seq_len, vocab_size) for loss calculation
#D Skip padding tokens (marked as -100)

Lines #A and #B might seem complicated at first, but they have a very simple mission: token shifting. Language models learn to predict the next token, so we need to shift the labels one position forward, as shown in figure 6.5. If we have "Work hard play hard" in the dataset, we want the model to predict ["hard", "play", "hard", "<eos>"]. This means the logits at position 0 ("Work") must be compared against the label at position 1 ("hard"), the logits at position 1 with the label at position 2, and so on. We achieve this alignment by removing the last token from the logits (which has no next token to predict) and the first from the labels (which isn't predicted by anything).

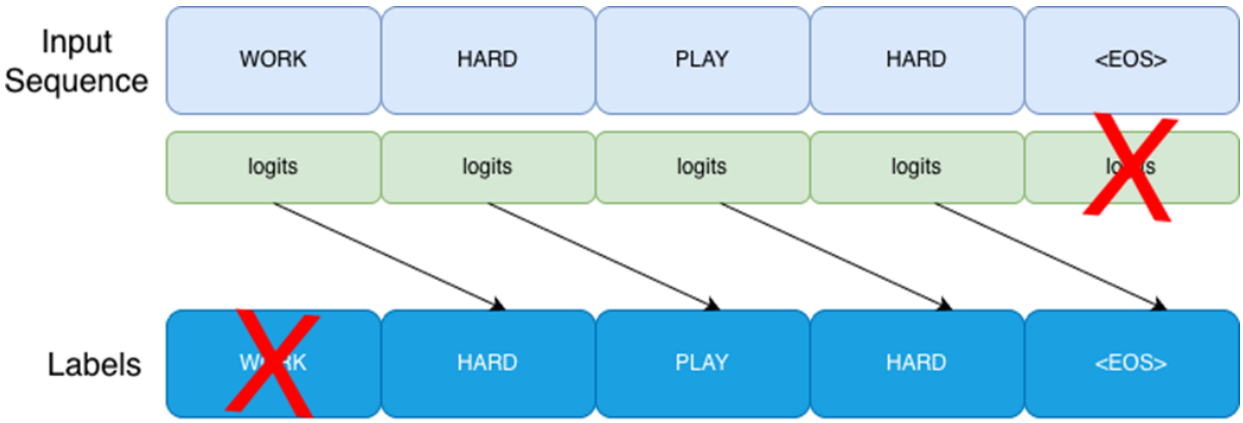


Figure 6.5 Token shifting aligns predictions with labels. Each logit position predicts the next token in the sequence. The first label (WORK) is discarded because no logit predicts it, and the last logit is discarded because there's no future token to predict after <EOS>. The arrows show how logits[0] compares against labels[1], logits[1] against labels[2], and so on.

The `ignore_index=-100` parameter is a PyTorch convention. When we tokenize with padding, tokens that aren't a real part of the text are marked with -100, and this line tells the loss function not to take them into account in the calculation. Without this, we'd be training the model to predict padding tokens, which makes no sense at all.

CALCULATING LOGIT LOSS

Soft labels (the probabilities the teacher assigns to all the words in the vocabulary) capture not only which token is correct, but how confident the model is in each option. To achieve this, we use PyTorch's `F.kl_div` (`torch.nn.functional.kl_div`) function, which acts like a ruler to measure the distance between the two probability distributions:

```
student_soft = F.log_softmax(
    student_logits / temperature,
    dim=-1) #A
teacher_soft = F.softmax(
    teacher_logits / temperature,
    dim=-1) #B

student_soft = student_soft[..., :-1, :].contiguous()
teacher_soft = teacher_soft[..., :-1, :].contiguous()

loss_logits = F.kl_div(
    student_soft.view(-1, student_soft.size(-1)),
    teacher_soft.view(-1, teacher_soft.size(-1)),
    reduction='batchmean' #C
) * (temperature ** 2) #D
```

#A Log-softmax for the student model
#B Standard softmax for the teacher model
#C Batchmean divides by the total number of elements
#D Multiply by T² to prevent gradients from becoming too small when dividing by temperature

Using `log_softmax` in the student and `softmax` in the teacher isn't arbitrary. PyTorch's `kl_div` function expects the first argument (the predictions) to be in log space to avoid numerical underflow issues when probabilities are very small.

NOTE

The difference between `softmax` and `log_softmax` is subtle but important. If you have logits [2.0, 1.0, 0.1], `softmax` produces probabilities [0.659, 0.242, 0.099], which sum to 1.0. Instead, `log_softmax` produces [-0.417, -1.417, -2.317], which are the natural logarithms of those probabilities. PyTorch uses `log_softmax` internally in functions like `kl_div` because working in logarithmic space avoids numerical issues with extremely low probabilities (for example, 0.0001 becomes -9.21, a manageable value). If you tried to calculate log(0.0001) after a `softmax`, you could get invalid values in extreme cases.

We've already calculated two components: the loss over the correct predictions from the dataset (task loss) and the divergence between the probability distributions of both models (logits loss). One critical component remains: aligning the internal representations that each block builds during the reasoning process.

CALCULATING HIDDEN LOSS

With Hidden Loss or feature alignment, we force the intermediate layers of the student to process information similarly to the teacher.

To measure the similarity between the student model’s block output and the teacher model block’s output, we use Cosine Distance, the same metric you used in chapter 4 to decide the contribution of the transformer block to the model's final result. Cosine similarity measures direction rather than magnitude. Imagine the representations as vectors in a 640-dimensional space; that’s the hidden size of Gemma-270M.

What matters isn't how "big" each component of the vector is, but where that vector points in multidimensional space. Two vectors [1, 2, 3] and [10, 20, 30] have exactly the same direction (cosine similarity = 1.0) even though their magnitudes are completely different.

Let's look at the code to perform this calculation:

```
if gamma > 0:
    loss_hidden = 0.0
    num_aligned_layers = len(layer_map)

    for student_idx, teacher_idx in enumerate(layer_map):
        student_h = student_hiddens[student_idx]
        teacher_h = teacher_hiddens[teacher_idx]

        student_flat = student_h.reshape(
           -1,
           student_h.size(-1)) #A
        teacher_flat = teacher_h.reshape(
           -1,
           teacher_h.size(-1)) #A

        student_norm = F.normalize(
           student_flat,
           p=2,
           dim=1) #B
        teacher_norm = F.normalize(
           teacher_flat,
           p=2,
           dim=1) #B

        cos_sim = (student_norm * teacher_norm).sum(dim=1).mean() #C
        loss_hidden += (1 - cos_sim) #D

    loss_hidden = loss_hidden / num_aligned_layers #E
else:
    loss_hidden = torch.tensor(0.0, device=student_logits.device)
```

#A Flatten batch and sequence into one dimension: [batch*seq, hidden]
#B Normalization removes scale differences
#C Compute cosine similarity
#D Convert to cosine distance
#E Average the loss across all aligned blocks

The loop over `layer_map` is where the three mapping strategies from section 6.2 come into play. If you're using uniform mapping, the student will learn from representations distributed uniformly throughout the teacher's depth. With last mapping, it'll focus only on the deepest representations. And with original mapping, each student block will learn

from the teacher block that originally occupied that position before pruning.

The reshape to `[batch * seq_len, hidden_dim]` converts the three-dimensional tensor into a matrix where each row is a representation vector. To understand what this means, let's look at a simplified example.

When we process text, the model works with batches (groups of sequences we process simultaneously to leverage GPU parallelism). Imagine you have a batch of two sequences: "Sand and Sun" and "Time to go". Each sequence has three tokens, and each token is represented with a 4-dimensional vector (in Gemma-270M it would be 640 dimensions, but we're simplifying for the example).

Before the reshape — Tensor of shape [2, 3, 4]:

- First dimension (2): number of sequences in the batch
- Second dimension (3): number of tokens per sequence
- Third dimension (4): size of the representation vector

Sequence 1 ("Sand and Sun"):

[0.1, 0.2, 0.5, 0.9], # "Sand"

[0.8, 0.1, 0.1, 0.5], # "and"

[0.3, 0.9, 0.2, 0.4] # "Sun"

Sequence 2 ("Time to go"):

[0.5, 0.5, 0.1, 0.1], # "Time"

[0.2, 0.2, 0.8, 0.8], # "to"

[0.9, 0.1, 0.5, 0.3] # "go"

After the reshape we get a matrix of shape [6, 4]. That is, the number of sequences in the batch * the number of tokens (2 * 3 = 6) and the size of the representation vector (4). With the content:

[0.1, 0.2, 0.5, 0.9], # Sand

[0.8, 0.1, 0.1, 0.5], # and

[0.3, 0.9, 0.2, 0.4], # sun

[0.5, 0.5, 0.1, 0.1], # Time

[0.2, 0.2, 0.8, 0.8], # to

[0.9, 0.1, 0.5, 0.3] # go

Now you have six independent vectors of four dimensions each. The L2 normalization and the cosine similarity calculation operate on each vector (each row) individually, regardless of which sequence or position they originally came from. In Gemma-270M, using a BATCH_SIZE of 8 sequences of 512 tokens, you would do this same process but with tensors of shape [8, 512, 640] → [4,096, 640].

The calculation is only performed for values of gamma > 0.

COMBINATION AND BALANCE OF WEIGHTS

The three components work together through a simple linear combination. Adding the three losses, applying the corresponding coefficient:

```
total_loss = alpha * loss_task + beta * loss_logits + gamma * loss_h
idden

loss_dict = {
    'total': total_loss.item(),
    'task': loss_task.item(),
    'logits': loss_logits.item(),
    'hidden': loss_hidden.item()
}

return total_loss, loss_dict
```

In the experiments, we assigned 80% of the total weight to the quality of the final response (0.4 Task Loss + 0.4 Logits Loss). The three weights sum to 1.0, which allows us to interpret them directly as percentages of relative importance. With these weights, the model prioritizes generating correct text and replicating the teacher's confidence distribution.

The Hidden Loss represents the remaining 20%, half of the weight assigned to each of the other two components, since Transformer blocks weren't originally trained to produce intermediate representations that are interpretable or comparable by external systems. Their goal during training was to collaborate in predicting the next token. However, by assigning a weight of 0.2, we leverage the semantic structures the teacher learned during its training and alignment phases. This allows us to transfer robustness to the student without restricting its ability to adapt.

ADJUSTING WEIGHTS ACCORDING TO THE OBJECTIVE

The values α=0.4, β=0.4, γ=0.2 work well for generalist models like Gemma-270M, but adjust them based on your goal:

- **Embedding or classification models:** If your final product is vector representations (embeddings for RAG or feature vectors for classification), quality of hidden states is critical. Increase gamma to 0.3 or 0.4 to force strict alignment of intermediate representations. These are the final product that the downstream system will consume, so their quality directly impacts performance.
- **Pure generative models**: If you only care about the text output, reduce gamma to 0.1 to give the student freedom. A small model can find more efficient ways to reach the correct answer, as long as the final result is equivalent and correct. This approach is useful when there's a large size difference between teacher and student.

With our loss function defined and calibrated, we're ready to integrate it into the training loop.

6.3.2 Training loop and strategy comparison

Now you’re ready to implement the training loop that orchestrates the entire recovery process using the composite loss function we just created.

This loop synchronizes the forward pass of two models simultaneously. In each iteration, the teacher model generates its predictions and internal representations, while the student model does the same. Our loss function

compares both outputs, calculates the error, and adjusts the student's weights to bring its behaviour closer to the teacher's.

CREATING THE TRAINING LOOP

The `train_student` function we're going to build encapsulates this complete cycle. It manages the optimizer, controls the data flow through both models, calls `compute_compound_loss` (our custom loss function) for each batch, and runs backpropagation to update only the student's parameters. Let's see how the `train_student` function is implemented.

Listing 6.7 The training loop

```
def train_student(
    student_model, teacher_model,dataloader,
    layer_map,alpha=0.4,beta=0.4,gamma=0.2,temperature=2.0,
    epochs=3,learning_rate=1e-5,experiment_name="experiment",accumul
ation_steps=4
):
    optimizer = torch.optim.AdamW(
        student_model.parameters(),
        lr=learning_rate
    ) #1

    student_model.train() #A
    teacher_model.eval()

    request_hidden_states = (gamma > 0) #B

    loss_history = {'total': [], 'task': [], 'logits': [], 'hidden':
[]}
    epoch_times = []
    total_start_time = time.time()

    for epoch in range(epochs):
        epoch_start_time = time.time()
        epoch_losses = {'total': [], 'task': [], 'logits': [], 'hidd
en': []}
        progress_bar = tqdm(dataloader, desc=f"Epoch {epoch+1}/{epoc
hs}")

        accumulated_losses = {'total': 0.0, 'task': 0.0, 'logits':
0.0, 'hidden': 0.0}
        accumulation_counter = 0

        for batch_idx, (input_ids, attention_mask) in enumerate(prog
ress_bar):
            input_ids = input_ids.to(device)
            attention_mask = attention_mask.to(device)
            labels = input_ids.clone()

            student_outputs = student_model(
                input_ids=input_ids,
                attention_mask=attention_mask,
                output_hidden_states=request_hidden_states #2
```

```
            ) #C

            with torch.no_grad(): #3
                teacher_outputs = teacher_model(
                    input_ids=input_ids,
                    attention_mask=attention_mask,
                    output_hidden_states=request_hidden_states
                ) #D

            student_hiddens = (
                student_outputs.hidden_states[1:]
                if request_hidden_states else None
            ) #4
            teacher_hiddens = (teacher_outputs.hidden_states[1:]
                               if request_hidden_states else None)

            loss, loss_dict = compute_compound_loss(
                student_logits=student_outputs.logits,
                teacher_logits=teacher_outputs.logits,
                student_hiddens=student_hiddens,
                teacher_hiddens=teacher_hiddens,
                labels=labels,
                layer_map=layer_map,
                alpha=alpha, beta=beta, gamma=gamma, temperature=tem
perature
            )

            scaled_loss = loss / accumulation_steps #5
            scaled_loss.backward()

            for key in accumulated_losses:
                accumulated_losses[key] += loss_dict[key]
            accumulation_counter += 1

            if (batch_idx + 1) % accumulation_steps == 0: #6
                torch.nn.utils.clip_grad_norm_(student_model.paramet
ers(), max_norm=1.0)
                optimizer.step()
                optimizer.zero_grad()

                avg_losses = {k: v / accumulation_counter for k, v i
n accumulated_losses.items()}
                for key in avg_losses:
```

```
                    epoch_losses[key].append(avg_losses[key])

                progress_bar.set_postfix({
                    'loss': f"{avg_losses['total']:.4f}",
                    'task': f"{avg_losses['task']:.4f}",
                    'logits': f"{avg_losses['logits']:.4f}",
                    'hidden': f"{avg_losses['hidden']:.4f}"
                })

                accumulated_losses = {'total': 0.0, 'task': 0.0, 'lo
gits': 0.0, 'hidden': 0.0}
                accumulation_counter = 0

        if accumulation_counter > 0: #7
            torch.nn.utils.clip_grad_norm_(student_model.parameters
(), max_norm=1.0)
            optimizer.step()
            optimizer.zero_grad()

            avg_losses = {k: v / accumulation_counter for k, v in ac
cumulated_losses.items()}
            for key in avg_losses:
                epoch_losses[key].append(avg_losses[key])

        for key in epoch_losses:
            if epoch_losses[key]:
                avg_loss = np.mean(epoch_losses[key])
                loss_history[key].append(avg_loss)

        epoch_time = time.time() - epoch_start_time
        epoch_times.append(epoch_time)

    total_time = time.time() - total_start_time

    loss_history['epoch_times_seconds'] = epoch_times
    loss_history['total_time_seconds'] = total_time

    return student_model, loss_history
```

#A Set the teacher model to evaluation mode
#B Only calculate hidden states if required
#C Forward pass of the student model
#D Forward pass of the teacher model

This function encapsulates the entire training cycle. The first step is to configure the AdamW optimizer #1, responsible for adjusting the student model weights. It takes the error measured by our compound loss function and translates it into concrete parameter updates that gradually bring the student closer to the teacher's behavior.

It executes synchronized forward passes on both models. Through the `output_hidden_states=True` parameter #2, we tell the Transformers API to return both the representations from each transformer block and the logits. Immediately after, the `torch.no_grad()` block wraps the teacher's forward pass #3, disabling gradient calculation for that model. This reduces memory consumption by approximately half and speeds up the process, since we only need the teacher's outputs for comparison, not for training.

After obtaining the outputs, the function extracts the hidden states by discarding the first element #4 through the `[1:]` slice. The `.hidden_states` method returns a tuple where position 0 contains the output from the embedding layer and the following positions contain the output from each transformer block. Since our goal is to align the transformer block representations between teacher and student, we discard that first embedding tensor, which doesn't correspond to any block, and work only with the outputs from the actual blocks.

The function implements gradient accumulation to simulate large batches without exploding GPU memory. Instead of updating weights after each batch, we accumulate gradients over several iterations, and only then run the optimizer. If we work with batches of 16 rows and accumulate during four iterations #5 #6, the optimizer receives information equivalent to having processed 64 rows (16 × 4), but the GPU only needs memory for 16. Dividing the loss by

`accumulation_steps` #5 ensures the average is correct. This pattern is fundamental for training language models on limited hardware, allowing us to use effective batch sizes that would otherwise require GPUs with more memory.

Finally, if the total number of batches isn't divisible by `accumulation_steps`, a conditional block processes the last incomplete group #7, avoiding the loss of training data. The function returns both the trained model and a `loss_history` dictionary with the evolution of each loss component, metrics that will be fundamental for diagnosing the recovery process in our experiments.

RESULTS ANALYSIS

With this function, we have everything to start the Knowledge Distillation process and we can continue analyzing the experiments we ran. We'd already selected the model using 2K quick tests; now, let's examine the impact of using hidden states versus not using them and the different layer mapping options.

Figure 6.6 shows the convergence patterns across these three data scales. The bars show perplexity evolution (lower is better), while the overlaid lines track capabilities retention (higher is better). Each strategy — Labels-Only, Features (Last), and Features (Selected) — follows a similar trajectory: dramatic improvement from 2K to 15K samples, followed by more gradual gains toward 40K. The steep drop in perplexity bars between 2K and 15K (from ~18-19 down to ~12-13) demonstrates how knowledge recovery is highly data-dependent in its initial phase.

Note that Features (Last) is absent from the 40K scale. At 15K, both strategies produced equivalent results with no meaningful differential, so the experiment continued with

Selected mapping only. This reflects a practical decision you'll face in real optimization projects: once two strategies converge, redirecting compute toward the more promising variables is the right call.

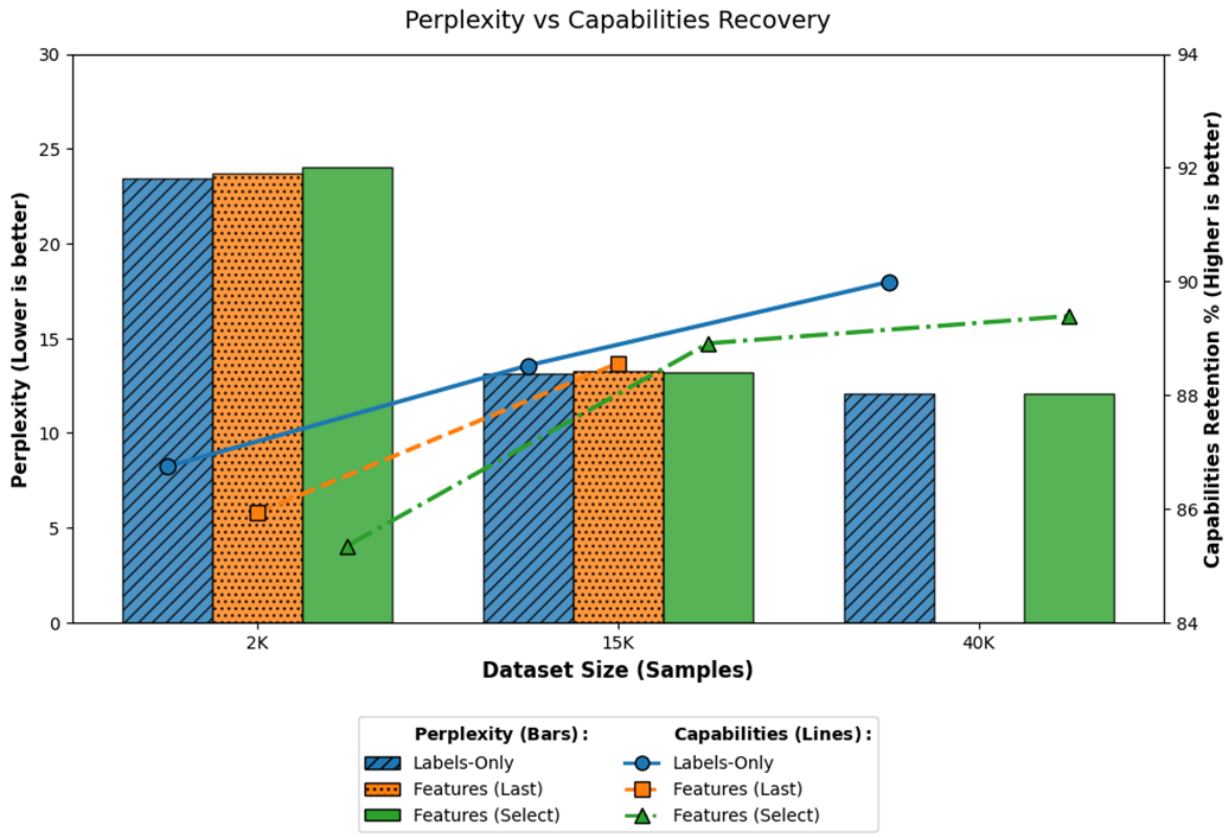


Figure 6.6 Convergence analysis showing perplexity (bars, left axis) and capabilities retention (lines with markers, right axis) across three training data scales. Different bar patterns (diagonal hatching, dots, solid) and line markers (circles, squares, triangles) distinguish the three distillation approaches tested at each scale.

We trained the student model with three data configurations: 2,000 samples from the Cosmopedia dataset, 15,000 samples, and 40,000 samples (extensive training). Using 5 epochs for the 2K case, and 3 epochs for 15K and 40K.

Table 6.3 shows three critical performance aspects: perplexity (how well the student predicts text sequences), capability retention (average performance on benchmarks

like ARC-Easy, HellaSwag, PIQA and Winogrande), and specifically LAMBADA (which measures long narrative context understanding, the capability most affected by depth pruning).

Table 6.3 Comparing Knowledge Distillation strategies

Experiment	Techniques	Map	Perplexity 2K / 15K / 40K	Capabilities Retention 2K / 15K / 40K	LAMBADA 2K / 15K / 40K
Baseline	Teacher	—	13.38 / 12.96 / 12.91	100 / 100 / 100	0.43 / 0.43 / 0.43
Baseline	Pruned	—	125.60 / 121.84 / 120.67	78.6 / 78.6 / 78.30	0.177 / 0.177 / 0.177
Labels-Only	Soft & Hard Labels KD	—	18.86 / 12.65 / 11.73	86.74 / 90.1 / 90.1	0.295 / 0.332 / 0.335
Features	Feature Alignment	Last Layer	18.85 / 12.75 / —	86.3 / 89.6 / —	0.293 / 0.330 / —
Advanced KD	Feature Alignment + FDD + SKD	Selected Layer	18.94 / 12.54 / 11.54	86 / 89.2 / 90	0.294 / 0.320 / 0.331
Advanced KD	Feature Alignment + FDD + SKD	Uniform	18.96 / 12.56 / —	86.3 / 89.5 / —	0.295 / 0.327 / —

NOTE

Capabilities Retention % measures average performance on benchmarks (arc_easy, hellaswag, piqa, winogrande) compared to the teacher. LAMBADA specifically measures long narrative context understanding, the most degraded ability after depth pruning (59% drop). The 2K trainings used 5 epochs; 15K and 40K used 3 epochs.

The first conclusion from observing the table is that the amount of data is crucial for recovering the pruned model. With 2,000 samples, all experiments achieve perplexities in the 18-19 range, far from the teacher (13.4). With 15,000 samples, logits distillation reaches 12.65 perplexity, a significant improvement that actually surpasses the teacher at that evaluation point (12.96).

With 40,000 samples, it reaches 11.73, even slightly surpassing the original teacher (12.91). This progression ($18.86 \rightarrow 12.65 \rightarrow 11.73$) shows that basic hard and soft labels distillation is extraordinarily effective when provided with enough training material, and that its effectiveness continues improving with more data.

Second, adding hidden state alignment with FDD and Skew KLD produces a consistent but nuanced pattern across training sizes. With 2K samples, features alignment is slightly worse on both dimensions: PPL 18.94 vs 18.86 for labels-only, and capability retention 86% vs 86.74%. With 15K, the picture shifts: Features (Selected) edges ahead in perplexity (12.54 vs 12.65), but loses slightly in capability retention (89.2% vs 90.1%). With 40K, that same pattern holds: Features (Selected) reaches 11.54 vs 11.73 for labels-only, again with a marginal capability cost (90% vs 90.1%). The additional computational cost is real, approximately 15% more training time, but the tradeoff is predictable: if perplexity is your primary optimization target and you have sufficient data, advanced KD provides a reliable, if modest, advantage. If capability retention matters equally, labels-only remains competitive and cheaper.

In contexts where hidden states are the final product, like embedding models for RAG or classifiers that directly consume vector representations (discussed in section 6.3.1), feature alignment is a valid strategy, as you're directly

optimizing the output that the downstream system will consume. It's also a widely used technique with multimodal or vision models because in these models, intermediate representations (hidden states) usually contain crucial information for the final task, such as object detection, image recognition, or fusion of information from multiple modalities.

These results show that logit distillation can effectively recover the model's knowledge, as long as we have enough data. However, research in Knowledge Distillation has developed more sophisticated techniques.

6.4 Advanced recovery techniques (FDD/ Skew KLD)

We can achieve very strong recovery results, matching the base model's perplexity, or even surpassing it, while maintaining between 78% and 90% of the model's capabilities depending on the number of rows we used for its recovery. But these numbers can improve, or even better, be matched using a lighter, and thus less costly, recovery process.

To achieve these results, we'll implement Feature Dynamics Distillation (FDD) and Skew KL Divergence.

Skew KL Divergence modifies the loss function used to compare the teacher and student logits. When the student model, after aggressive pruning, produces probability distributions very different from the teacher's, standard KL Divergence doesn't penalize those differences optimally. Skew KLD corrects this behavior, allowing the student to learn more efficiently when starting from a configuration far from the teacher.

Feature Dynamics Distillation (FDD) captures how internal representations evolve as information passes from one block to the next.

We can see the results of this approach by analyzing the results of our experiments with Gemma-3-270m. Both conventional logit distillation and hidden states show solid recovery. The pattern that emerges across training sizes is consistent: advanced KD provides a perplexity advantage when enough data is available, at a small capability cost:

1. With 2,000 samples: advanced KD reaches 18.94, marginally worse than labels-only (18.86), with a small capability loss (86% vs 86.74%). At this scale, the technique provides no advantage.
2. With 15,000 samples: advanced KD improves perplexity to 12.54 versus 12.65 for labels-only, while losing slightly in capability retention (89.2% vs 90.1%).
3. With 40,000 samples: advanced KD reaches 11.54 versus 11.73 for labels-only, a gap of 0.19 points, roughly double the improvement seen at 15K. Capability retention remains essentially identical (90% vs 90.1%).

These results suggest that FDD and Skew KLD become increasingly valuable as training data grows. With limited data, basic logit distillation is equally effective and computationally cheaper. But as your dataset scales, advanced KD delivers a progressively larger perplexity advantage at a minimal capability cost.

NOTE

The compound function that contains these two additions is called `compute_compound_loss_advanced` and the code can be found in the experiment notebooks and in the notebook CH06_NB01_Knowledge_Recovery_T4.ipynb.

Let's start by understanding Skew KL Divergence and why standard KL divergence can lead us to suboptimal results when the student and teacher are far apart.

6.4.1 Skew KLD: correcting distribution bias

Calculating Logits Loss (using KL Divergence) to measure the difference between the teacher and student probability distributions has worked well so far, but it has an important limitation that shows up when the student and teacher have very different distributions, such as after a very aggressive pruning process.

The problem comes from how KL Divergence penalizes discrepancies. Imagine the teacher assigns a 70% probability to "France" as the next token after "Paris is the capital of", while the newly pruned student, in its initial state, only assigns 30%, simply because it has lost part of the knowledge it has about France. This creates a very steep gradient that pushes the student to imitate the teacher's confidence prematurely.

Skew KLD introduces a simple but powerful conceptual change: instead of comparing the teacher's prediction directly with the student's, we compare the teacher with a mixture of both. This mixture is controlled by a new parameter `skew_alpha`. With `skew_alpha=0.1` (the value we'll use), the target is a mixture of 90% teacher and 10%

student. This makes the gradient smoother at the beginning of training, when the student is very far off, and it becomes stricter as the student converges, because the 10% of the student's distribution in the mixture increasingly resembles the teacher. Figure 6.7 shows this difference concretely with the France/Europe/Other example.

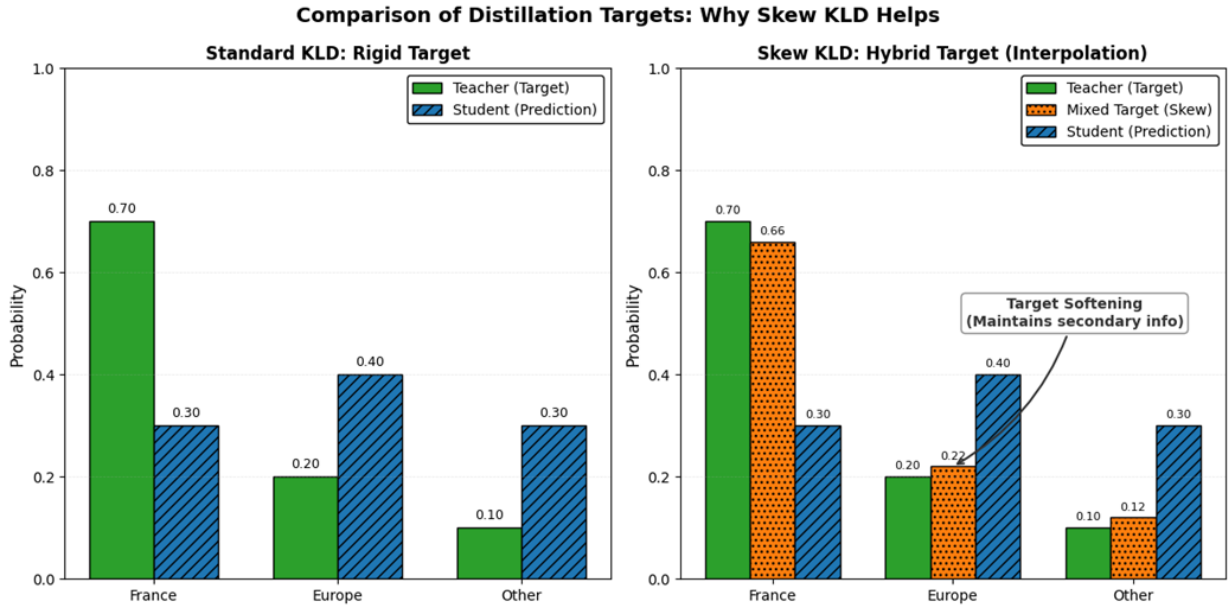


Figure 6.7 Target distribution comparison. Left panel: Standard KLD with teacher targets (solid) and student predictions (diagonal hatching). Right panel: Skew KLD adds hybrid targets (dotted) computed as 90% teacher + 10% student.

Let’s see how this translates into code. The standard implementation of KL Divergence that you’ve used so far in the `compute_compound_loss` function is:

```
student_soft = F.log_softmax(student_logits / temperature, dim=-1)
teacher_soft = F.softmax(teacher_logits / temperature, dim=-1)

    student_soft = student_soft[..., :-1, :].contiguous()
    teacher_soft = teacher_soft[..., :-1, :].contiguous()

    loss_logits = F.kl_div(
        student_soft.view(-1, student_soft.size(-1)),
        teacher_soft.view(-1, teacher_soft.size(-1)),
        reduction='batchmean'
    ) * (temperature ** 2)
```

The new implementation with Skew KLD introduces an additional step before the calculation:

```
with torch.no_grad():
    student_probs = F.softmax(student_logits[..., :-1, :] / temperat
ure, dim=-1)
    teacher_probs = F.softmax(teacher_logits[..., :-1, :] / temperat
ure, dim=-1)

    mixed_probs = skew_alpha * student_probs +
➥(1 - skew_alpha) * teacher_probs #A

student_log_probs = F.log_softmax(student_logits[..., :-1, :] / temp
erature, dim=-1)
kl_elementwise = student_probs * (student_log_probs -
➥torch.log(mixed_probs + 1e-9)) #B
loss_logits = kl_elementwise.sum(dim=-1).mean() * (temperature ** 2)
```

#A Mixes 10% student predictions with 90% teacher predictions
#B Computes KL divergence using mixed distribution as reference

Instead of directly comparing `student_probs` with `teacher_probs`, we compare against `mixed_probs`. At the beginning of training, when the student generates distributions very different from the teacher, that 10% of "its own voice" in the mix softens the penalty. As the student improves, that 10% converges toward the teacher, so the effective target automatically hardens.

The `1e-9` term is a numerical stability constant to avoid calculating the logarithm of zero when `mixed_probs` has extremely small values.

This technique is particularly valuable in two scenarios. First, when you have applied aggressive pruning. For a small model like Gemma-3-270M, removing more than 20% of the Transformer blocks can be considered aggressive, as the student model starts from a very degraded state. Second, when your distillation dataset is limited (fewer than 10,000 samples), because the smoothed gradients allow the model to learn more efficiently with each example.

If your dataset is large (30K+ samples) and pruning was moderate, standard KL is probably enough and simpler to implement. As your data budget grows beyond 15K samples, Skew KLD begins to show a consistent advantage, smoothing gradients in a way that standard KL cannot. The larger the dataset, the more that difference compounds.

6.4.2 Feature Dynamics Distillation

Until now we've aligned the student's intermediate representations with the teacher's layer by layer: "Does your layer 5 look like my layer 9?". But this strategy doesn't capture the rate at which those representations change.

Feature Dynamics Distillation (FDD) solves this by capturing the difference between consecutive layers. In the case where the student's layer 5 is mapped to the teacher's layer 9, instead of just comparing hidden[5] with hidden_teacher[9], we calculate the deltas for both and align them using cosine similarity.

Let's look at the difference in the code. This is the Feature Alignment implementation, in the `compute_compound_loss`

function you've used in section 6.3.

```
for student_idx, teacher_idx in enumerate(layer_map):
   student_h = student_hiddens[student_idx] #A
   teacher_h = teacher_hiddens[teacher_idx]

   student_flat = student_h.reshape(-1, student_h.size(-1)) #B
   teacher_flat = teacher_h.reshape(-1, teacher_h.size(-1))

   student_norm = F.normalize(student_flat, p=2, dim=1) #C
   teacher_norm = F.normalize(teacher_flat, p=2, dim=1)

   cos_sim = (student_norm * teacher_norm).sum(dim=1).mean() #D
   loss_hidden += (1 - cos_sim) #E
```

#A Gets corresponding hidden states from student and teacher layers
#B Reshapes to treat each token independently: [batch×seq_len, hidden_dim]
#C Normalizes vectors to unit length
#D Computes cosine similarity via dot product
#E Loss = 1 - similarity (ranges from 0 for identical to 2 for opposite)

The new implementation with FDD, which you can find in `compute_compount_loss_advanced`, introduces the calculation of the differentials (deltas) before normalization:

```
for student_idx in range(len(layer_map) - 1): #A
    teacher_idx = layer_map[student_idx]
    teacher_idx_next = layer_map[student_idx + 1]

    student_delta = student_hiddens[student_idx + 1] - #B
➥student_hiddens[student_idx]
    teacher_delta = teacher_hiddens[teacher_idx_next] -
➥teacher_hiddens[teacher_idx]

    student_delta_flat =
➥student_delta.reshape(-1, student_delta.size(-1)) #C
    teacher_delta_flat = teacher_delta.reshape(-1, teacher_delta.siz
e(-1))

    student_delta_norm =
➥F.normalize(student_delta_flat, p=2, dim=1) #D
    teacher_delta_norm = F.normalize(teacher_delta_flat, p=2, dim=1)

    cos_sim_deriv = (student_delta_norm *
➥teacher_delta_norm).sum(dim=1).mean() #E
    loss_derivative += (1 - cos_sim_deriv)
```

#A Iterates to N-1 since we compare consecutive layer pairs
#B Computes change vector between consecutive layers
#C Flattens deltas for vectorized computation
#D Normalizes transformations to unit length
#E Measures alignment of transformation directions

The key difference is in calculating `student_delta = student_hiddens[student_idx + 1] - student_hiddens[student_idx]`. We compare transformation vectors instead of absolute states. We use cosine similarity because, just like in the previous section, we care about the direction of change, not its absolute magnitude.

6.4.3 Results: when advanced techniques make a difference

With both techniques implemented, it's time to see whether the added complexity pays off. Table 6.4.consolidates all

experiments. Pay particular attention to the 2K column, where the difference is more revealing.

Table 6.4 Evaluating Knowledge Distillation Strategies

Experiment	Techniques	Map	Perplexity 2K / 15K / 40K	Capabilities Retention 2K / 15K / 40K	LAMBADA 2K / 15K / 40K
Baseline	Teacher	—	13.38 / 12.96 / 12.91	100 / 100 / 100	0.43 / 0.43 / 0.43
Baseline	Pruned	—	125.60 / 121.84 / 120.67	78.6 / 78.30 / 78.30	0.177 / 0.177 / 0.177
Labels-Only	Hard & Soft Labels KD	—	18.86 / 12.65 / 11.73	86.74 / 90.1 / 90.1	0.295 / 0.332 / 0.335
Features	Feature Alignment	Last Layer	18.85 / 12.75 / —	86.3 / 89.6 / —	0.293 / 0.330 / —
Features	Feature Alignment	Selected Layer	18.84 / 12.66 / 11.74	86.4 / 89.6 / 90.3	0.294 / 0.329 / 0.334
Features	Feature Alignment	Uniform	18.85 / — / —	86.5 / — / —	0.295 / — / —
Advanced KD	Feature + FDD + KLD	Selected	18.94 / 12.54 / 11.54	86.0 / 89.2 / 90.0	0.294 / 0.320 / 0.331
Advanced KD	Feature + FDD + KLD	Uniform	18.96 / 12.56 / —	86.3 / 89.5 / —	0.295 / 0.327 / —

The results reveals fundamental patterns regarding knowledge recovery:

With 2K samples, all techniques produce similar results. Advanced KD reaches perplexities of 18.94-18.96, essentially tied with labels-only (18.86) and feature alignment (18.84-18.85). At this scale, the additional complexity of FDD and Skew KLD provides no measurable advantage, and capability retention is virtually identical across all methods (~86%).

With 15K samples, all techniques converge toward the teacher. Perplexity drops dramatically: Labels-Only reaches 12.65, Feature Alignment 12.66-12.75, and Advanced KD 12.54-12.56. At this point, the volume of data allows even simpler techniques to effectively recover the knowledge. Advanced KD shows a modest advantage of approximately 1% in perplexity.

With 40K samples, we recover and surpass the teacher in perplexity. Selected Advanced KD reaches 11.54, better than the teacher (12.91), with capability retention of 90%, essentially identical to labels-only (90.1%). The perplexity advantage of Advanced KD over labels-only grows to 0.19 points, roughly double the gap seen at 15K.

The LAMBADA benchmark shows a consistent pattern: all methods recover from the catastrophic 0.177 of the pruned model up to approximately 0.320-0.335, but none reaches the teacher's 0.43. This suggests that long-context language modeling is one of the capabilities most affected by layer removal and hardest to fully recover.

FDD and Skew KLD techniques prove most valuable when sufficient training data is available. With limited data, basic logit distillation is equally effective and computationally cheaper. As your dataset scales, advanced KD delivers a progressively larger perplexity advantage at a minimal capability cost.

6.5 Practical guidelines for knowledge recovery

After exploring the foundations of knowledge distillation and analyzing the results of multiple experiments with different techniques, you're probably wondering how to apply all this knowledge to your specific use case. As you’ve likely already

guessed, there is no magic formula that works for every situation. You have to analyze, run experiments like the ones in this chapter, and over time you'll end up developing an intuition. It depends not only on the data and the task you're preparing the LLM for, but also on its shape and how its original training was conducted.

Even so, we can condense all the knowledge acquired during this chapter into practical decisions you can apply to your projects.

6.5.1 Composite loss configuration

The first decision to make is whether you want to use feature alignment. With `gamma=0` you completely deactivate this loss and reduce computational cost. Our experiments show that for pure generative models, `gamma=0.2` or even `gamma=0` produces equivalent or better results. Reserve higher gamma values (0.3-0.4) for cases where intermediate representations are the final product: embedding models for RAG, classifiers that consume hidden states directly, or multimodal models where information fusion happens in intermediate layers.

For the rest, you need to adjust the balance between alpha (hard labels) and beta (soft labels). Both operate on the model's final output, but with different objectives: alpha forces correct predictions according to the dataset, while beta transfers the teacher's confidence nuances. For general-purpose models (say you've decided to use 0.2 for gamma), the 0.4/0.4 balance works well. If your distillation dataset is very high quality and perfectly aligned with your final task, you can increase alpha to 0.5 and reduce beta to 0.3. On the other hand, if your dataset is noisy or generic, prioritize beta so the student learns mainly from the teacher's reasoning.

Temperature softens the teacher's probability distributions. Values close to 2.0 work well in most scenarios. Use higher temperatures (2.5-3.5) when the teacher has very concentrated distributions, with over 90% probability on a single token, and you want the student to learn from the alternatives. Lower temperatures (1.0-2.0) are appropriate when the teacher already produces smooth distributions.

6.5.2 When to use advanced techniques

The `compute_compound_loss_advanced` function incorporates two additional techniques: Skew KLD and FDD. Skew KLD is integrated into the logits loss and controlled via `skew_alpha`: with the recommended value of 0.1, you get a 90% teacher, 10% student mix that smooths gradients when the model starts from a heavily degraded state. To disable it, use `skew_alpha=0`. FDD is activated through a fourth parameter, `delta`. With `delta=0`, the technique is disabled; with `delta=0.1`, you activate aligning dynamics between consecutive layers. When using FDD, remember to adjust the other parameters to keep the sum at 1.0: for example, with `delta=0.1` and `gamma=0.2`, you can use `alpha=0.35` and `beta=0.35`.

Both techniques show increasing value as training data grows. With small datasets of fewer than 10,000 samples, they provide no measurable advantage over basic distillation, so you can simplify the process using `delta=0` and `skew_alpha=0`. With larger datasets of 15,000 or more samples, Advanced KD begins to show a consistent perplexity advantage, which grows further beyond 30,000 samples. If perplexity is your primary optimization target and you have sufficient data, activating both techniques is a reasonable choice at a modest additional computational cost.

6.5.3 Combining with width pruning

Finally, everything you've learned in this chapter works just as well if your student model has been previously subjected to width pruning. Width pruning modifies the `intermediate_size` of the MLP modules, the internal dimension of the `gate_proj`, `up_proj`, and, `down_proj` layers, but the `hidden_dim` of the Transformer block remains intact.

This means that the input and output representations of each block keep exactly the same dimensions, and the alignment between teacher and student works without any changes. The notebook CH06_NB02_Width_Pruned_Model_Recovery.ipynb demonstrates this compatibility, validating that you can first apply width pruning to reduce the computational cost of the MLPs, and then depth pruning followed by Knowledge Distillation to remove entire blocks. The pipeline is completely combinable, allowing you to stack these techniques to create a highly efficient custom model, as shown in figure 6.8.

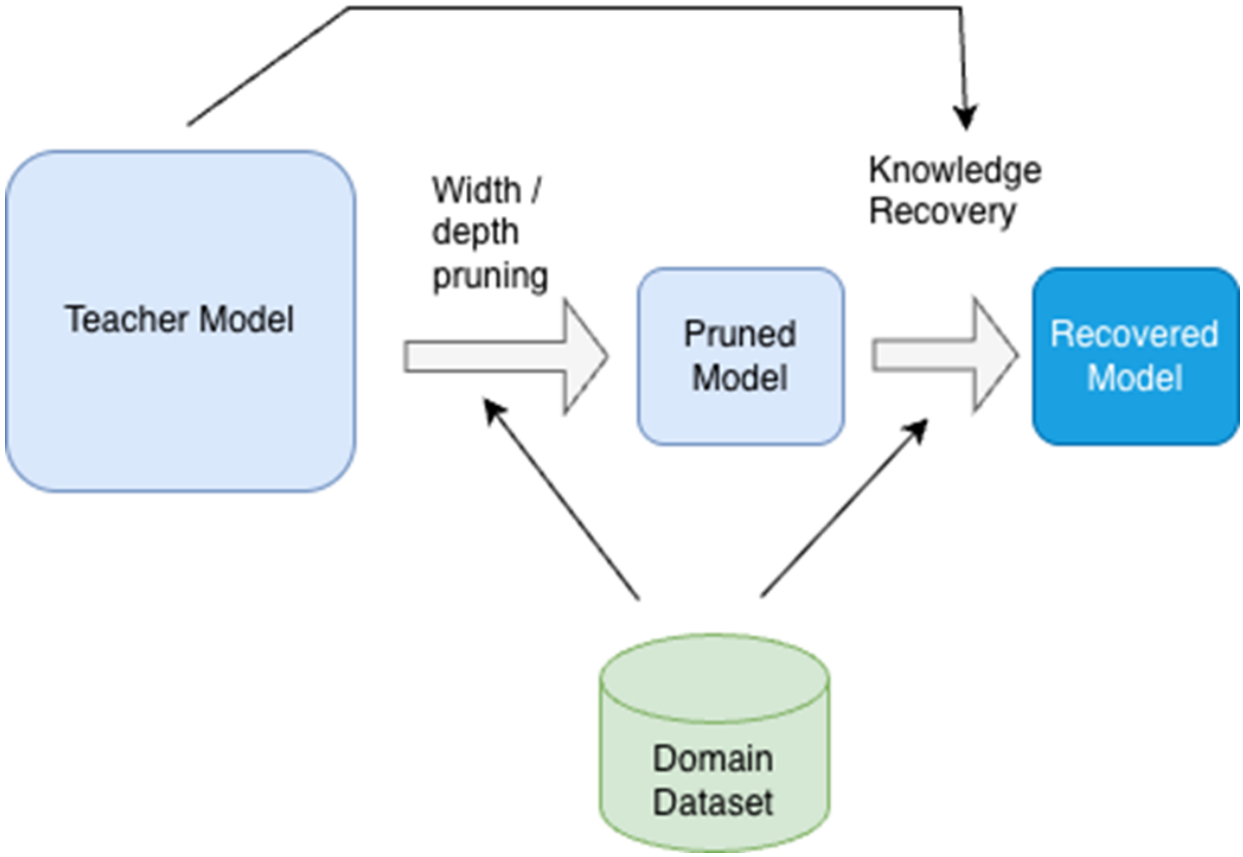


Figure 6.8 The rearchitecting pipeline is flexible and modular. You can apply Width Pruning to reduce MLP dimensions, Depth Pruning to remove blocks, or combine both as shown here for maximum efficiency. Regardless of the structural changes chosen, the final step uses the original Base Model as the Teacher to recover lost knowledge through Distillation.

With these configurations in mind, Table 6.5 serves as a quick reference when you are looking at the code and deciding which values to use.

Table 6.5 Recommended Knowledge Distillation Configurations

Scenario	alpha	beta	gamma	Skew KLD (skew_alpha)	FDD (delta)	Notes
Generalist (>30K)	0.4	0.4	0.0-0.2	0.1	0.1	Advanced KD delivers growing perplexity advantage at scale
Generalist (15K-30K)	0.4	0.4	0.0-0.2	0.1	0	Skew KLD begins to show consistent advantage; FDD optional
Generalist, limited data (<10K)	0.4	0.4	0.2	0	0	Basic distillation is equally effective and cheape
Aggressive pruning (>20% blocks)	0.25	0.45	0.2	0.1 - 0.15	0.1	Prioritize imitating the teacher
Embeddings/RAG model	0.25	0.25	0.4	0.1	0.1-0.2	The hidden states are the final product
High-quality dataset, specific task	0.5	0.3	0.1	0	0.1	Trust the dataset labels more

These values are a starting point, not fixed rules. Each model and dataset has its particularities, and part of the optimization work is adjusting these parameters based on the results you observe during validation. The important thing is that now you have a complete and flexible framework: you can combine depth pruning with width pruning, activate or deactivate advanced techniques according to your resources, and adapt the balance of the compound loss to your final objective.

6.6 From paper to practice

Throughout this chapter, you've built a complete knowledge recovery pipeline. You've pruned a model, calculated distillation losses, and retrained the weights to recover the lost accuracy. It's natural to wonder: Is this an academic exercise or is this actually how production models are built?

The answer is found in two recent papers that have defined the state of the art in efficient models: Minitron (2024,https://arxiv.org/abs/2408.11796) from Nvidia and Ministral 3 (2026,https://arxiv.org/abs/2601.08584) from Mistral AI. Both confirm that the path you've followed, structural pruning followed by distillation, is the current industry standard for creating high-performance SLMs.

The paper "LLM Pruning and Distillation in Practice: The Minitron Approach" is the main inspiration for the pipeline presented in chapter one, although it has been adapted for creating SLMs in environments with more restrictions than those available to Nvidia.

The experiments in this chapter converge with the fundamental principles for effective distillation outlined in both papers.

The standard distillation setup, hard and soft labels, combines two complementary objectives: task loss (cross-entropy with hard labels from the dataset) and logits loss (KL divergence with soft labels from the teacher). This combination, while conceptually simple, is surprisingly robust.

Both papers compare knowledge distillation against simply continuing to train the pruned model with standard fine-tuning. Nvidia tested both approaches directly and found

that distillation-based retraining consistently outperforms standard fine-tuning. Mistral reached the same conclusion, finding that training with the pure distillation objective outperformed any hybrid combination of distillation and next-token prediction loss.

The papers validate this approach at a large scale. Mistral mentions that using only the logits distillation objective outperforms any weighted combination, suggesting that task loss might be optional. In our composite loss function, the exact proportion between alpha (hard labels) and beta (soft labels) can be adjusted depending on the case. Typical values range between (0.3-0.5) for alpha and (0.5-0.7) for beta. In the experiments, hard labels are not discarded to achieve an earlier confluence with the teacher model's results. The logic is that when training the model for a specific task, we want to prioritize the results obtained in that task; in our case, logits are used to stabilize learning and help the model generalize better.

6.6.1 Layer selection and alignment

Before distilling, we face a critical decision: which layers to remove from the model. The most straightforward approach is removing the last *N* layers, based on the hypothesis that final layers contain specific refinements while early layers capture more fundamental patterns. However, a data-driven approach, measuring the actual contribution of each layer through activation analysis, can better identify candidates for removal.

After layer removal, the mapping problem arises: how do we align the student's layers with the teacher's during distillation? The Nvidia paper explores three strategies: uniform (proportional mapping), last (all student layers learn from the teacher's final layers), and original (preserving

specific layers). For moderate compression ratios (<50% of removed layers), uniform mapping works well. The "last" strategy becomes relevant only in aggressive depth pruning where you remove more than half of the model's layers.

NOTE

The difference between what Nvidia considers aggressive pruning (+50%) and our approach (+20%) is due to the scale of the models used. The larger the model, the better it handles depth pruning.

The effectiveness of feature alignment depends on the scenario. In models with moderate depth pruning and abundant data, the improvement can be marginal: most of the knowledge is already transferred via hard or soft labels. On the other hand, in scenarios with aggressive compression or limited data, that additional signal helps the student "navigate" the teacher's representation space.

6.6.2 Advanced techniques: when data is scarce

Both Nvidia and Mistral work with massive amounts of data, beyond the reach of most companies. When the compression ratio is high, or when you work in specialized domains, the techniques we explored in sections 6.4.1 (Skew KLD) and 6.4.2 (Feature Dynamics Distillation) offer measurable advantages. What they do is maximize the use of each available sample, helping both models converge more efficiently.

IMPORTANT

Advanced distillation techniques such as Skew KLD or Feature Dynamics Distillation don't replace the need for quality data or a solid pruning strategy. If your dataset is noisy or your block selection was poor, no loss function will fully compensate for that.

The papers show what's possible with enterprise-scale resources. This chapter has shown you how to apply the same principles when those resources aren't available, which is the reality for most of us building specialized and efficient models.

You're using techniques validated by leading LLM research labs, at least the ones that publish their research.

6.7 Hands-on lab

The practical impact of knowledge distillation depends on countless decisions: which blocks to prune, how to balance the loss components, which mapping strategy to use, and how much data is truly necessary. This section provides structured experiments designed to build your intuition about these tradeoffs.

The primary challenge is to beat the baseline model we've published at `oopere/gemma-3-270m-14L-distilled`. This model was created using the `CH06_NB03_Hands_on.ipynb` notebook with a configuration that differs from the chapter experiments: a stronger task loss weight (`alpha=0.6`, `beta=0.4`), a more aggressive Skew KLD interpolation (`skew_alpha=0.4`), and a higher learning rate (`4e-5`). Trained on 40K Cosmopedia samples for 5 epochs with hard and soft labels only (no

hidden state alignment), it achieved 89.5% capability retention and 10.85 perplexity. Your mission: surpass it using the techniques from this chapter.

NOTE

The baseline model training (`CH06_NB03_Hands_on.ipynb`) was executed on an NVIDIA A100 GPU and took approximately 68 minutes with 40K samples. If you don't have access to this hardware, you can still participate in the challenge by experimenting with the configuration parameters (loss weights, mapping strategies, block selection) and sharing your optimization ideas in the book's discussion forum or in the GitHub repository discussions.

- The main challenge (beat the baseline):
  - What to do: Using the `CH06_NB03_Hands_on.ipynb` notebook, create a 14-block student model that surpasses the baseline (`oopere/gemma-3-270m-14L-distilled`) in at least 1 out of 5 benchmarks (arc_easy, winogrande, hellaswag, lambada_openai, piqa), and in overall recovery. You can modify any aspect of the pipeline: block selection strategy, loss function weights (alpha, beta, gamma), mapping strategy, dataset composition, or training hyperparameters. The notebook automatically generates a model card for you. Once you're satisfied with your results, make it public on the Hugging Face Hub.
  - Questions to answer: Which benchmark did you improve the most, and why do you think your configuration worked better for that specific capability? Did improving one benchmark cause

degradation in another? What was your total training time compared to the baseline (68 minutes on A100)? Which single change had the biggest impact on your results?

- The data efficiency frontier:
  - What to do: The baseline model used 40,000 Cosmopedia samples. Your constraint: achieve comparable results (within 2% capability retention) using only 15,000 samples. Experiment with advanced techniques from section 6.4: activate FDD (`delta=0.1-0.2`) and Skew KLD (`skew_alpha=0.1`) to maximize learning from limited data. Try different combinations of alpha, beta, gamma, and delta weights to find the optimal configuration for your reduced dataset.
  - Questions to answer: How much did advanced techniques accelerate convergence compared to basic logits-only distillation with the same data budget? Did the final perplexity improve, stay the same, or degrade? What's the minimum sample count where you can still recover >85% of capabilities?
- Combining width and depth pruning (advanced):
  - What to do: Open the `CH06_NB02_Width_Pruned_Model_Recovery.ipynb` notebook, which already applies width pruning to reduce the MLP intermediate size. Your challenge: before the knowledge distillation process begins, add depth pruning using `optipfair.prune_model` to remove two additional blocks from the width-pruned model. You'll need to decide which two blocks to remove (use the layer importance analysis you learned in section 6.1) and then apply the compound loss distillation to this doubly-pruned architecture.

    - Questions to answer: What's the total parameter reduction when combining both techniques compared to depth-only or width-only pruning? Does pruning width first affect which blocks you should remove with depth pruning? Do you need to adjust the loss function weights (alpha, beta, gamma) differently for a model that's been pruned in both dimensions? What's the cumulative impact on capabilities retention, and which type of capability (reasoning, instruction-following, or factual recall) degrades most?
- Challenge for the community: Once you've beaten the baseline, publish your optimized model to the Hugging Face Hub using the automatically generated model card.

6.8 Summary

- Efficient model recovery doesn't start at training, but in the design phase, and that intelligently selecting which blocks to keep (instead of pruning blindly) drastically reduces the computational cost later on.
- Compound loss functions combine three complementary signals: hard labels from the dataset (task loss), soft probability distributions from the teacher (logits loss), and intermediate representations (hidden loss). The balance between these signals should align your final objective. Generative models prioritize logits, while embedding models need higher hidden loss weights.
- You can design Layer Mapping strategies to align a large Teacher with a small Student, addressing the challenge of mismatched architectural depth.
- You've seen how to implement specialized loss functions like Skew KLD and Feature Dynamics, designed to improve convergence when data is scarce or pruning has been aggressive.

- This approach isn't theoretical, but the production standard used by Nvidia (Minitron) and Mistral (Ministral), which allows you to apply top-tier engineering techniques at an accessible scale.

7 Model specialization

This chapter covers

- Adapting models to structured outputs
- Replacing long prompts through fine-tuning
- Efficient fine-tuning with LoRA and DoRA
- Specializing LLMs without losing capabilities

Throughout the previous chapters we've worked on the model's structure through pruning and distillation to gain speed without losing general capability. Now the focus shifts: we're going to adapt the LLM's behavior to our specific data and formats.

To adapt a model to a specific domain, you have two fundamentally different options. RAG gives the model access to external information at inference time: the model itself doesn't change, but it can consult a knowledge base before responding. Fine-tuning takes a different approach: instead of changing what the model sees, it modifies the model's weights directly, reshaping its behavior. This can range from adjusting how a model responds, to introducing vocabulary and concepts from a domain it barely encountered during pretraining.

One clear example is adapting the LLM’s response to structured formats. A generalist LLM tends to converse and add unnecessary explanations—like "Here's the JSON..."—that break production pipelines. Fine-tuning corrects this at the root by transferring the instructions from the prompt directly into the model's weights. The result is robust,

predictable outputs that eliminate the need for lengthy prompts.

Since full retraining is expensive, we'll use parameter-efficient fine-tuning (PEFT) to specialize the model by training less than 1% of its parameters. In this chapter you'll learn to apply QLoRA and QDoRA, exploring everything from matrix decomposition to quantization on accessible hardware. By the end, you'll not only know how to use these techniques; you'll understand their foundations and why they work.

7.1 Efficient fine-tuning

Using fine-tuning to add specialized knowledge to a model or modify its behavior presents two main problems:

- The computational cost. Updating all the weights of a model with billions of parameters requires keeping in memory not just those weights but also the gradients and optimizer states, which can multiply VRAM consumption by a factor of 4x to 8x relative to the model size.
- Modifying the weights can alter knowledge the model acquired during pre-training that we want to preserve. This phenomenon is known as *catastrophic forgetting*.

To solve these two problems, parameter efficient fine-tuning (PEFT) techniques were created. The core idea is to train a small set of new parameters that are added strategically on top of the frozen architecture. The base model is left untouched; the adapters learn the delta: the minimal parameter adjustments required to adapt the model's behavior. This way, we can modify a model's behavior by training less than 1% of its parameters, sometimes even less than 0.1%.

Hugging Face's PEFT library (https://github.com/huggingface/peft) offers different ways to implement this idea. In this chapter, we'll focus on two: LoRA and its extension DoRA. Both are based on the same mathematical concept: low-rank matrix decomposition.

7.1.1 The foundations of LoRA

When we train a full model, we update the weight matrices `W` of each layer. In the model we'll use in the chapter's notebook (SmolLM2-1.7B), its `q_proj` layer (which belongs to the model's Attention module) consists of a 2048 x 2048 matrix; that's more than 4 million parameters. You have to multiply that by the 24 Transformer Blocks in the model, and the four attention matrices each one contains.

Listing 7.1 Attention structure of SmolLM2-1.7B

```
(0-23): 24 x LlamaDecoderLayer(
    (self_attn): LlamaAttention(
        (q_proj): Linear(in_features=2048, out_features=2048, bias=F
alse)
        (k_proj): Linear(in_features=2048, out_features=2048, bias=F
alse)
        (v_proj): Linear(in_features=2048, out_features=2048, bias=F
alse)
        (o_proj): Linear(in_features=2048, out_features=2048, bias=F
alse)
)
```

LoRA (Low-Rank Adaptation) proposes training a delta matrix ΔW that is inserted into the model instead of training the original matrix, which remains frozen. The trainable matrix is the product of two smaller matrices A and B, as shown in figure 7.1.

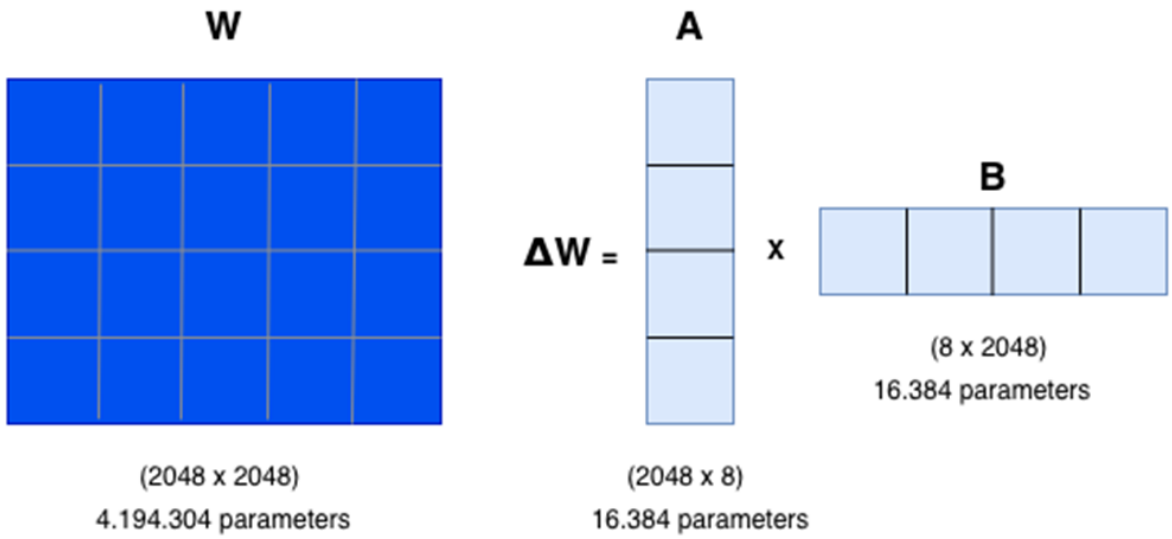


Figure 7.1 LoRA decomposes the weight matrix W (2048×2048) into two low-rank matrices A and B. Together they contain 32,768 parameters — a 99.2% reduction compared to the 4,194,304 of the original matrix.

The size of these two matrices is determined by two values: the original dimension of `W`, which we call `d`, and a hyperparameter you define when creating the LoRA configuration, which we call `r` or rank. The larger `r` is, the more expressive capacity `ΔW` will have, but the more weights you'll need to train.

The dimension `r` cancels out in the matrix multiplication (`A` is `dr` and `B` is `rd`) so that the result `AB` has the same dimensions as `W` and can be added to it at inference time.

The reduction in trainable parameters is remarkable. For `q_proj` with rank 8, instead of training the original 4,194,304 parameters, we train 2048×8 + 8×2048 = 32,768. A 99.2% reduction.

In practice, values of 4, 8, and 16 are the most common starting points. Rank 8 offers a good balance: enough expressive capacity to adapt the model's behavior without adding significant overhead. It's the value you'll use throughout this chapter, and a reasonable default for most

fine-tuning tasks before you have evidence that a higher rank is needed.

In the following listing, we build the decomposition step by step, using the actual dimensions of `q_proj` in SmolLM2-1.7B. The goal is to visualize the structure of LoRA and check the parameter savings.

Listing 7.2 LoRA applied to q_proj of SmolLM2-1.7B

```
import torch

d = 2048 #1
r = 8 #2

W = torch.randn(d, d) #3

A = torch.randn(d, r)
B = torch.zeros(r, d)  #4

delta_W = A @ B

print(f"W dimensions:        {W.shape}")
print(f"ΔW dimensions:       {delta_W.shape}")
print()
params_W    = d * d
params_lora = d * r + r * d
reduction   = (1 - params_lora / params_W) * 100
print(f"Parameters in W:     {params_W:,}")
print(f"Parameters in A + B: {params_lora:,}")
print(f"Reduction:           {reduction:.1f}%")

B_trained = torch.randn(r, d) * 0.01  #5
delta_W_trained = A @ B_trained

W_adapted = W + delta_W_trained  #6

print(f"\nW dimensions (unchanged):        {W_adapted.shape}")
print(f"Sample weights before adaptation: {W[0, :4]}")
print(f"Sample weights after adaptation:  {W_adapted[0, :4]}")
```

The response is:

```
W dimensions:         torch.Size([2048, 2048])
ΔW dimensions:        torch.Size([2048, 2048])

Parameters in W:      4,194,304
Parameters in A + B: 32,768
Reduction:            99.2%

W dimensions (unchanged):         torch.Size([2048, 2048])
Sample weights before adaptation: tensor([-1.1684, -0.1492, -0.4845,
-0.5751])
Sample weights after adaptation:  tensor([-1.1576, -0.1631, -0.4860,
-0.6224])
```

The code defines `d` #1 with the same size as the `q_proj` dimension in `SmolLM2-1.7B`, and `r` #2 as the hyperparameter controlling the adapter size, which we'll use later in the LoRA configuration. `W` #3 represents the original weight matrix, which remains completely frozen during real training. `B` is initialized to zeros #4, following the original LoRA paper. This ensures that `ΔW = AB = 0` at the start of training. The `@` operator performs matrix multiplication in Python: It takes the rows of `A` and the columns of `B` and computes their dot product, producing `ΔW` with the same dimensions as `W`.

To demonstrate the adaptation effect, we simulate a post-training state by assigning small values to `B` #5. The 0.01 scale reflects typical magnitudes observed after fine-tuning.

Finally, `W_adapted` #6 confirms the theoretical promise. The sum `W + ΔW` yields a matrix with identical dimensions to `W`, with subtly modified weights but an intact model structure.

LoRA has proven extremely efficient and has become the de facto standard for model fine-tuning. Recently, an evolution called DoRA was introduced, enabling fine-grained control during the fine-tuning process.

7.1.2 Extending LoRA with weight decomposition

The main limitation of LoRA is that when computing $\Delta W = A \cdot B$, it simultaneously alters both the vector's magnitude and direction. As discussed in Chapter 4, when measuring the cosine similarity of Transformer block outputs, we distinguished between these two components: magnitude affects strength, while direction represents semantic meaning.

Full fine-tuning does not alter these variables with the same intensity. Their correlation is -0.62. When one increases, the other tends to decrease. With LoRA, however, this correlation rises to 0.83 (as shown in the papers we'll review later), meaning magnitude and direction almost always change together, limiting flexibility. This is the problem that DoRA (Weight-Decomposed Low-Rank Adaptation) was designed to solve.

Figure 7.2 shows how DoRA addresses this limitation by decomposing the original matrix W into two components: a magnitude vector (m) and a direction matrix (v). LoRA adjusts the direction matrix (v), while (m) is treated as an independent trainable parameter.

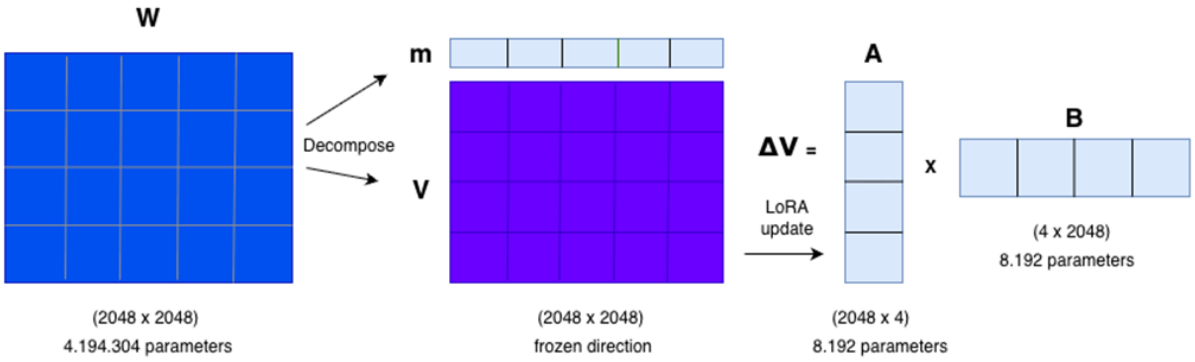


Figure 7.2 W decomposes into a magnitude vector m and a frozen direction matrix V. LoRA trains matrices A and B to compute the directional update ΔV, while m is updated independently as a trainable parameter.

By isolating the direction update, DoRA gives us three key advantages for model specialization:

1. Less capabilities degradation: By not forcing unnecessary changes to the magnitude, the model's base knowledge is much less affected. On top of that, this efficiency allows using much smaller ranks (r) without performance collapsing.
2. Greater semantic learning capacity: By decoupling magnitude and direction, its update pattern mimics full fine-tuning, which allows it to assimilate new domain knowledge and complex reasoning with greater precision.
3. Effectiveness with small datasets: By simplifying the optimization problem, DoRA achieves better adaptation than LoRA even when we have very few training examples.

Let's see how this decomposition translates into code.

Listing 7.3 DoRA applied to q_proj of SmolLM2-1.7B

```
import torch

d = 2048
r = 4 #1

W0 = torch.randn(d, d)  #2

m = torch.norm(W0, dim=0, keepdim=True)  #3
V = W0 / m #4

A = torch.randn(d, r)
B = torch.zeros(r, d)

B_trained = torch.randn(r, d) * 0.01 #5

V_updated     = V + A @ B_trained #6
V_updated_norm = torch.norm(V_updated, dim=0, keepdim=True)
W_dora = m * (V_updated / V_updated_norm) #7

params_lora   = d * r + r * d
params_dora_m = d

print(f"W0 shape:                    {W0.shape}")
print(f"W_dora shape:                {W_dora.shape}")
print()
print(f"Parameters in A + B:         {params_lora:,}")
print(f"Additional magnitude params: {params_dora_m:,}")
print()
print(f"Sample W0[0, :4]:     {W0[0, :4]}")
print(f"Sample W_dora[0, :4]: {W_dora[0, :4]}")
```

The response is:

```
W0 shape:                    torch.Size([2048, 2048])
W_dora shape:                torch.Size([2048, 2048])

Parameters in A + B:         16,384
Additional magnitude params: 2,048

Sample W0[0, :4]:     tensor([-1.3111, -0.1597, -0.5458,  0.9187])
Sample W_dora[0, :4]: tensor([-1.0166, -0.3212, -0.2501,  1.6506])
```

We've lowered `r` to 4 #1 to illustrate one of DoRA's theoretical advantages: its ability to maintain performance at lower ranks than LoRA. In practice, the right value of `r` remains an empirical decision. In the training experiments later in this chapter, both techniques use `r=8` for a fair comparison. `W0` #2 is the frozen pre-trained weight.

The DoRA update follows four sequential steps:

1. **Decompose**: `torch.norm` #3 extracts the magnitude of each column, just like the normalization we applied in chapter 4 to isolate direction before computing cosine similarity. Dividing by those magnitudes #4 produces `V`, where each column has unit norm and represents only the direction of the original weight.
2. **Update**: The post-training state is simulated with small values in `B` #5, stored here as `B_trained`. The update #6 shifts each direction column by the low-rank term `A @ B_trained`, the component LoRA learns during training.
3. **Renormalize**: The renormalization ensures the columns return to unit length, keeping the separation between magnitude and direction intact.
4. **Recombine**: `W_dora` #7 scales each normalized column by its corresponding magnitude from `m`, producing a matrix with identical dimensions to `W0` that merges at inference time exactly like a standard LoRA update, with no additional computational cost.

With the decomposition clearly laid out in code, the next step is fitting this training into a real GPU. For that, we need quantization.

7.2 Quantization and the memory bottleneck

One of the main problems we run into with fine-tuning is memory consumption. With PEFT, we've solved the gradients and optimizer states problem. Because the base model is frozen, we only train the tiny adapters. But the frozen base model still takes up all its original memory, and that alone can be enough to not fit in the GPU we have available.

Quantization helps us solve this problem by reducing the numerical precision of the base model's weights. The idea is simple. Instead of representing each weight as a 32-bit floating point number, we represent it with 16, 8, or even 4 bits. The base model takes up much less memory this way, which frees up space for the gradients and optimizer states that training needs.

NOTE

Quantization during fine-tuning is not a universal rule. If the model fits in memory, training without quantization is faster. We quantize when memory is the real bottleneck, which is a common scenario outside a lab with dedicated hardware.

Quantization also brings a loss of precision, as shown in figure 7.3.

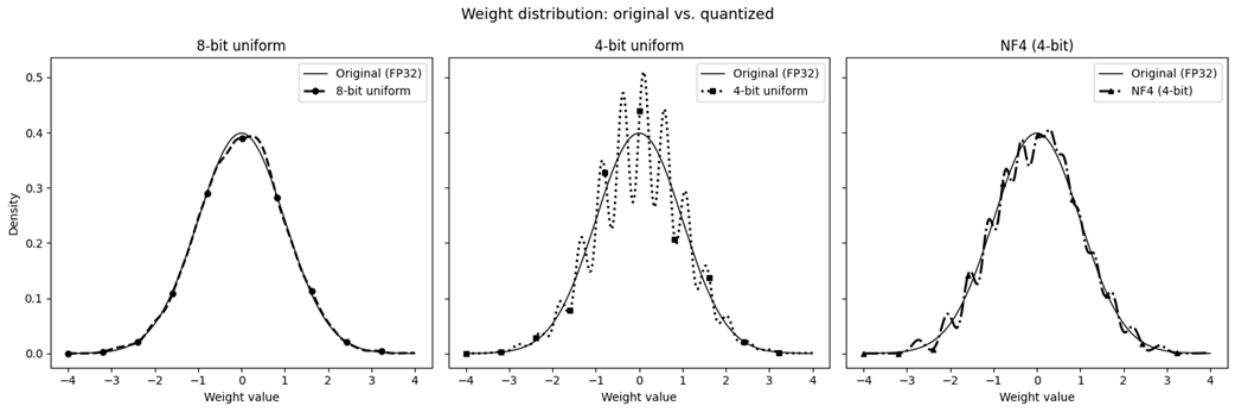


Figure 7.3 Weight distribution after quantization and dequantization across three methods. 8-bit uniform (left) closely matches the original Gaussian. 4-bit uniform (center) introduces severe distortion, with values collapsing into a small number of discrete levels. NF4 (right), also 4-bit, recovers the distribution more faithfully by concentrating its quantization levels where weights are densest.

Looking at the panels in figure 7.3 you might ask yourself: why not always use 8 bits, which clearly preserves the original distribution much better? The reason is simple: going from 16 to 8 bits cuts the model size in half, but going to 4 bits reduces it to a quarter, and that difference is usually what determines whether the model fits in the GPU during training.

NF4 (NormalFloat 4-bit) is a data type introduced in the QLoRA paper (https://arxiv.org/abs/2305.14314) for the weights of pretrained LLMs. With 4 bits you can only represent 16 distinct values, and every weight in the model must be mapped to one of them. Uniform quantization distributes those values at equal intervals across the full range: if your weights run from -1.0 to 1.0, you get levels like -1.0, -0.87, -0.73... 0.0, 0.13, 0.27... 1.0.

The problem is that LLM weights follow a normal distribution: the vast majority are clustered around zero. In that dense region, weights that were originally 0.08, 0.10, and 0.12 might all get rounded to the same value, 0.13, losing the differences between them. NF4 fixes this by

packing more quantization levels near zero, and fewer at the extremes. The result, visible in the right panel of figure 7.3, is the smallest possible 4-bit memory footprint with significantly less degradation than uniform quantization, which is why NF4 has become the de facto standard for quantizing LLMs.

NOTE

NF4 is a storage format, not a compute format. No current GPU architecture—including NVIDIA Ampere or Hopper—executes matrix multiplications natively in 4 bits. In practice, weights are dequantized on the fly to 16-bit precision before each operation. The memory savings are real; the compute simply happens at higher precision.

In practice, the memory footprint of this setup is modest. Running on a T4 GPU, the complete training configuration (base model in NF4, adapters, gradients, and optimizer states) peaks at 2.34 GB for QLoRA and 2.69 GB for QDoRA. Both fit comfortably within an 8 GB card, the lower end of what most consumer hardware offers today.

Combining LoRA and DoRA with quantization gives us QLoRA and QDoRA, the two fine-tuning techniques we'll be testing to adapt our model.

7.3 Specializing with QLoRa and QDoRA

Up to this point we've built the theoretical foundation. We know how LoRA decomposes a weight update into two small matrices, how DoRA separates magnitude and direction for more precise tuning, and how NF4 quantization lets us work

with models that would otherwise not fit in our GPU. Now we put it all together to demonstrate how fine-tuning can transfer behavioral instructions permanently from a prompt into a model's weights.

To demonstrate it you'll specialize SmolLM2-1.7B-Instruct to extract clinical information from medical notes and return it as structured JSON with a strict schema.

You'll work with the notebook CH07_NB02_L4_QLoRA_QDoRA.ipynb, which contains the full training pipeline with QLoRA and QDoRA. You can also choose to use CH07_NB02_T4_QLoRA_QDoRA.ipynb, specifically prepared to run on a smaller GPU and which doesn't execute the full benchmark suite. The training pursues two goals: replacing the fifteen-plus line prompt with a single instruction while preserving as much of the model's general capabilities as possible.

7.3.1 Base model and clinical dataset

The task we're going to solve is structured clinical information extraction: given the free text of a medical note, the model must identify entities such as symptoms, medications, or vital signs and return them in a JSON with a strict schema.

This is a classic production problem. The model already has the necessary knowledge, but it doesn't perform reliably without very detailed instructions.

The model we're going to use is SmolLM2-1.7B-Instruct, capable of identifying those entities, but it needs us to give it a very complete prompt so it adheres to the expected response schema, and even then it doesn't always comply.

We will train the model to extract information from medical notes and return the response as a JSON object with the schema shown in the following listing.

Listing 7.4 Clinical NER JSON schema

```
{
  "patient_age": int or null,
  "symptoms": [list of normalized strings],
  "vital_signs": {
    "temperature": float or null,
    "heart_rate": int or null,
    "blood_pressure": string or null
  },
  "medications": [
    {"name": string, "dose": string, "frequency": string}
  ],
  "duration_days": int or null
}
```

Any missing key, or an incorrect data type (like returning the age as a string "32" instead of the integer 32) will be considered a validation failure.

To perform the fine-tuning, we need a dataset. In this case you'll use a synthetic dataset I created specifically for the occasion, which you can find on Hugging Face: https://huggingface.co/datasets/oopere/clinical-ner-qdora.

The dataset consists of 400 samples (360 for training and 40 for testing) that pair medical text with the structured output format from listing 7.4. To ensure the model learns to internalize the structure and isolate the useful information, the dataset intentionally injects noise by splitting the samples into five categories:

- Clean (100 samples): Standard, well-written professional notes.

- Abbreviations (80 samples): Texts that include common medical jargon and abbreviations.
- Implicit (80 samples): Cases where symptoms are described indirectly (for example, "the patient notices the room is spinning" instead of "vertigo").
- Typos (70 samples): Typos and informal grammar.
- Irrelevant (70 samples): Notes that contain personal context or anecdotal details that the model must learn to discard.

The dataset has been synthetically generated for use in this chapter using LiteLLM across different language models, with automatic validation against the schema before including each sample. The schema is intentionally simple and designed for demonstration purposes. Since synthetic data creation is an important step in many real-world projects, the repository includes the notebook CH07_NB_dataset_generator.ipynb in case you want to explore or replicate this process.

This complete training pursues three measurable objectives that you'll evaluate at the end:

1. Inference efficiency: Drastically reduce token consumption by replacing the multi-line system prompt with a single one-word instruction ("Extract:").
2. Reliability: Maintain or improve the reliability achieved by the base model with the expanded prompt.
3. Knowledge preservation (Capabilities): Achieve specialization while avoiding catastrophic forgetting in the base model's capabilities.

Once these objectives are clear, you're ready to configure your first training run using QLoRA.

7.3.2 Establishing the baseline: reliability and capabilities

Like in any experiment, the first thing we need to do is establish the baseline we're starting from. In this case, we need to evaluate the behavior in two different areas: how reliable the model is at generating responses that follow the established schema, and what level of capabilities it has before any modifications.

We need to maintain a general performance baseline, because even when you're training the model for a very specific task, you're altering weights that can affect its behavior in other areas.

We then load SmolLM2-1.7B-Instruct in 4 bits using the NF4 configuration:

Listing 7.5 Loading quantized SmolLM

```
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4", #A
    bnb_4bit_compute_dtype=torch.bfloat16, #B
    bnb_4bit_use_double_quant=True, #C
)

tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right" #D

model = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    quantization_config=bnb_config,
    device_map="auto",
)
```

#A Sets NF4 as the data type for storing the quantized weights.
#B NF4 weights are dequantized to this precision before each matrix multiplication.
#C Reduces memory consumption with minimal impact on quality.
#D Required for training; use left padding for batched inference.

With the model loaded, the next step is to define what "correct response" actually means. For this task we won't use perplexity or accuracy. We'll define the `check_schema_compliance` function, which checks field by field whether the model's response is a JSON like the one in listing 7.4.

Listing 7.6 Schema validation

```
REQUIRED_SCHEMA = {
    "patient_age":  (int, type(None)),
    "symptoms":     list,
    "vital_signs":  dict,
    "medications":  list,
    "duration_days": (int, type(None)),
}

VITAL_SIGNS_KEYS = {"temperature", "heart_rate", "blood_pressure"}

def check_schema_compliance(response_text: str) -> dict:
    cleaned = response_text.strip()\
                      .removeprefix("```json")\
                      .removesuffix("```").strip() #A

    try:
        parsed = json.loads(cleaned)
    except json.JSONDecodeError as e:
        return {
            "is_valid_json":  False,
            "matches_schema": False,
            "failure_reason": f"JSON parse error: {e}",
        }

    for key, expected_type in REQUIRED_SCHEMA.items(): #B
        if key not in parsed:
            return {
                "is_valid_json":  True,
                "matches_schema": False,
                "failure_reason": f"Missing key: '{key}'",
            }
        if not isinstance(parsed[key], expected_type):
            return {
                "is_valid_json":  True,
                "matches_schema": False,
                "failure_reason": f"Wrong type for '{key}': "
                                  f"got {type(parsed[key]).__name_
_}",
            }

    for vk in VITAL_SIGNS_KEYS:  #C
        if vk not in parsed["vital_signs"]:
```

```
            return {
                "is_valid_json":  True,
                "matches_schema": False,
                "failure_reason": f"Missing vital_signs key: '{v
k}'",
            }

    return {
        "is_valid_json":  True,
        "matches_schema": True,
        "failure_reason": None,
    }
```

#A The model can wrap its response in markdown blocks. We remove them before parsing.
#B We verify that all schema keys are present and with the correct type. A patient_age returned as string "32" instead of integer 32 counts as a failure.
#C vital_signs has its own validation because it's a nested object with three required sub-keys.

The function returns three fields: `is_valid_json` indicates whether the response is parseable JSON, `matches_schema` indicates whether it also respects the full contract, and `failure_reason` explains exactly what failed.

Before we evaluate the base model, we need to define the exact instructions it requires to perform this task. Since the base model has not been optimized for structured extraction, we must rely on prompt engineering to enforce the constraints. Here is the strict prompt we will use to guide our LLM:

```
STRICT_PROMPT = """You are an automated medical data extraction syst
em.
Your only task is to extract clinical information into a strict JSON
format.
Use EXACTLY this schema:
{
  "patient_age": int or null,
  "symptoms": [list of normalized strings],
  "vital_signs": {
    "temperature": float or null,
    "heart_rate": int or null,
    "blood_pressure": string or null
  },
  "medications": [
    {"name": string, "dose": string, "frequency": string}
  ],
  "duration_days": int or null
}
Respond ONLY with the raw JSON. No markdown, no explanation, no extr
a text."""
```

With the metric and the prompt defined, we evaluate the base model on the 40 test set samples using the fifteen-line `STRICT_PROMPT`. The aggregated result is 85% schema compliance. However, the breakdown by category reveals something more interesting:

```
                    samples  valid_json_pct  schema_ok_pct
category
abbreviations         6            100.0            100.0
clean                 8            100.0            100.0
implicit              7            100.0            100.0
irrelevant           12             58.3             58.3
typos                 7            100.0            100.0
```

With only 40 samples, these numbers are directional signals, not statistically robust measurements. The model handles clean notes, medical abbreviations, implicit symptoms, and typos without trouble. It fails on irrelevant; when the text

includes contextual noise that doesn't fit the prompt's instructions, the model gets lost.

A system guided solely by a prompt tends to be robust under controlled conditions but vulnerable as soon as the text drifts from what the instructions anticipated.

IMPORTANT

The 58.3% failure on irrelevant inputs illustrates a hard limit of prompt-only systems. The prompt can define the task, but it cannot generalize the model's behavior to inputs it was never designed to handle. Fine-tuning addresses precisely this gap, not by writing better instructions, but by reshaping what the model has learned.

With schema compliance established, we complete the baseline by measuring the model's general capabilities before any training. We find the execution of five standard benchmarks on 500 samples each:

Listing 7.7 Obtaining benchmarks

```
BENCHMARK_TASKS = [
    "arc_easy",
    "winogrande",
    "hellaswag",
    "lambada_openai",
    "piqa"
]
BENCHMARK_LIMIT = 500

baseline_benchmarks = model_evaluation(
    model, tokenizer, BENCHMARK_TASKS,
    device=DEVICE, limit=BENCHMARK_LIMIT, batch_size=4
)

for task, metrics in baseline_benchmarks.items():
    print(f"  {task}: {metrics}")
```

What it returns:

```
Baseline benchmark results:
  arc_easy: {'accuracy': '0.6520', 'acc_norm': '0.6200'}
  hellaswag: {'accuracy': '0.4860', 'acc_norm': '0.6100'}
  lambada_openai: {'perplexity': '6.08', 'accuracy': '0.6020'}
  piqa: {'accuracy': '0.7740', 'acc_norm': '0.7940'}
  winogrande: {'accuracy': '0.6540'}
```

With these results you have the complete baseline: 85% schema compliance with strict prompt and reference benchmarks to measure knowledge preservation. These metrics represent the two axes you'll use to evaluate whether fine-tuning has met its three objectives.

In the next section, you'll train the model with QLoRA and measure how much these numbers move.

7.3.3 QLoRA training on the clinical dataset

Up to this point, you've established the baseline of the SmolLM2-1.7B-Instruct model in 4 bits, which reaches 85%

reliability (schema compliance accuracy) using a structured fifteen-line prompt.

Now you're going to train the model to perform the same function, but using a prompt of just one word: "Extract:". To achieve what's illustrated in figure 7.4, replace a prompt with detailed instructions with a single instruction.

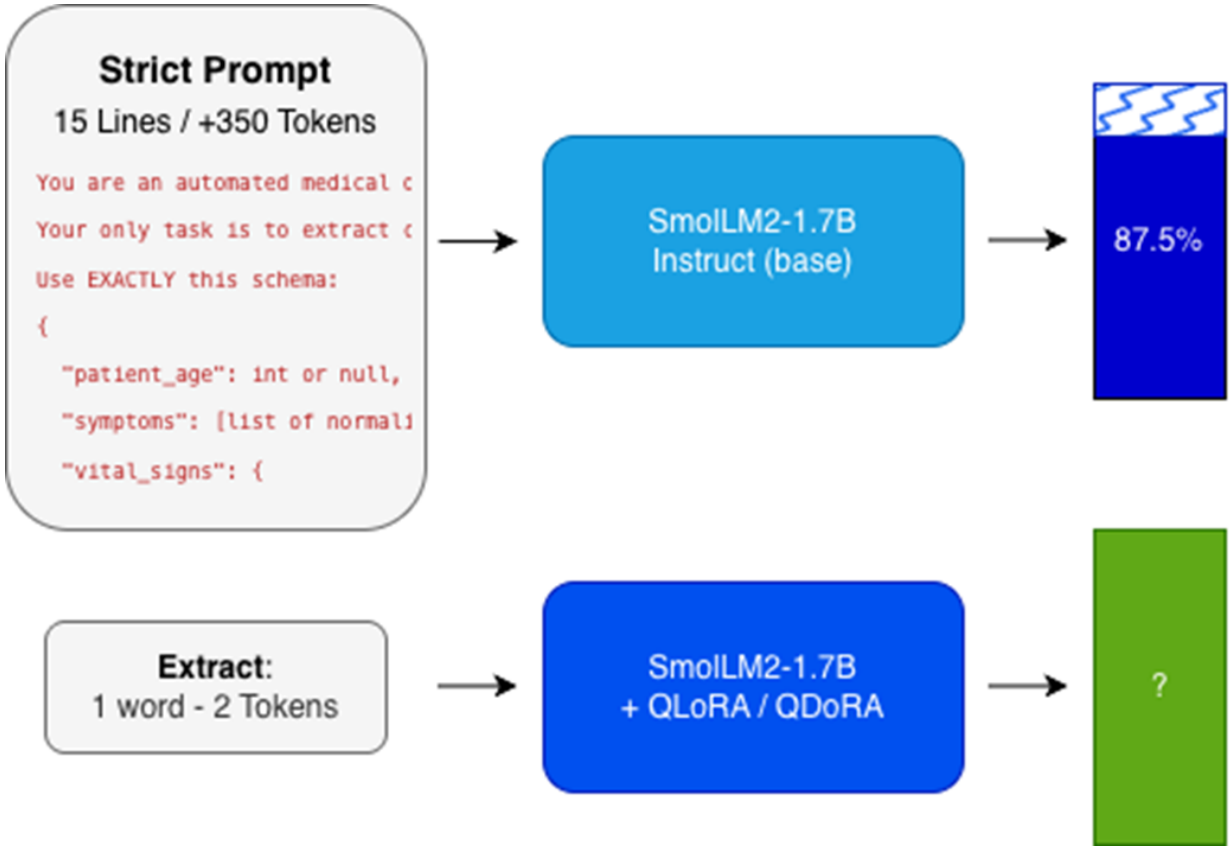


Figure 7.4 Prompt compression through fine-tuning. The base model (top) requires a 15-line system prompt with explicit schema instructions to reach 85% schema compliance. After fine-tuning with QLoRA or QDoRA (bottom), the same task is triggered by a single word. The medical note is present in both cases, what disappears is the instruction overhead.

The first step is to use the training dataset so the model sees the same interaction format during training that we want at inference time.

Instruct models (like SmolLM2-1.7B-Instruct) don't process text as a flat string. They're trained to interpret data following a conversation structure based on three specific roles. In listing 7.8, we build each training example by mapping our dataset to these roles through the tokenizer's chat template:

- System: Defines the global context, rules, or expected behavior. Here we inject our MINIMAL_PROMPT ("Extract:") so the model associates this short instruction as its main directive.
- User: Represents the input data or the request. In our case, we insert the raw free-text clinical note here.
- Assistant: Represents the model's output. During training, we place the ground truth here (the structured, validated JSON).

By enforcing this structure and passing it through the chat template, we teach the model the relationship it needs to learn: when the system says "Extract:" and the user provides a medical text, the only valid response from the assistant is a strict JSON schema.

Listing 7.8 Formatting the training examples

```
def format_training_example(sample: dict) -> str:

    label = json.loads(sample["label"]) if isinstance(sample["labe
l"], str) \
            else sample["label"]

    messages = [
        {"role": "system",    "content": MINIMAL_PROMPT},  #A
        {"role": "user",      "content": sample["note"]},
        {"role": "assistant", "content": json.dumps(label, indent=
2)},
    ]

    return tokenizer.apply_chat_template(  #B
        messages,
        tokenize=False,
        add_generation_prompt=False,  #C
    )

train_dataset = dataset["train"].map(
    lambda x: {"text": format_training_example(x)},
    remove_columns=dataset["train"].column_names,
)
```

#A We inject MINIMAL_PROMPT ("Extract:") in the system role so the model learns to associate this instruction with the JSON it needs to generate.
#B apply_chat_template serializes the conversation in the exact format the model expects.
#C The assistant's response is already included in the example. Adding the turn-start token would alter the conversation structure.

The next step is to prepare the quantized model for training. A model loaded in 4-bit has its weights in a format that doesn't allow gradients to be computed directly. We use the `prepare_model_for_kbit_training` function from Hugging Face's PEFT library to enable the model to compute gradients so it can learn.

Listing 7.9 Preparing the quantized model to receive gradients

```
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_
training

model = prepare_model_for_kbit_training(model)
```

With the model ready, we then define the LoRA adapter configuration using the PEFT library.

Listing 7.10 LoRA configuration

```
lora_config = LoraConfig(
    r=8,    #1
    lora_alpha=16, #2
    lora_dropout=0.05, #3
    bias="none",
    task_type="CAUSAL_LM",
    use_dora=False, #4
    target_modules=[
        "q_proj", "k_proj", "v_proj", "o_proj",
        "gate_proj", "up_proj", "down_proj",
    ], #5
)

model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
```

Which outputs:

```
trainable params: 9,043,968 || all params: 1,720,420,352 || trainabl
e%: 0.5257
```

Remember, we're only training 0.53% of the model's parameters; the rest remain frozen.

The value `r=8` #1 is the same hyperparameter, with the same value, that you saw in section 7.1.1, where you could check how it produces a 99.2% saving in trainable parameters. The hyperparameter `lora_alpha=16` #2 scales the contribution of the adapters over the original weights following the common

practice of using double the rank. `lora_dropout=0.05` #3 randomly disables 5% of connections during training to prevent overfitting. The parameter `use_dora=False` #4 indicates that we'll use QLoRA. (In the next section, you'll change this single value to `True` and repeat the experiment under identical conditions.) The seven modules in #5 cover the full Attention Module (`q_proj, k_proj, v_proj, o_proj`) and the full MLP Module (`gate_proj, up_proj, down_proj`) that we defined in chapter 3, allowing fine-tuning to adjust both how the model attends to information and how it transforms it.

With the adapters initialized, we can configure and launch the training.

Listing 7.11 Configuring and training QLoRA

```
from trl import SFTTrainer, SFTConfig #1

training_args = SFTConfig(
    output_dir="./qlora_output",
    num_train_epochs=3, #2
    per_device_train_batch_size=8,
    gradient_accumulation_steps=1,
    learning_rate=2e-4, #3
    lr_scheduler_type="cosine", #4
    warmup_steps=7, #5
    fp16=False,
    bf16=True, #6
    logging_steps=10,
    save_strategy="no",
    max_length=MAX_SEQ_LENGTH,
    dataset_text_field="text", #7
    report_to="none",
)

trainer = SFTTrainer(
    model=model,
    train_dataset=train_dataset,
    args=training_args,
)
trainer.train()
```

The `SFTConfig` class #1 centralizes the training configuration. The model trains for three epochs #2, with a `learning_rate` of `2e-4` #3, a standard value for LoRA training. The cosine scheduler #4 will gradually reduce the learning rate throughout training.

Since the dataset consists of 360 samples and we run them in batches of 8 elements over 3 epochs, we have 135 steps. The model will update the trainable weights 135 times during this training process. Since our adapter matrices have just been initialized, applying the maximum learning rate (2e-4) from the first step could cause overly aggressive

weight updates. The warmup #5 acts as a soft start, progressively increasing the learning rate from zero up to our target value.

Additionally, `bf16=True` #6 is enabled because our L4 GPU architecture supports it natively; this format is considerably more stable than Float16 for training, as it avoids overflow issues by preserving the same dynamic range as Float32.

Finally, `dataset_text_field="text"` #7 tells the `SFTTrainer` which column of the dataset contains the examples formatted by `format_training_example()` from listing 7.8, which the model uses to learn the next token to generate.

Once training is complete, we evaluate the model on the 40 test samples from the dataset using the prompt "Extract:" to check whether the objectives have been met:

```
QLORA — Schema compliance by category (minimal prompt)

               samples  valid_json_pct  schema_ok_pct
category
abbreviations        6           100.0          100.0
clean                8           100.0          100.0
implicit             7           100.0          100.0
irrelevant          12           100.0          100.0
typos                7           100.0          100.0

Baseline strict prompt : 85.0%
QLoRA   minimal prompt : 100.0%
```

The model reaches 100% schema compliance across all categories. The most notable thing is that we've fixed the 50% failure rate we had in the irrelevant category when evaluating the baseline with the strict prompt. The knowledge of how to structure the output and how to ignore contextual noise no longer lives in the input text; it has been permanently absorbed into the LoRA weights. You haven't

just managed to reduce the prompt size. You've also gotten a model that actually improves on the baseline.

With schema compliance confirmed, what remains is to see the impact on capabilities. Figure 7.5 shows the comparison between the base model and the QLoRA fine-tuned model across the five general capability benchmarks.

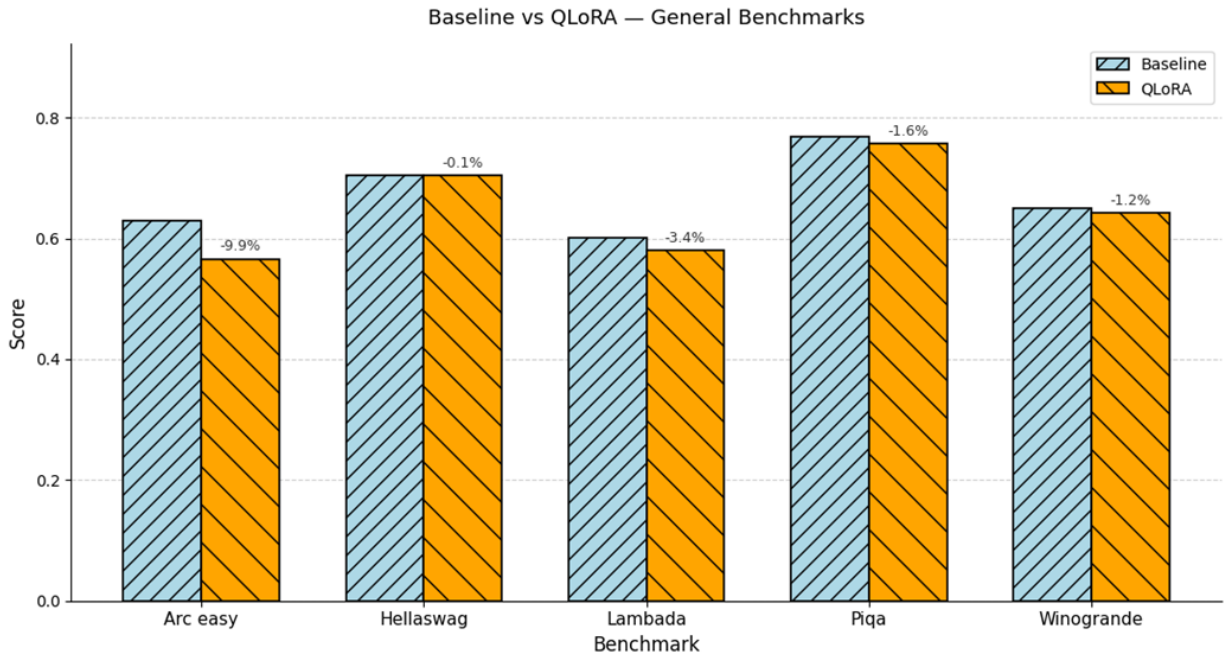


Figure 7.5 Baseline vs QLoRA benchmark comparison on five general-knowledge tasks. Scores use acc_norm where available; accuracy otherwise. The 10% drop in ARC-Easy is the only notable degradation; the remaining four benchmarks show smaller variations,indicating that specialization preserves the model's general capabilities with minor trade-off.

The weighted average degradation is 3.2%, a result that indicates the model retains its general capabilities with only a minor and capability drop. The specialized model is still, in essence, the same general model. There's one number that deserves attention: `arc_easy` drops 10% relative, the only noticeable decline across the five benchmarks.

ARC-Easy evaluates commonsense reasoning on elementary science questions, a domain with no relationship whatsoever to the clinical notes in the dataset. The most likely hypothesis is that the model has reoriented part of its generative distribution toward structured medical language, and that reorientation slightly impacts general-purpose reasoning. The other four benchmarks remain practically stable, with variations below 3%.

Perplexity on lambada goes up from 5.96 to 6.90, an increase of 15.8%, while accuracy drops 3.4%. The most likely explanation is not a degradation of general reasoning but a narrowing of output style: the model has adapted to a very specific format and prompting pattern, which reduces its flexibility when evaluated under different prompting conventions. It still completes correctly at nearly the same rate, but with less confidence when the expected output format differs from what it learned during fine-tuning.

7.3.4 QDoRA training on the clinical dataset

Applying QDoRA instead of QLoRA is as simple as changing a single parameter in the LoRA configuration and running the training on a new model. We clean up the environment and load the model again.

Listing 7.12 Reloading the base model for QDoRA

```
del model
torch.cuda.empty_cache()

model = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    quantization_config=bnb_config,
    device_map="auto",
) #A

model = prepare_model_for_kbit_training(model) #B
```

#A We reload the base model from Hugging Face, with no active adapters.
#B We prepare the quantized model to receive gradients again, just like we did before QLoRA training.

With a clean base model, the QDoRA configuration is almost identical to QLoRA. The only real change is one parameter: `use_dora=True`. Nothing else to touch.

Listing 7.13 QDoRA configuration

```
dora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
    use_dora=True, #A
    target_modules=[
        "q_proj", "k_proj", "v_proj", "o_proj",
        "gate_proj", "up_proj", "down_proj",
    ],
)

model = get_peft_model(model, dora_config)
model.print_trainable_parameters()
```

#A The only change compared to QLoRA. PEFT takes care of adding the magnitude vectors m.

Which outputs:

```
trainable params: 9,682,944 || all params: 1,721,059,328 || trainabl
e%: 0.5626
```

If you compare it with the 9,043,968 trainable parameters from QLoRA, you'll see an increase of about 639,000 units. It's not arbitrary. Those are exactly the magnitude vectors `m` that DoRA adds per column of the target matrices. The same ones you saw separated from the direction `V` during the decomposition in section 7.1.2. The `SFTConfig` and `SFTTrainer` are exactly the same ones you used with QLoRA, without touching any hyperparameter.

The training process is also exactly the same as the one used with QLoRA. Let's see if the results have changed.

```
QDORA — Schema compliance by category (minimal prompt)

               samples  valid_json_pct  schema_ok_pct
category
abbreviations        6           100.0          100.0
clean                8           100.0          100.0
implicit             7           100.0          100.0
irrelevant          12           100.0          100.0
typos                7           100.0          100.0

Baseline strict prompt : 85.0%
QLoRA   minimal prompt : 100.0%
QDoRA   minimal prompt : 100.0%
```

Both methods reach 100% in schema compliance, absorbing the prompt into the model's new weights.

The benchmark results are shown in figure 7.6.

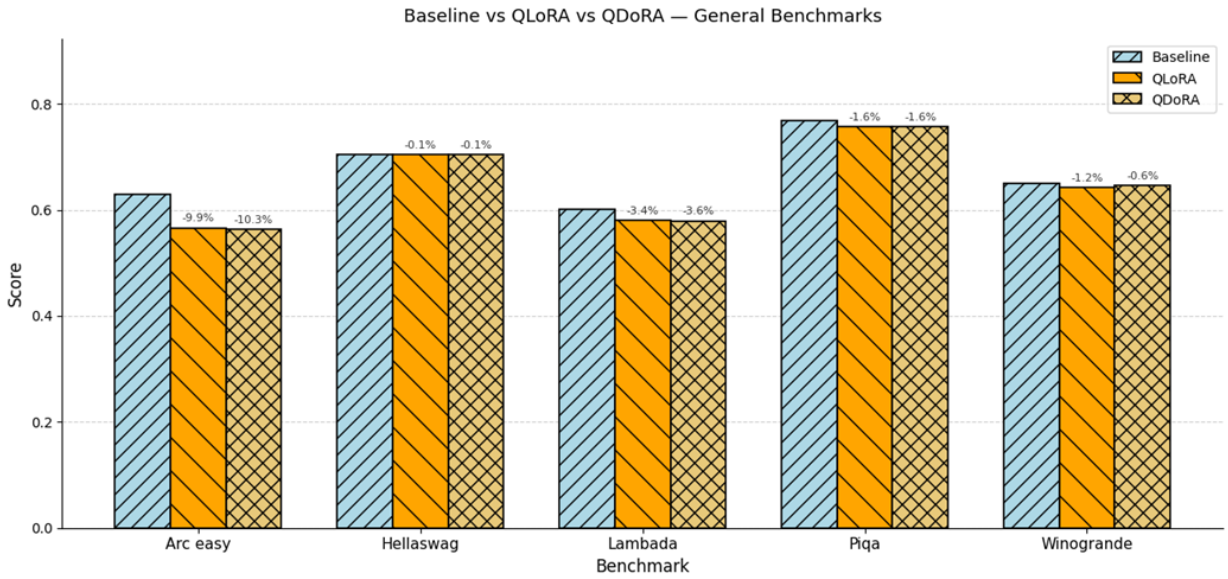


Figure 7.6 Benchmark comparison across Baseline, QLoRA, and QDoRA on five standard tasks. Both fine-tuned models show moderate capability drops relative to the baseline, with ARC-Easy registering the largest decline (-10.3%) in both cases. Differences between QLoRA and QDoRA are marginal across all tasks.

Both techniques produce a moderate drop compared to the baseline, which is expected because we're modifying weights that were trained in general tasks to specialize them in a single structured task. What stands out is that QDoRA doesn't improve on QLoRA, not even slightly. The values are practically interchangeable.

DoRA's theory predicts advantages especially on small datasets and with low ranks. The truth is that in this specific experiment that advantage doesn't materialize. A reasonable hypothesis is that the synthetic dataset, generated with a very deterministic output schema, doesn't present the semantic variation needed for the magnitude-direction decomposition to provide a measurable improvement. Papers propose; data disposes.

7.3.5 Evaluating the results against our goals

At the beginning of section 7.3.1 we set three measurable objectives for this training. It's time to evaluate them with the data in hand.

The first was inference efficiency: replacing the fifteen-line prompt with a single instruction. Both QLoRA and QDoRA achieve this without reservations. The model has internalized the format contract in its weights and responds correctly to "Extract:" in one hundred percent of the test cases, including the irrelevant category, which was the weak point of the baseline at 50% compliance. The knowledge that used to live in the prompt now lives in the model.

The second objective was reliability. The baseline reached 85.0% schema compliance using the strict prompt. Both techniques bring it to 100% using the minimal prompt. Not only do we not lose reliability by simplifying the instruction but we improve it.

The third one is the most nuanced. Preserving general knowledge comes at a cost. Both models show a moderate drop in benchmarks compared to the baseline, more pronounced in ARC-Easy (-10.3%) and more contained in the rest. This trade-off is expected and, in most production contexts, perfectly acceptable because you're trading general capability for specialized behavior.

On the choice between QLoRA and QDoRA, the results of this experiment don't give a clear answer on quality, but they do on efficiency: QLoRA trains faster because it doesn't manage the magnitude vectors m as independent parameters. For structural or format adaptation tasks like this one, QLoRA is the pragmatic choice. QDoRA's advantage may emerge when the task requires deep semantic adaptation, where the decoupling of magnitude and direction provides benefits that justify the additional training time.

In section 7.4, we will examine the specific conditions where DoRA should offer a real advantage, and what the research indicates.

7.3.6 Merging weights and dynamic adapter routing

During model training, its weights remained frozen. It was the PEFT library that was responsible for creating a set of trainable weights and combining them transparently with the model during inference. The new weights were created in a separate artifact known as an adapter. The *adapter* is a simple directory that gets created with the following code:

```
model.save_pretrained("./qdora_adapter")
```

The `save_pretrained` function can be called at the end of training and will create the directory with the necessary files to load and use the adapter later.

To use the adapter you can choose between two strategies: keeping the adapter separate or permanently merging it with the base model.

The simplest option is the merge. The operation adds the adapter weights to the original model weights and removes all PEFT infrastructure. The result is a standard model, with no external dependencies, that loads and serves exactly like any other model. It's the natural choice when you have a single specialization and want maximum inference speed.

Listing 7.14 Merging the Adapter into the Base Model

```
merged_model = model.merge_and_unload() #1

merged_model.save_pretrained("./qdora_merged") #2
tokenizer.save_pretrained("./qdora_merged")
```

The weights are merged in the `merge_and_unload()` function #1, returning a standard model. To serialize it you use `save_pretrained()` #2. From here you can load it with `AutoModelForCausalLM.from_pretrained()` like any other model, with no PEFT dependency at all.

The other option is to keep the base model in memory and load the adapter when needed. If you have different adapters—for example, if you needed two completely different formats depending on the data source (different clinics or hospitals)—you could keep a single base model and load the right adapter to process the data.

The following listing demonstrates how to load the two adapters you trained in this chapter onto the same base model.

Listing 7.15 Dynamic Adapter Loading over the Base Model

```
from peft import PeftModel

base_model2 = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    quantization_config=bnb_config,
    device_map="auto",
) #1



model_with_adapter = PeftModel.from_pretrained(
    base_model2,
    "./qdora_adapter"
) #2


model_with_adapter = PeftModel.from_pretrained(
    base_model2,
    "./qlora_adapter"
) #3
```

The base model is loaded once #1 and stays in memory for the entire session. The call to `PeftModel.from_pretrained()` wraps the base model with the specified adapter #2 #3 without modifying its original weights. LoRA and DoRA are compatible with this pattern because each PeftModel manages its own adapter architecture independently.

This architecture has a real operational cost: every adapter switch takes time. In production, switching adapters on each individual request can easily negate the performance benefit. The efficient strategy is to group requests by task type and process them in batches with the corresponding adapter active.

Figure 7.7 illustrates this pattern. A router classifies each incoming request by task type and queues it in the corresponding channel. The model processes the full batch of requests with one active adapter, then switches adapters and processes the next batch.

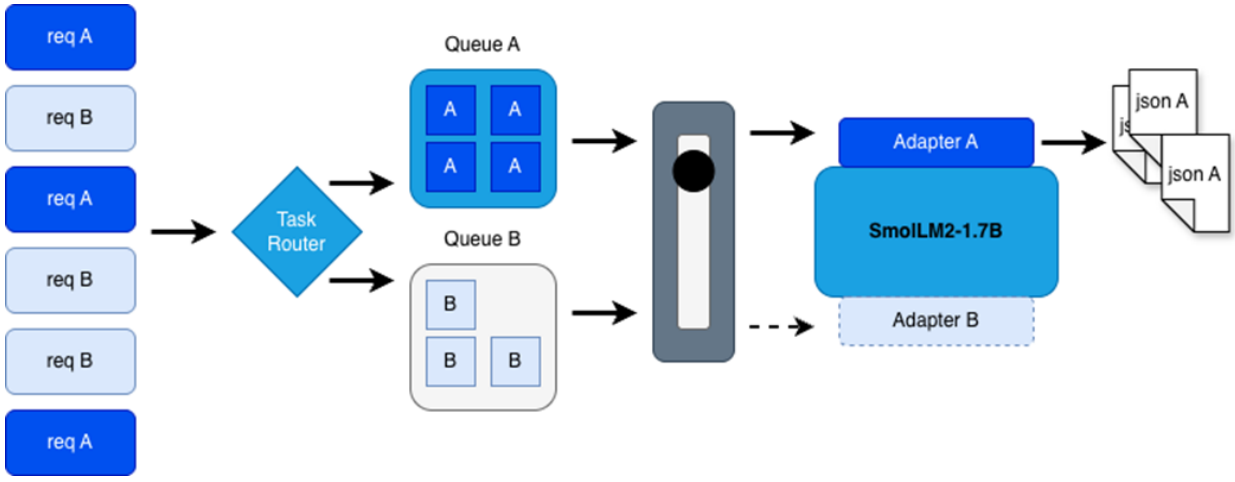


Figure 7.7 Adapter routing pattern for multiple specializations over a single base model. A task router classifies incoming requests and buffers them into separate queues. A switch selects which adapter is active: the model processes one full batch, swaps the adapter, and processes the next. The base model stays in memory throughout — only the lightweight adapter changes between batches. The dashed outline indicates the inactive adapter waiting for its turn.

The choice between merging and swappable adapters doesn't have a universal answer. It depends on how many specializations you need, how often you switch between them, and the hardware resources available. What both strategies do share is the starting point: a solid base model, a well-trained adapter, and rigorous measurement of the results. With that, you have everything you need to take a specialized model to production.

7.4 From paper to practice

The specialization work you've done in this chapter is rooted in four concrete studies. The first, "LoRA: Low-Rank Adaptation of Large Language Models" (Hu et al., Microsoft Research, 2021, https://arxiv.org/abs/2106.09685), is the mathematical foundation on which everything else is built. The second, "QLoRA: Efficient Finetuning of Quantized LLMs" (Dettmers et al., University of Washington, 2023, https://arxiv.org/abs/2305.14314), is the logistical bridge to consumer hardware. The third, "DoRA: Weight-Decomposed Low-Rank Adaptation" (Liu et al., arxiv:2402.09353), is the one that introduced the separation between magnitude and direction. The fourth, "Fine-Tuning LLMs on Small Medical Datasets" (Losch et al., University of Münster, 2025, arxiv:2503.21349), works in a domain almost identical to ours and offers independent validation of the decisions you made in the experiment design.

The foundational LoRA paper establishes that computing a small directional delta is enough to adapt a model to a new task. This intuition is what justifies that, in listing 7.10, you were able to completely restructure the model's output by training just 0.53% of its parameters. However, LoRA alone doesn't address the high memory consumption of the base model. QLoRA solves this by introducing 4-bit NormalFloat quantization, optimized for the natural distribution of LLM

weights. This paper demonstrated that you can fine-tune a 65B model on a 48GB GPU, and that's exactly why you set `bnb_4bit_quant_type="nf4"` in listing 7.5.

DoRA is an evolution that emerged from an observation: when performing full fine-tuning, the magnitude and direction of the weights are updated almost independently, with a correlation of -0.62 between them. LoRA, on the other hand, drags them together almost always, with a correlation of +0.83. That means LoRA can't do something that full fine-tuning can. It can increase the magnitude a lot while moving the direction very little, or the other way around. DoRA solves this by explicitly decomposing both components, just as you did in listing 7.3, and training them separately, achieving a correlation of -0.31, much closer to the behavior of full fine-tuning.

But after explaining these differences, you might wonder why the results between QLoRa and QDoRA have been almost identical. If we study the results in the paper where DoRA and LoRA are compared on a Llama-7B model, DoRA's results are much better using a rank of 8 (exactly the one used in our experiment). We need to revisit the paper to understand the nature of the experiments performed.

The experiments in the paper validate DoRA on tasks that require deep semantic adaptation: common-sense reasoning, visual understanding, and multimodal instruction. Our experiment, however, focuses solely on structural adaptation. It doesn't generate the semantic variation needed for the magnitude-direction separation to provide a measurable advantage. DoRA is designed for tasks where the model needs to change what something means; in our extraction task, the model only needs to change how it says it.

The last paper is a study published in March 2025. "Fine-Tuning LLMs on Small Medical Datasets" (Losch et al.) works in territory almost identical to ours. It extracts clinical entities from free medical text with output in structured JSON format. The authors show that fine-tuning with only 300 examples allows a LLaMA3-8B to match the performance of a LLaMA3-70B in zero-shot, and that the number of parsing errors drops from 17 to 1 as the model specializes. This second data point directly supports what we saw in section 7.3.2. The untrained base model fails on the irrelevant category with 50% schema compliance; the fine-tuned model reaches 100% across all categories.

The knowledge that previously lived in the prompt now lives in the weights, and that translates into structural reliability. This isn't a result exclusive to our experiment but a documented pattern in the literature.

Top-tier papers offer you sophisticated techniques like DoRA, but it's your problem constraints and your own data that dictate whether you actually need that complexity. At the same time, applied research like the clinical study confirms that you don't need a big lab's budget to solve real problems. If you understand the theoretical foundation behind the weights and measure your results rigorously, you can build highly specialized, production-ready models using the hardware you already have available.

7.5 Hands-on lab

In this chapter, you've used the notebook CH07_NB02_L4_QLoRA_QDoRA.ipynb, which uses a very standard LoRA configuration, with an `r` of 8 and trains the model for three epochs. Now you'll use a new notebook with a much more aggressive configuration: `r` of 1, a single epoch, and a much larger lr. Schema compliance stays at

100% with both techniques, but with better preservation of the model's capabilities. The three exercises that follow build on that foundation and put you in front of decisions any engineer faces when taking a fine-tuning experiment to production.

- The data efficiency frontier:
  - What to do: In the real world, you'll rarely have the luxury of thousands of perfectly annotated examples. Using the NB02 notebook with the standard configuration (r=8, 3 epochs), train different versions of the model with subsets of the training dataset ranging from 250 down to 30 samples. Keep the test set intact so results stay comparable.
  - Questions to answer: At what point does schema compliance collapse? Is it gradual, or is there a sharp threshold? Which dataset category fails first: irrelevant, typos, or implicit? How many samples do you need to recover 100% compliance across all categories?
- Target module selection:
  - What to do: Train three QLoRA variants (r=8, 3 epochs), changing only the target_modules parameter: first the full Attention Module only (`q_proj`, `k_proj`, `v_proj`, `o_proj`), then the full MLP Module only (`gate_proj`, `up_proj`, `down_proj`), and finally only query and value matrices (`q_proj`, `v_proj`), which is the minimal configuration from the original LoRA paper.
  - Questions to answer: Does any variant reach 100% schema compliance? Which module has more impact on the task — attention or MLP? Is there a linear relationship between the number of trainable

parameters and schema compliance, or do you observe a point of diminishing returns?

- Finding a task where DoRA wins:
  - What to do: Let's put the hypothesis from section 7.4 to the test. Replace the clinical dataset with one that requires semantic adaptation or reasoning; for example, a subset of databricks-dolly-15k. Keep the rest of the code and hyperparameters from this notebook intact.
  - Questions to answer: By demanding semantic variation from the model instead of deterministic extraction, does the technical tie between both techniques break? Does QDoRA manage to better retain general capability on the benchmarks, or converge more stably than QLoRA during training?

7.6 Summary

- You can move behavioral instructions from the prompt into the model's weights through fine-tuning, saving tokens, memory, and latency. It lets you replace long, complex prompts with single-word triggers.
- Parameter-efficient fine-tuning (PEFT) drastically reduces training costs. By freezing the base model and learning only a low-rank decomposition of weight updates, you can adapt model behavior by training less than 1% of its parameters.
- DoRA isolates the magnitude and direction of model weights during training. This separation prevents capability degradation when the target dataset requires deep semantic adaptation, although its benefits can be marginal in purely structural tasks.
- 4-bit NormalFloat quantization (NF4) solves the memory bottleneck of keeping the base model in the GPU, freeing

up space for gradients and optimizer states on a consumer GPU.

- Establishing a general capabilities baseline using standard benchmarks is necessary before training to be able to measure catastrophic forgetting. A successful specialization radically improves reliability in the target domain while causing only a marginal and acceptable drop in general knowledge.
- Once trained, the adapter can be permanently merged with the base model using `merge_and_unload()` to get a standard model with no PEFT dependencies, or kept separate and loaded dynamically with `PeftModel.from_pretrained()` when multiple specializations on the same base model are needed.
- Specializing models doesn't require massive budgets or millions of data points. It's possible to build reliable, domain-specific extractors using small synthetic datasets and accessible hardware.

8 Attention optimization

This chapter covers

- Quantizing the KV cache for longer contexts
- Scoring attention layers with cosine similarity
- Removing attention modules from the architecture
- Modifying model execution flow with custom classes
- Persisting and reloading modified models

Throughout the book we've worked on modifying model weights, either by removing them through different forms of pruning or by modifying them through Knowledge Distillation or fine-tuning.

In this chapter, we're going to focus on the attention mechanism, the main bottleneck in LLMs during inference. Both its computational demand and its dynamic memory consumption grow as context length increases, limiting the amount of data and the maximum batch size the model can process.

We'll explore two ways to optimize the attention mechanism. The first, purely practical, is KV cache quantization. We'll look at how to apply it with Hugging Face and with vLLM, and what the results tell us about when it makes sense to use each library.

The second part has a much more direct connection to recent research. In it, we'll analyze the redundancy of Attention Modules and apply tensor similarity-based techniques, as you already saw in the depth pruning chapter,

to remove those attention layers that don't contribute value to the output.

8.1 KV cache fundamentals and Hugging Face implementation

You've used quantization to train a model with QLoRA and QDoRA (Chapter 7), reducing the precision of the model's weights, and therefore its size.

Now, using the same principle, we're going to quantize the KV cache, the memory that grows with the context. This will allow dynamic memory consumption to be lower and the model to process larger prompts without running out of GPU memory.

We'll use two different inference engines: the Hugging Face API, accessible and well integrated into the Transformer’s ecosystem, and vLLM, an engine oriented toward production environments that manages the KV cache differently. We'll also work with two GPU families, the T4, which will be our main reference, and the more advanced L4, where we'll contrast some key results. As we'll see, the hardware choice is not a minor detail. The same quantization type can behave in a completely different way depending on the GPU architecture.

8.1.1 What the KV cache is and why it matters

In Chapter 3, we studied the role of the attention mechanism and how architectures like GQA (Grouped-Query Attention) mitigate KV cache growth by sharing attention heads. We also saw that disabling this cache is unviable, since forcing the model to recalculate tensors for every

token degrades inference speed by collapsing the GPU's processing capacity.

KV cache tensors are stored in FP16 or BF16 by default, depending on the GPU architecture. Just like with model weights, we can use quantization to reduce their precision and store them in smaller formats to minimize dynamic memory consumption.

The inference engine intercepts the K and V tensors generated by the Attention Module, converts them from their original 16-bit precision (2 bytes) to compressed formats like FP8 (1 byte) or 4-bit formats (0.5 bytes) before writing them to memory, and dequantizes them at the moment they're crossed with a new Query.

Figure 8.1 illustrates this cycle: each K, V pair generated at step T is compressed before being written to memory and restored at step T+1 just before being crossed with the new Query.

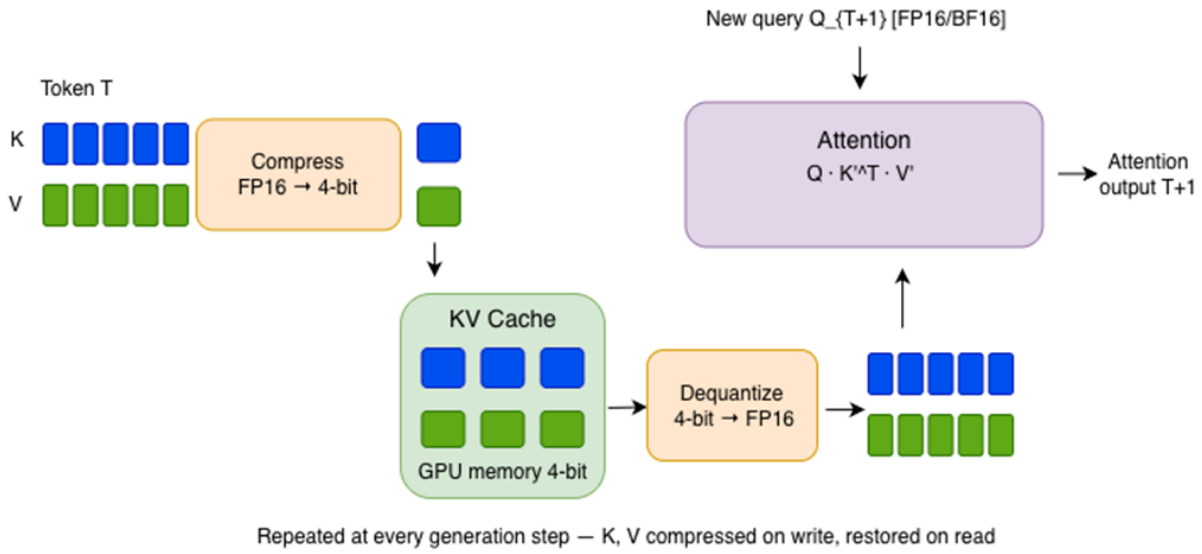


Figure 8.1 KV cache quantization write-read cycle. At each generation step, the K and V tensors produced by the Attention module are compressed from FP16 to 4-bit before being written to GPU memory. When the next token is processed, the cached values are dequantized back to FP16 before the attention computation. The cycle repeats for every generated token.

This memory savings comes at a cost in precision that you've already identified in Chapter 7. Also, depending on the inference library used and the GPU architecture, it can also lead to a performance penalty.

8.1.2 Hugging Face experiment setup

In this section we'll use two notebooks, CH08_NB01_L4_KVCache_HuggingFace.ipynb and CH08_NB01_T4_KVCache_HuggingFace.ipynb, both available in the book's repository (https://github.com/peremartra/Rearchitecting-LLMs/tree/main/CH08). The only difference between them is the GPU used. We'll use the notebook that runs on a T4 (Google Colab free tier) as the base for this section, but we'll discuss results from both.

The goal of this experiment is to demonstrate that weight quantization and KV cache quantization are two independent levers that act on different areas of GPU memory. To do this, we'll use a prompt that we'll run on the same model loaded in three different ways:

- **Base model:** Weights and KV cache in original precision (FP16 for T4, BF16 for L4).
- **Weight quantized:** Weights at 4-bit using BitsAndBytes, with the KV cache in original precision.
- **KV cache quantized:** Weights in original precision, but with the KV cache compressed to 4-bit using the HQQ backend.

For each model, we'll take three measurements: the static RAM used by the model, the dynamic RAM growth, and the throughput.

Let's focus on the code needed to load the model with a quantized KV cache. For its implementation we need a specialized backend that manages tensor compression. The transformers API supports integrations with several libraries for this purpose, with quanto and HQQ being the main options.

NOTE

quanto is no longer distributed as a standalone package. It has been integrated into Hugging Face's Optimum library, which may introduce additional compatibility requirements depending on your environment.

We'll use HQQ, though in the notebook you can find the code to use quanto as well. HQQ shows better performance and

has no compatibility issues with older Turing architecture GPUs like the T4.

The first step is to load the HQQ (https://github.com/dropbox/hqq) library and prepare the environment to load the model with the compressed KV cache.

Listing 8.1 Loading HQQ

```
!pip install -q transformers==4.51.3
!pip install -q hqq==0.2.8.post1

from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig,
    QuantizedCacheConfig,
)

!wget -q https://raw.githubusercontent.com
➥/peremartra/Rearchitecting-LLMs/main/utils.py
from utils import measure_memory_allocation #1

MODEL_NAME = "meta-llama/Llama-3.2-3B"

# Long prompt to make KV cache growth visible in memory measurements
PROMPT = (
    "Explain in detail the history of artificial intelligence, "
    "covering all major milestones from the 1950s to the present da
y."
) #A
MAX_NEW_TOKENS = 500
```

#A A long 500 token generation expands the KV cache, making the quantization effect measurable.

The `measure_memory_allocation` function #1, available in the `__utils.py__` file in the root directory of the book's repository, accepts an optional cache configuration. When provided, it

translates it into the arguments that `model.generate()` needs to enable KV cache compression during generation:

Listing 8.2 measure_memory_allocation function

```
def measure_memory_allocation(model, tokenizer, prompt,
    max_new_tokens=100,
    cache_config=None,
    cache_implementation=None):

    device = next(model.parameters()).device

    inputs = tokenizer(prompt, return_tensors="pt").to(device)
    input_len = inputs["input_ids"].shape[1]

    if torch.cuda.is_available():
        torch.cuda.synchronize()
        torch.cuda.reset_peak_memory_stats()
        static_vram = torch.cuda.memory_allocated() / (1024 ** 2)
    else:
        static_vram = 0.0

    generate_kwargs = dict(
        **inputs,
        max_new_tokens=max_new_tokens,
        do_sample=False,
        pad_token_id=tokenizer.eos_token_id,
    ) #1

    if cache_config is not None:
        generate_kwargs["cache_config"] = cache_config
        generate_kwargs["cache_implementation"]
➥= cache_implementation #2

    start_time = time.time()
    with torch.no_grad():
        outputs = model.generate(**generate_kwargs) #3

    torch.cuda.synchronize()
    elapsed = time.time() - start_time

    peak_vram = torch.cuda.max_memory_allocated() / (1024 ** 2)

    num_new_tokens = outputs.shape[1] - input_len
    throughput = num_new_tokens / elapsed if elapsed > 0 else 0.0
```

```
    generated_text = tokenizer.decode(
        outputs[0][input_len:],
        skip_special_tokens=True)

    return {
        "static_vram_mb": round(static_vram, 2),
        "dynamic_delta_mb": round(peak_vram - static_vram, 2),
        "throughput_tokens_s": round(throughput, 2),
        "generated_text": generated_text,
    }
```

The generation kwargs are built dynamically in #1 so the same code works for all three experiments in the notebook. When `cache_config` is `None`, block #2 is skipped and `generate()` uses the standard KV cache at the model's precision. If a configuration is passed, `cache_implementation="quantized"` tells Transformers to instantiate a `QuantizedCache` instead of the default `DynamicCache`; without this argument, `cache_config` would be silently ignored.

Quantization happens inside `model.generate()` #3, token by token. Each time the Attention Module generates new `K` and `V` tensors, the HQQ backend compresses them to 4 bits before writing to memory, and restores them to the original format before crossing them with the next Query

With the measurement function ready, we move on to setting up the configuration for quantizing the KV cache.

Listing 8.3 cache KV configuration

```
model_kvcache = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    torch_dtype=torch.float16, #1
    device_map="auto"
)

cache_config = QuantizedCacheConfig(  #2
    backend="HQQ",
    nbits=4,
    device="cuda:0"
)

results_kvcache = measure_memory_allocation(
    model_kvcache, tokenizer, PROMPT,
    max_new_tokens=MAX_NEW_TOKENS,
    cache_config=cache_config,
    cache_implementation="quantized"
)
```

The type of precision differs depending on the GPU #1, `float16` for T4 and `bfloat16` for L4, while static VRAM doesn't change. The only difference is `QuantizedCacheConfig` #2, where you specify the parameters that will be used in the `generate()` call. We'll use the HQQ algorithm to compress the Key and Value tensors to 4 bits, and load them onto the GPU.

8.1.3 Hugging Face results analysis

With the three configurations loaded and measured, we can now compare how each quantization strategy affects static VRAM, dynamic memory growth, and throughput.

Figure 8.2 shows the results of the T4 notebook.

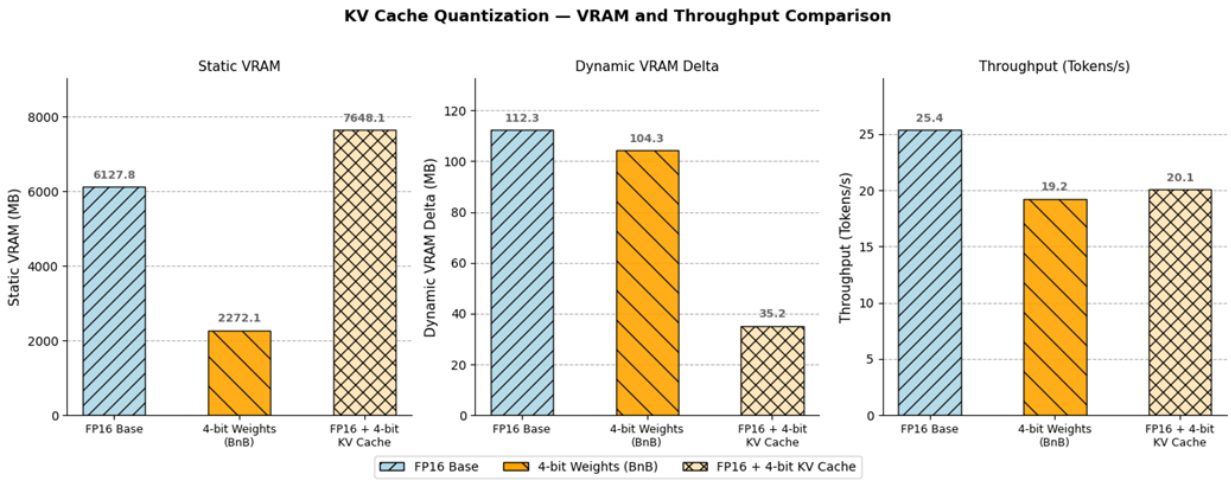


Figure 8.2 Three bar charts comparing static VRAM, dynamic VRAM delta, and throughput across three inference configurations of Llama-3.2-3B on a T4 GPU: FP16 baseline, 4-bit weight quantization (BitsAndBytes), and FP16 with 4-bit KV cache quantization (HQQ).

The static VRAM behavior is counterintuitive, as we observe an increase in model consumption with quantized KV cache, even though the model weights are untouched. This behavior is due to the HQQ kernel; it's a one-time memory reservation that will be offset by the lower dynamic VRAM consumption. The model quantized to four-bit precision cuts static VRAM usage from 6.1 GB to 2.2 GB, a nearly 3x reduction.

With Dynamic VRAM the result is different; you can see how the model using the quantized KV Cache shows around a 3x reduction compared to the other two models.

The throughput shows how the base model delivers the best performance, while the other two experience a slight drop.

The results with an L4 GPU, shown in figure 8.3, are very similar to those obtained with the T4 GPU, but with two differences worth mentioning.

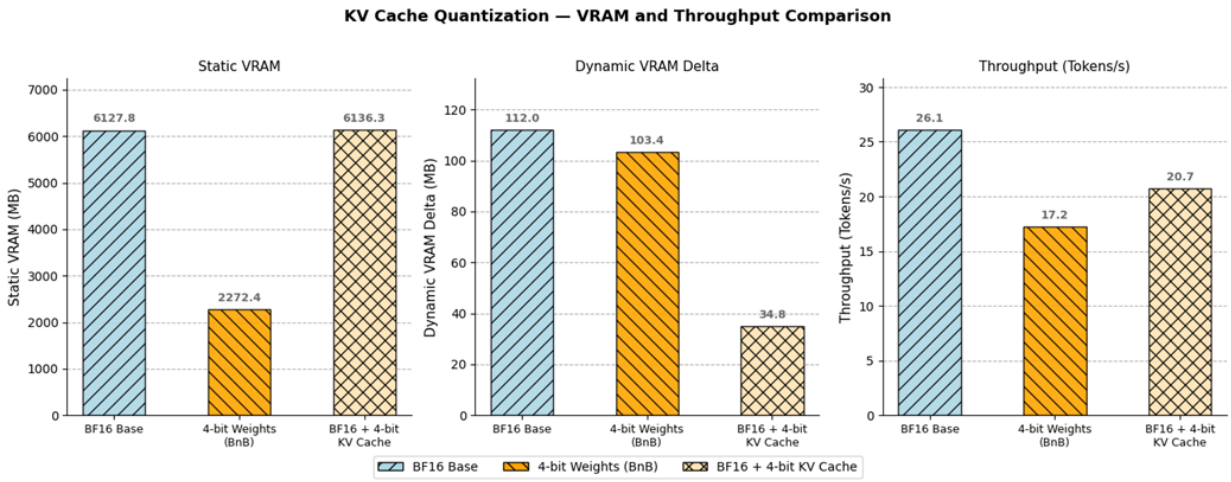


Figure 8.3 Three bar charts comparing static VRAM, dynamic VRAM delta, and throughput across three inference configurations of Llama-3.2-3B on an L4 GPU: BF16 baseline, 4-bit weight quantization (BitsAndBytes), and BF16 with 4-bit KV cache quantization (HQQ).

First, we can see that on the L4 GPU there's no increase in memory consumption with the KV-quantized model. It uses the same amount of memory, which confirms that the increase we saw on the T4 was coming from the HQQ kernel.

Dynamic VRAM behavior stays consistent with what we observed on the T4. Interestingly, the quantized-weights model actually performs worse on the more modern and powerful L4 GPU than on the T4. This is likely because `BitsAndBytes` doesn't leverage the GPU's native quantization capabilities.

In the next section we'll see how these models behave with a more optimized inference library like vLLM. As you've already seen, quantization comes with a quality loss that's easy to spot if you look at the `generated_text` field returned by the `measure_memory_allocation` function.

Let's look at the first 100 characters of each response:

```
Base:
"What are the key challenges and opportunities for AI in the future?
 What are the ethical implications of AI?..."

4-bit weights (BnB):
"What are the major milestones in the history of artificial intellig
ence?
 What are the major milestones in..."

4-bit KV cache (HQQ):
"What are the key challenges and opportunities for AI in the future?
 Artificial intelligence
 (AI) is a field..."
```

The quantized-weights model enters a repetition loop from the first generated token and never exits it within the 500-token limit. The base model and the quantized KV cache model produce coherent, structurally similar responses. All three share the same prompt and the same generation limit; the only difference is where quantization is applied. Smaller models like this 3B have less parameter redundancy and are highly sensitive to weight quantization. The BitsAndBytes method compresses all weights equally without checking which ones are the most important (the outliers). Quantization algorithms like AWQ or GPTQ, implemented in libraries like AutoAWQ and AutoGPTQ, include a preliminary calibration step that identifies and protects these key weights before compressing them. Without it, the model loses critical information, leading to the immediate degradation we see in the generated text. KV cache quantization, since it doesn't touch the weights, doesn't introduce this problem.

In the next section, we replicate the experiments using the same infrastructure but with the vLLM inference engine.

8.2 Scaling KV cache for production with vLLM

The main difference between vLLM and the Hugging Face libraries lies in memory management. vLLM uses the PagedAttention technique and pre-allocates a fixed percentage of the available VRAM during engine startup. Because of this pre-allocation, any VRAM memory we save from the model weights automatically turns into a larger capacity of available KV tokens.

When we quantize the KV cache, the relevant metric becomes the total number of KV tokens the engine can handle. The more blocks that fit in memory, the greater the context capacity and batch size the server will be able to support. Figure 8.4 shows how each strategy acts on vLLM's VRAM budget.

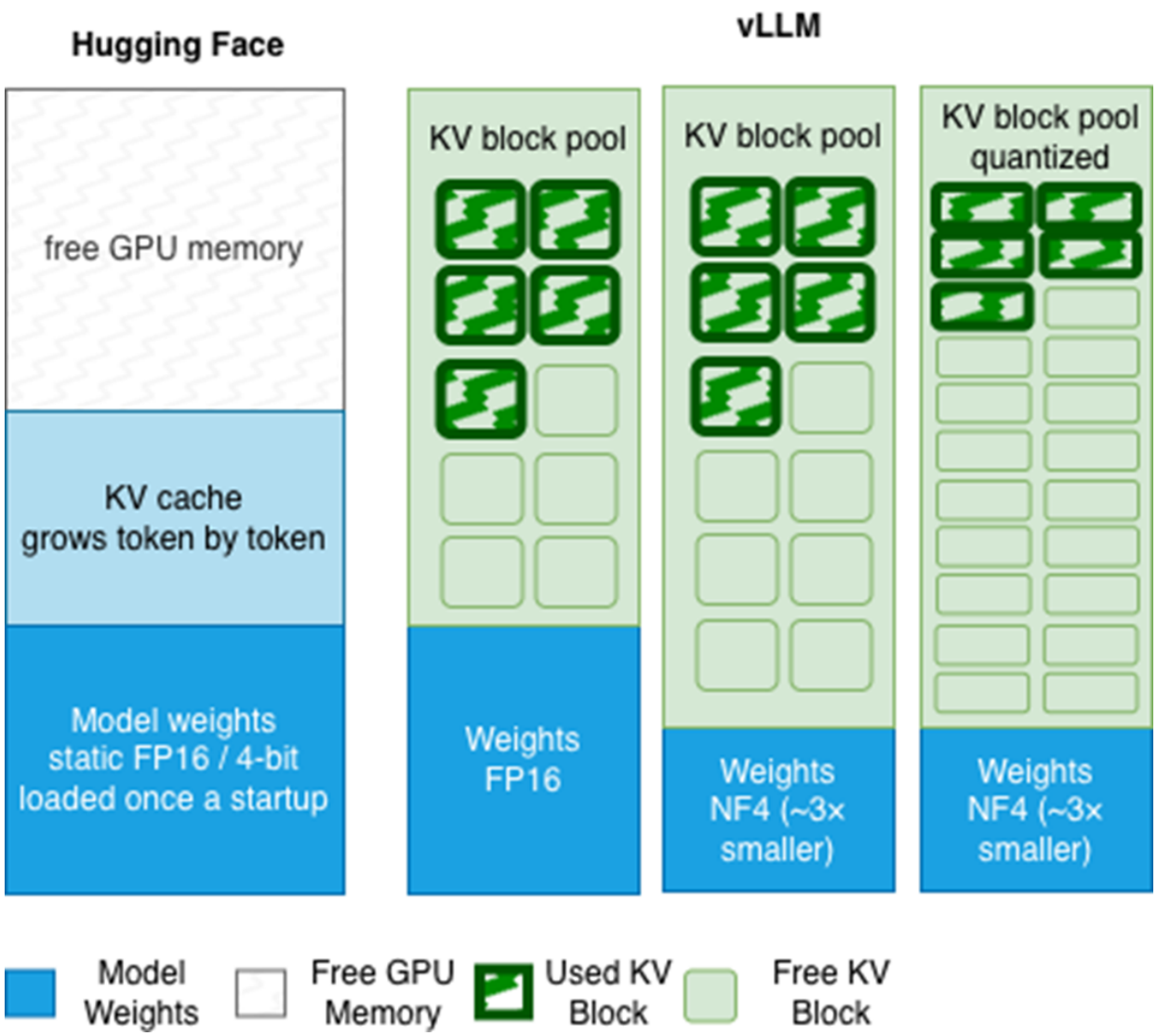


Figure 8.4 VRAM allocation across inference engines and quantization strategies. In Hugging Face, the KV cache grows dynamically into available memory. In vLLM, a fixed memory budget is divided at startup between model weights and a pre-allocated KV block pool. Reducing weight precision to NF4 shrinks the weight footprint, expanding the KV block pool within the same budget. Quantizing the KV blocks themselves reduces each block's size, fitting more tokens into the same pool.

For this experiment, we'll use the notebooks CH08_NB02_T4_KVCache_vLLM and CH08_NB02_L4_KVCache_vLLM, available in the book's repository (https://github.com/peremartra/Rearchitecting-LLMs/tree/main/CH08). It's basically the same experiment

adapted to the two GPUs. In this section we discuss the results of both.

NOTE

vLLM's calls `fileno()` on stdout and stderr during initialization. In Jupyter and Colab, these streams are custom objects that do not support `fileno()`, causing an `UnsupportedOperation` error. The cleanest workaround is to run vLLM in a standalone Python process where stdout and stderr are real file descriptors. The notebook writes a self-contained script to disk and launches it from the cell, collecting results as a JSON line.

8.2.1 vLLM experiment configuration

The `run_vllm_test.py` script accepts two optional flags: `-fp8` to enable KV cache quantization, and `-nf4` to quantize the model weights. Internally, it captures the KV tokens metric from the vLLM initialization log through a custom logging handler.

Unlike the previous experiment with Hugging Face, here you don't reduce the cache to 4 bits. vLLM is focused on maximizing throughput in production and the decision has a purely hardware-based rationale. The byte is the smallest addressable unit on the GPU (8 bits). Working with 4-bit tensors forces the GPU to execute additional calculations on every read, penalizing latency.

Listing 8.4 Schema validation

```
%%writefile /content/run_vllm_test.py
import os, re, gc, sys, json, time, logging, argparse

os.environ["VLLM_ENABLE_V1_MULTIPROCESSING"] = "0" #A
from vllm import LLM, SamplingParams

MODEL_NAME = "meta-llama/Llama-3.2-3B"
PROMPT = (
    "Explain in detail the history of artificial intelligence, "
    "covering all major milestones from the 1950s to the present da
y."
)
MAX_NEW_TOKENS = 500

class _GpuBlockHandler(logging.Handler): #B
    def __init__(self):
        super().__init__()
        self.gpu_blocks = None
        self.kv_tokens = None
    def emit(self, record):
        msg = record.getMessage()
        m = re.search(r'#\s*GPU blocks:\s*(\d+)', msg)
        if m:
            self.gpu_blocks = int(m.group(1))
        m2 = re.search(r'GPU KV cache size:\s*([\d,]+)\s*tokens', ms
g)
        if m2:
            self.kv_tokens = int(m2.group(1).replace(',', ''))

handler = _GpuBlockHandler()
logging.getLogger("vllm").addHandler(handler)

parser = argparse.ArgumentParser()
parser.add_argument("-fp8", action="store_true")
parser.add_argument("-nf4", action="store_true")
args = parser.parse_args()

llm_kwargs = dict(
    model=MODEL_NAME,
    dtype="float16",
    gpu_memory_utilization=0.85, #C
    max_model_len=4096,
```

```
)
if args.fp8:
    llm_kwargs["kv_cache_dtype"] = "fp8" #D
if args.nf4:
        llm_kwargs["quantization"] = "bitsandbytes"
        llm_kwargs["load_format"] = "bitsandbytes"
llm = LLM(**llm_kwargs)
```

#A Disables V1 engine multiprocessing to avoid the fileno() conflict in Jupyter.
#B Intercepts vLLM logs to extract the total number of KV tokens pre-allocated at startup.
#C Reserves 85% of VRAM for the model and the KV cache.
#D A single parameter converts the KV cache from FP16 (2 bytes per value) to FP8 (1 byte per value).

The script ends by launching generation, measuring throughput, and serializing the results so the notebook can retrieve them:

```
sampling_params = SamplingParams(temperature=0.0, max_tokens=MAX_NEW
_TOKENS)
start = time.perf_counter()
outputs = llm.generate([PROMPT], sampling_params)
elapsed = time.perf_counter() - start

output_text = outputs[0].outputs[0].text
n_tokens = len(outputs[0].outputs[0].token_ids)
throughput = round(n_tokens / elapsed, 2)

# --- Output JSON to stdout ---
result = {
    "gpu_blocks": handler.gpu_blocks,
    "kv_tokens": handler.kv_tokens,
    "throughput": throughput,
    "n_tokens": n_tokens,
    "elapsed": round(elapsed, 2),
    "text": output_text[:500],
}
print(f"###RESULT###{json.dumps(result)}###END###") #A

del llm #B
gc.collect()
```

#A Marks the result JSON so the notebook can extract it from vLLM's log output.
#B Releases the engine to free VRAM between experiments.

With the script on disk, running each configuration is reduced to a single line per experiment:

```
raw_fp16 = !python /content/run_vllm_test.py #A
result_fp16 = parse_result("\n".join(raw_fp16))

raw_nf4 = !python /content/run_vllm_test.py -nf4 #B
result_nf4 = parse_result("\n".join(raw_nf4))

raw_fp8 = !python /content/run_vllm_test.py -fp8 #C
result_fp8 = parse_result("\n".join(raw_fp8))
```

#A FP16 weights + FP16 KV cache
#B NF4 weights + FP16 KV cache
#C FP16 weights + FP8 KV cache

In the second experiment, quantizing the model weights releases static VRAM. Since vLLM pre-allocates a fixed percentage of the available VRAM (the 85% you configured in Listing 8.4), the memory not occupied by the model becomes available for the KV Cache.

8.2.2 vLLM results analysis

Unlike the Hugging Face measurements, which tracked memory in megabytes, vLLM exposes its efficiency through a different lens: the total number of KV tokens the engine can hold. That difference in metric reflects a difference in architecture, and it changes how we read the results below.

In Figure 8.5 we can find the results produced by running the experiments on a T4 GPU.

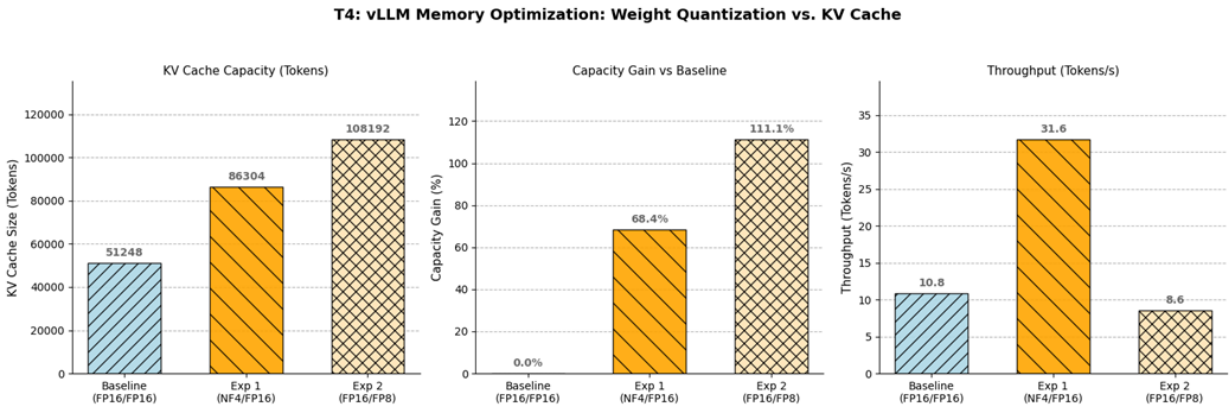


Figure 8.5 Comparison of the three configurations run on a T4 GPU: KV cache capacity in tokens (left), percentage gain over baseline (center), and generation throughput in tokens per second (right). On this GPU, FP8 delivers the highest capacity gain (+111.1%) but penalizes throughput, while NF4 triples the baseline throughput.

FP8 quantization of the KV cache more than doubles the available capacity: 108,192 tokens compared to the 51,248 of the baseline (+111.1%). The capacity gain is calculated simply as the percentage increase relative to the baseline: (kv_tokens_new / kv_tokens_baseline - 1) × 100. However,

throughput drops from 10.8 to 8.6 tokens/s compared to the base experiment. The T4 GPU has no native FP8 acceleration, so the cache has to be converted back to FP16 before each attention operation, and that conversion overhead penalizes generation speed.

The experiment with NF4 weights produces a different result; the capacity gain is more moderate (+68.4%), but throughput rises noticeably, to 31.64 tokens/s. Autoregressive decoding in LLMs is memory-bandwidth-bound, not compute-bound: at each step the model has to stream its weights from VRAM through the GPU's memory hierarchy to compute a single token. A model that occupies less memory moves fewer bytes per step, and that translates directly into higher throughput.

Figure 8.6 shows the result of running the three experiments on the L4 GPU.

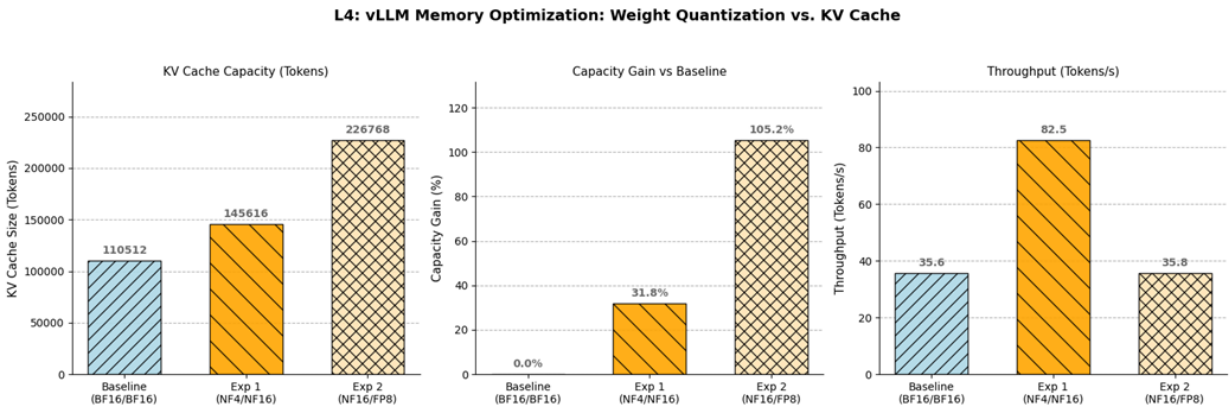


Figure 8.6 The same three experiments run on an L4 GPU. On this GPU with native FP8 support, KV cache quantization doubles the available capacity (+105.2%) while keeping throughput virtually identical to the baseline (35.8 vs 35.6 tok/s). NF4 weight quantization offers a different profile: lower capacity gain (+31.8%), but throughput more than twice the baseline (82.5 tok/s). Each technique serves a different goal.

FP8 quantization keeps doubling capacity (+105.2%, up to 226,768 tokens), and this time throughput stays practically identical to the baseline: 35.8 vs 35.6 tokens/s. The L4

handles FP8 with virtually no computational cost. The NF4 experiment offers a different profile: the capacity gain is more modest (+31.8%), but throughput doubles compared to the baseline, reaching 82.5 tokens/s.

No technique is universally better, and the choice depends on the goal and the hardware available. On a modern GPU with native FP8 support, quantizing the KV cache doubles context capacity at no speed cost. Quantizing weights to NF4 is the right choice when the main goal is throughput. On older hardware like the T4, the priorities flip. FP8 still offers the largest KV cache capacity gain, but you have to accept a speed penalty that NF4 doesn't have.

It's also worth looking at the generated text. The baseline and FP8 configuration produce coherent responses on both GPUs. The NF4 experiment, on the other hand, falls into a repetition loop in both environments. The model generates the same sentence indefinitely; a clear sign of quality degradation. Aggressive weight quantization affects generation quality. It's a reminder that capacity and speed metrics only tell part of the story.

So far we've optimized the KV cache without touching the architecture. In the next section, we'll apply recent research to identify and eliminate attention redundancy in modern Transformers.

8.3 Measuring attention redundancy

In Chapter 4, when applying depth pruning to the model, you already used PyTorch hooks to study how transformer blocks contribute to data transformation using cosine distance between the block's input and output. Here you'll do the same thing, but measuring only the attention layer of the transformer block.

Instead of identifying transformer blocks that don't contribute, you'll identify attention layers, and when removing them you'll keep the MLP layer of the Transformer block.

The technique is inspired by the work of He et al. (2024), "What Matters in Transformers? Not All Attention is Needed" (https://arxiv.org/abs/2406.15786), which indicates that redundancy in a transformer block is not distributed uniformly within a block and that attention layers tend to be more dispensable and redundant than MLP layers.

The reference notebook for this section is CH08_NB03_Remove_Attention.ipynb (https://github.com/peremartra/Rearchitecting-LLMs/blob/main/CH08/CH08_NB03_Remove_Attention_.ipynb), which works with Llama-3.2-3B on a T4. We'll walk through the process in four steps: (1) we'll measure the contribution of each Attention Module with a calibration set; (2) we'll apply the physical removal to the model's architecture; (3) we'll see how to solve the compatibility issue that comes up when saving and reloading the modified model; (4) we'll evaluate the real impact on speed, memory, and generation quality.

8.3.1 Scoring with cosine similarity

In Chapter 4, you registered one hook per Transformer block, anchored directly at the `DecoderLayer`. That hook captured the tensor entering the full block and the one coming out after passing through the Attention Module and MLP Module together. The cosine similarity between those two tensors told you how much the block had transformed the information as a unit. Here we'll apply the same mathematical foundation, but adjusting the location of our PyTorch hooks to achieve greater granularity.

When capturing activations to assess layer importance in Chapter 4, you registered hooks at the full block level (`model.model.layers[i]`) to measure whether the Transformer block (Attention + MLP) was dispensable."

To measure exclusively the transformation of the attention phase, we take advantage of how the Hugging Face API standardizes access to the normalization components that encapsulate it. Regardless of whether we're working with Llama, Qwen, Gemma, or other LLMs, we'll inject our hooks at two key points in each block:

1. `input_layernorm`: Will give us the hidden state right before entering the attention projections.
2. `post_attention_layernorm`: Will give us the tensor processed by attention and added to its residual connection, right before entering the MLP module.

If you inspect the structure of a `LlamaDecoderLayer`:

```
(0-15): 16 x LlamaDecoderLayer(
    (self_attn): LlamaAttention(
       ...
    )
    (mlp): LlamaMLP(
        ...
    )
    (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
    (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
)
```

and that of a `Gemma3DecoderLayer`:

```
(0-15): 16 x Gemma3DecoderLayer(
    (self_attn): Gemma3Attention(
       ...
    )
    (mlp): Gemma3MLP(
        ...
    )
    (input_layernorm): Gemma3RMSNorm((640,), eps=1e-06)
    (post_attention_layernorm): Gemma3RMSNorm((640,), eps=1e-06)
    (pre_feedforward_layernorm): Gemma3RMSNorm((640,), eps=1e-06)
    (post_feedforward_layernorm): Gemma3RMSNorm((640,), eps=1e-06)
)
```

As you can see, both structures have the `input_layernorm` and `post_attention_layernorm` layers. This is a standardization followed by all decoder-only models from families like Llama, Mistral, Qwen, and Gemma, among others.

Gemma3 adds two additional LayerNorms that wrap its MLP Module, but `input_layernorm` and `post_attention_layernorm` are present and serve the same structural function as in Llama: they mark the input and output of the Attention Module including its residual. The hook code doesn't reference any specific architecture name, only these two attributes by name, which makes it directly applicable to any model that exposes them.

CALIBRATION DATA

Layer importance scores are not absolute properties; they depend on the type of text the model processes during measurement. In this case, since the goal is not to specialize the model but to get a faster and more efficient general-purpose one, you won't use a specific dataset. To get representative scores, the notebook builds a calibration dataset by sampling six subsets of Cosmopedia with different weights.

Listing 8.5 Calibration dataset composition

```
from datasets import load_dataset, Dataset
from torch.utils.data import DataLoader

BATCH_SIZE = 4
MAX_LENGTH = 2048
CALIBRATION_SAMPLES = 400

dataset_name = "HuggingFaceTB/cosmopedia"

subsets = [
    ("stories",        0.300),
    ("web_samples_v2", 0.200),
    ("web_samples_v1", 0.150),
    ("wikihow",        0.150),
    ("openstax",       0.125),
    ("stanford",       0.075),
]

all_samples = []
for subset, weight in subsets:
    n_samples = int(CALIBRATION_SAMPLES * weight) #A
    subset_data = load_dataset(
            dataset_name,
            subset,
            split="train",
            streaming=True)
    subset_samples = list(subset_data.take(n_samples))
    all_samples.extend(subset_samples)
    print(f"    Collected {len(subset_samples):,} samples")

calibration_dataset = Dataset.from_dict(
            {"text": [s["text"] for s in all_samples]}
            )
```

#A The number of samples per subset is calculated proportionally to the assigned weight, out of a total of CALIBRATION_SAMPLES.

The subset distribution follows a specific logic: we want the calibration data to cover the same range of capabilities that the evaluation benchmarks probe. `stories` and `web_samples`

provide the kind of narrative and general-domain text that supports the contextual reasoning evaluated by HellaSwag, Winogrande and Lambada. `wikihow`, with its procedural structure, maps naturally to the everyday physical reasoning that PIQA tests. `openstax` and `stanford` bring academic and textbook-style content closer to what ARC-Easy expects. The result is 400 samples that don't represent a single domain, but a deliberately balanced distribution of tasks.

The construction of the calibration dataset must respond to the task the model is being built for and the data it will work with.

REGISTERING HOOKS AND SCORING

The hooks are registered in the `setup_attention_hooks` function in Listing 8.6. Compare it with Listing 4.4 from Chapter 4; you'll see the overall structure is the same, with the necessary differences to change the element being measured.

Listing 8.6 Registering PyTorch hooks

```
import torch.nn.functional as F
import numpy as np
from tqdm import tqdm

def setup_attention_hooks(model):
    num_layers = len(model.model.layers)
    layer_inputs  = {}  #1
    layer_outputs = {}  #1
    hooks = []

    def make_input_hook(layer_idx): #2
        def hook(module, args):
            layer_inputs[layer_idx] = args[0].detach() #3
        return hook

    def make_output_hook(layer_idx): #2
        def hook(module, args):
            layer_outputs[layer_idx] = args[0].detach() #4
        return hook
   for i, layer in enumerate(model.model.layers):
       hooks.append(
           layer.input_layernorm.register_forward_pre_hook(
               make_input_hook(i) #5
           )
        )
       hooks.append(
           layer.post_attention_layernorm.register_forward_pre_hook(
               make_output_hook(i) #5
            )
        )

   return hooks, layer_inputs, layer_outputs, num_layers
```

Instead of a single hook per block like in Chapter 4, here we register two. The dictionaries `layer_inputs` and `layer_outputs` #1 store the captured tensors indexed by layer number, just as in Chapter 4.

The difference is that two separate factory functions #2 are used instead of just one, because now the capture points are

different modules. Figure 8.7 shows where the hooks are placed.

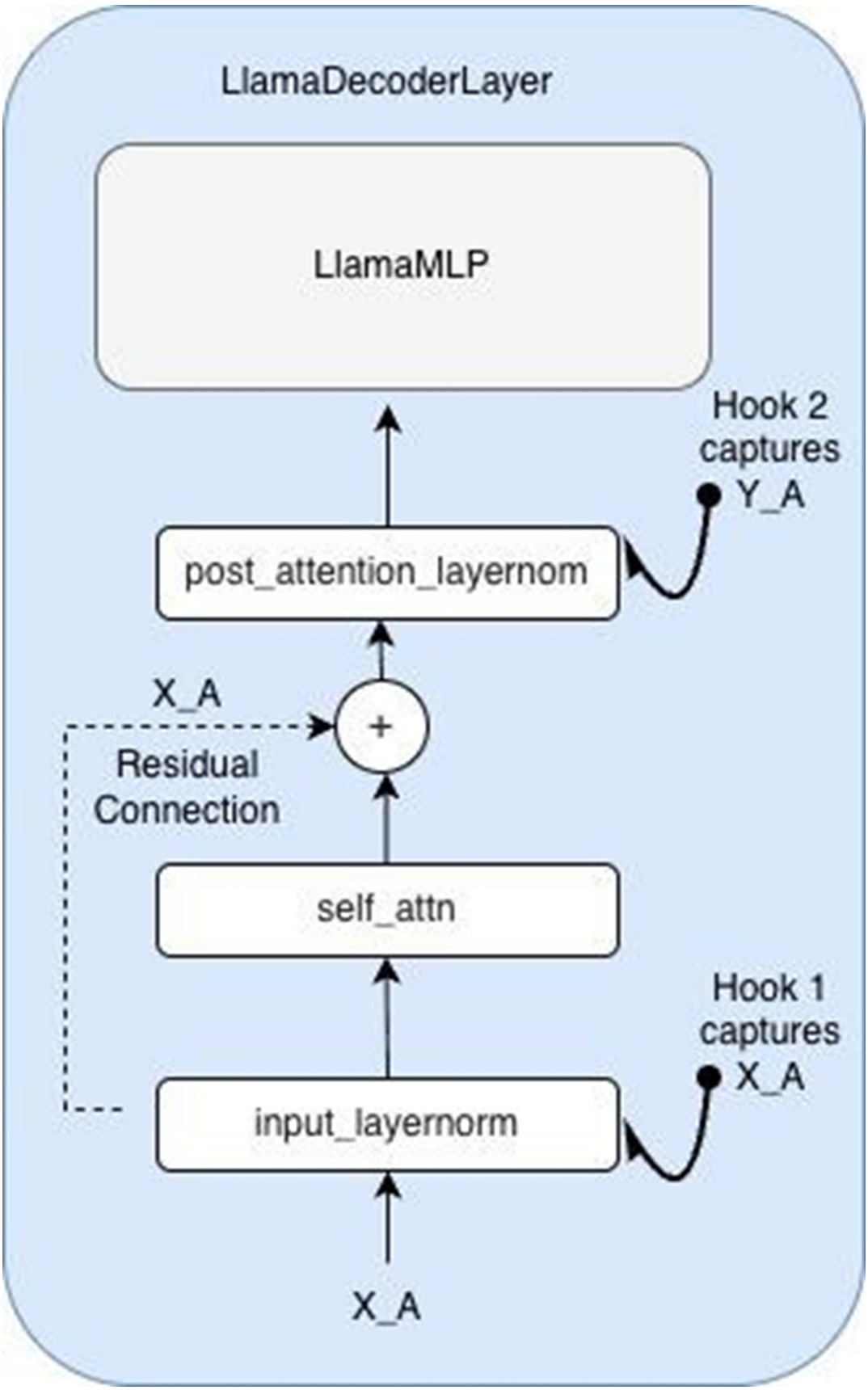

LlamaDecoderLayer
LlamaMLP
Hook 2
captures
Y_A
post_attention_layernom
X_A
+
Residual
Connection
self_attn
Hook 1
captures
X_A
input_layernorm
X_A

Figure 8.7 Internal structure of a LlamaDecoderLayer showing the two hook anchor points used to measure attention layer importance. Hook 1 intercepts the hidden state at input_layernorm, capturing X_A before the attention computation. Hook 2 intercepts the tensor at post_attention_layernorm, capturing Y_A after the attention output has been added back through the residual connection.

To understand what we're capturing, we'll call X_A the original input vector of the block and Y_A the vector after attention has been applied and added back through the residual connection.

- The first hook stores in `layer_inputs` the tensor `args[0]` that arrives at `input_layernorm` #3. This is exactly X_A, the hidden state before any transformation.
- The second hook stores in `layer_outputs` the tensor that arrives at `post_attention_layernorm` #4. This is Y_A. At this point the architecture has already computed the attention output and added it to the original input through the residual connection, so this tensor contains X_A + Attention(LayerNorm(X_A)), ready for the MLP phase.

Finally, both hooks are registered via `register_forward_pre_hook` #5, not with `register_forward_hook`. A pre-hook receives only the input arguments of a module before it executes, which is exactly the clean read we need at these two anchor points.

NOTE

Residual connections work by adding a block's input to its output before moving on. Instead of learning a direct transformation F(x), the block learns the difference F(x) - x, which stabilizes training in deep networks and helps gradients flow. In our case, this happens between the two points we're spying on: `input_layernorm` receives X_A and normalizes it to pass it to attention, but the original X_A travels along a parallel path and gets added to the attention output before that value reaches `post_attention_layernorm`. That's why Y_A already contains the residual by the time we capture it.

The `calculate_cosine_importance` function is identical to Listing 4.6 in chapter 4. It takes X_A and Y_A, computes the cosine similarity along the hidden state dimension for each token, applies the attention mask to exclude padding tokens, and returns 1 minus the mean similarity across valid tokens as the importance score.

The `calculate_attention_importance` function orchestrates the full process of evaluating attention layers.

Listing 8.7 Calculating the importance of Attention Modules

```
def calculate_attention_importance(model, dataloader, device):

    hooks, layer_inputs, layer_outputs, num_layers =
➥setup_attention_hooks(model) #1
    accumulated_scores = {i: [] for i in range(num_layers)}

    model.eval()
    with torch.no_grad():
        for batch_idx, batch in enumerate(
               tqdm(dataloader, desc="Scoring attention layers")):
            inputs = {k: v.to(device) for k, v in batch.items()}
            model(**inputs) #A

            for layer_idx in range(num_layers):
                score = calculate_cosine_importance(
                    layer_inputs[layer_idx],
                    layer_outputs[layer_idx],
                    batch["attention_mask"],
                    layer_idx,
                    is_first_batch=(batch_idx == 0)
                )
                accumulated_scores[layer_idx].append(score)

            layer_inputs.clear() #B
            layer_outputs.clear() #B

    for hook in hooks:
        hook.remove() #2

    final_scores = {}
    for layer_idx, scores in accumulated_scores.items():
        valid = [s for s in scores if np.isfinite(s)]
        final_scores[layer_idx] = np.mean(valid) #3

    return final_scores
```

#A A single forward pass per batch is enough, the hooks capture X_A and Y_A across all blocks at once.
#B The dictionaries are cleared after each batch to free GPU memory.

The function starts by calling `setup_attention_hooks` #1, which registers the pre-hooks on all blocks and returns the capture

dictionaries. From that point on, every forward pass updates `layer_inputs` and `layer_outputs`.

Once the loop is done, the hooks are removed #2, so they don't interfere with subsequent forward passes, including the evaluation of the pruned model.

The final score for each layer #3 is the average across all batches, which smooths out the dependence on any individual sample from the calibration dataset.

8.3.2 Analyzing results

Running the function gives us the importance scores for each Attention Module. Figure 8.8 makes the pattern visible in a way the numerical table can't convey.

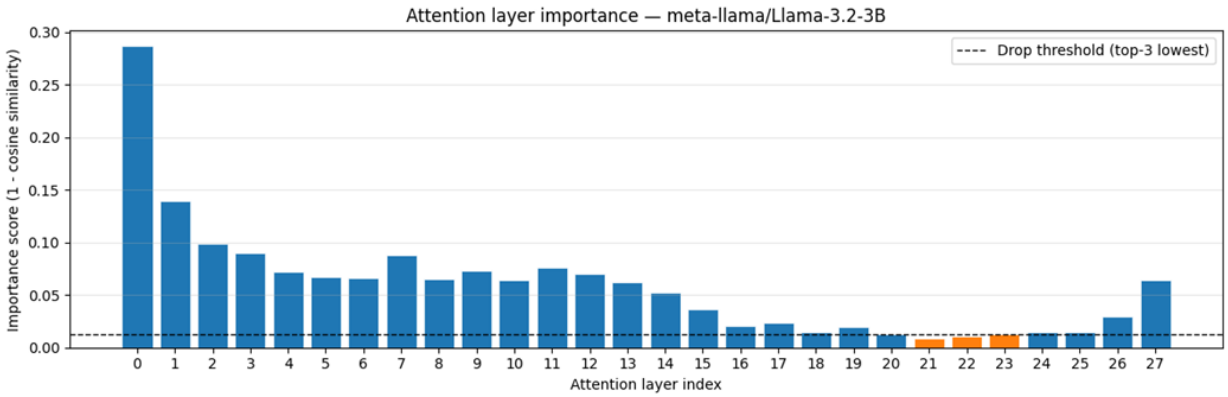


Figure 8.8 Attention Module importance in Llama-3.2-3B, sorted by layer index. Layers 0 and 1 stand clearly above the rest. Layers 2 through 17 form a moderately active band. The genuinely low-score region concentrates in layers 18 to 25, where the three removal candidates (marked with a different pattern) fall below the dashed threshold line. Layers 26 and 27 show a slight recovery..

The initial layers build the fundamental representations from the input tokens, which explains why they're the most active ones. From layer 2 onward, scores drop sharply and remain at moderate levels through layer 17. Then a genuinely low-score region emerges in the latter half, roughly layers 18 to

25, where all the pruning candidates fall. The final layers recover slightly, consistent with their role in consolidating representations before the language model head.

What's more interesting is comparing this pattern with one from a larger model. Table 8.1 shows the five lowest scores obtained with the same code, without modification, on Llama-3.1-8B running on an L4 GPU with quantization:

Table 8.1 Attention modules with the lowest importance score in Llama-3.2-3B and Llama-3.1-8B

Rank	Llama-3.2-3B	Score	Llama-3.1-8B	Score
1	Layer 21	0.009053	Layer 24	0.004922
2	Layer 22	0.010405	Layer 23	0.007695
3	Layer 23	0.012417	Layer 22	0.008828
4	Layer 20	0.012783	Layer 25	0.009258
5	Layer 18	0.014297	Layer 26	0.010508

The overall structure is consistent. In both models, the first and last layers remain the most active, and the redundancy concentrates in the region between them. More precisely, with the Cosmopedia dataset, the least important layers in both models fall between 60% and 80% of the network depth.

What changes with scale is the proportion. Applying a threshold of 0.015, roughly 18% of the 3B's layers fall below it (5 out of 28), while in the 8B that figure rises to 34% (11 out of 32). The redundancy space widens as the model grows, which is exactly what makes larger models more tolerant of attention pruning, as we'll see in section 8.5.1.

8.4 Removing attention modules

The score distribution we just visualized makes the selection immediate. As the paper proposes, you just sort by ascending score and take the first indices: the gap between layers 0 and 1 and the rest is so pronounced that no additional heuristic is needed.

Listing 8.8 Selecting Attention Modules to remove

```
def select_layers_to_drop(importance_scores, num_layers_to_drop):

    sorted_layers = sorted(importance_scores.items(), key=lambda x:
x[1])
    return [idx for idx, _ in sorted_layers[:num_layers_to_drop]]

layers_to_drop = select_layers_to_drop(attention_importance, 3)
print(f"Layers selected for removal: {sorted(layers_to_drop)}")
```

Layers are sorted by score in ascending order and the first three (21, 22, and 23) are selected for removal. The choice of removing three layers is conservative by design. As we'll see in section 8.4.2, adding just three more layers already produces visible degradation in short generation and immediate collapse in long-form output. With a 3B model the redundancy space is narrower than in larger architectures, and the transition from manageable degradation to generation collapse happens quickly. As detailed in section 8.5.1, the paper provides evidence showing that larger models tolerate significantly higher pruning ratios, which helps calibrate expectations for different model sizes.

To understand what we're about to remove, it's worth recalling the internal structure of `LlamaDecoderLayer`, which is similar across all modern LLMs.

```
(0-27): 28 x LlamaDecoderLayer(
    (self_attn): LlamaAttention(
      (q_proj): Linear(in_features=3072, out_features=3072, bias=Fal
se)
      (k_proj): Linear(in_features=3072, out_features=1024, bias=Fal
se)
      (v_proj): Linear(in_features=3072, out_features=1024, bias=Fal
se)
      (o_proj): Linear(in_features=3072, out_features=3072, bias=Fal
se)
    )
    (mlp): LlamaMLP(
       ...
    )
    (input_layernorm): LlamaRMSNorm((3072,), eps=1e-05)
    (post_attention_layernorm): LlamaRMSNorm((3072,), eps=1e-05)
)
```

Each block runs two phases in sequence, attention and MLP, each with its own LayerNorm preceding it. `input_layernorm` normalizes the hidden state before it enters `self_attn`; the two together form the attention phase. Then, `post_attention_layernorm` runs after attention and normalizes the result before it enters `mlp`; those two form the MLP phase. We're going to remove the first two, `input_layernorm` and `self_attn`, and leave the last two intact.

There's no point in removing `self_attn` without `input_layernorm`. A LayerNorm with nothing to feed into is an empty compute step that just burns cycles. We delete them together with `delattr`, which frees their parameters from VRAM immediately.

Listing 8.9 Removing the attention module

```
from typing import Optional, Tuple

def drop_attention_layer(model, layer_idx):
    layer = model.model.layers[layer_idx]

    delattr(layer, "self_attn") #A
    delattr(layer, "input_layernorm") #A

    def forward_no_attn( #1
        self,
        hidden_states: torch.Tensor,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        past_key_value=None,
        use_cache: Optional[bool] = False,
        cache_position: Optional[torch.LongTensor] = None,
        position_embeddings: Optional[
          Tuple[torch.Tensor, torch.Tensor]
          ] = None,
        **kwargs,
    ):

        residual = hidden_states
        hidden_states = self.post_attention_layernorm(hidden_states)
#2
        hidden_states = self.mlp(hidden_states) #2
        hidden_states = residual + hidden_states #2
        return hidden_states

    PrunedLayer = type(
        "PrunedDecoderLayer",
        (type(layer),),
        {"forward": forward_no_attn}
    )
    layer.__class__ = PrunedLayer #B
```

#A delattr removes the submodule and frees its parameters from VRAM immediately.
#B The patch is applied at the class level, not the instance level, to prevent transformers' internal mechanism from overwriting the method during generation.

The problem that appears after the delattr is that the original `forward()` method of the `DecoderLayer` still tries to call `self.input_layernorm` and `self.self_attn`. Every complex module has its own `forward()` that controls the internal flow (which we discuss in Chapter 3). The LlamaDecoderLayer's `forward()` calls the Attention Module and the MLP Module sequentially. Without a fix, the next forward pass would throw an AttributeError when trying to execute a submodule that no longer exists. That's why we redefine `forward()` with a version that skips the Attention Module entirely.

In the new `forward_no_attn` function there are no calls to `self.input_layernorm` or `self.self_attn`; they've simply disappeared. The tensor passes directly to `post_attention_layernorm` and `mlp` #2, which do their work while preserving the residual connection exactly as in the original forward.

To prevent Transformers from overwriting the function, we create the `PrunedDecodeLayer` subclass with `Type()` and inject the new `forward_no_attn` method into it.

To remove the layers identified as candidates, we iterate over the `layers_to_drop` list:

```
for idx in layers_to_drop:
    drop_attention_layer(model, idx)
    print(f"Dropped attention layer {idx}")
```

What it returns:

```
Dropped attention layer 21
Dropped attention layer 22
Dropped attention layer 23
```

A `print(model)` confirms that the modified layers no longer contain `self_attn` or `input_layernorm` and appear as

`PrunedDecoderLayer`:

```
...
      (21-23): 3 x PrunedDecoderLayer(
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=3072, out_features=8192, b
ias=False)
          (up_proj): Linear(in_features=3072, out_features=8192, bia
s=False)
          (down_proj): Linear(in_features=8192, out_features=3072, b
ias=False)
          (act_fn): SiLU()
        )
        (post_attention_layernorm): LlamaRMSNorm((3072,), eps=1e-05)
      )
      (24-27): 4 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
          (q_proj): Linear(in_features=3072, out_features=3072, bias
=False)
          (k_proj): Linear(in_features=3072, out_features=1024, bias
=False)
          (v_proj): Linear(in_features=3072, out_features=1024, bias
=False)
          (o_proj): Linear(in_features=3072, out_features=3072, bias
=False)
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=3072, out_features=8192, b
ias=False)
          (up_proj): Linear(in_features=3072, out_features=8192, bia
s=False)
          (down_proj): Linear(in_features=8192, out_features=3072, b
ias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm((3072,), eps=1e-05)
        (post_attention_layernorm): LlamaRMSNorm((3072,), eps=1e-05)
      )
...
```

Blocks 21-23 doesn't contain `self_attn` or `input_layernorm`, while the rest of the blocks shown keep the original structure

with the full attention module.

The model is functional right now, but it's not ready to be saved and reloaded, which is exactly the problem we address in the next section.

8.4.1 Structural implications and framework compatibility

To save the model and be able to reload it later, keep in mind that `save_pretrained` serializes the weights and the `config.json` file, but not the custom Python code with the new forward function. If you tried to reload the model with `AutoModelForCausalLM.from_pretrained`, Hugging Face would reconstruct the architecture using the original `LlamaForCausalLM` class, which builds every block with `self_attn` and `input_layernorm`. The weights for those submodules in layers 21, 22, and 23 don't exist in the saved `.safetensors`, so Hugging Face would fall back to initializing them randomly and emit "missing keys" warnings. The model would load without errors but generate garbage: the pruning is lost, and three attention layers now contain random weights.

Figure 8.9 shows the steps we're going to implement next so the pruned model can be saved and reloaded without losing its modified architecture.

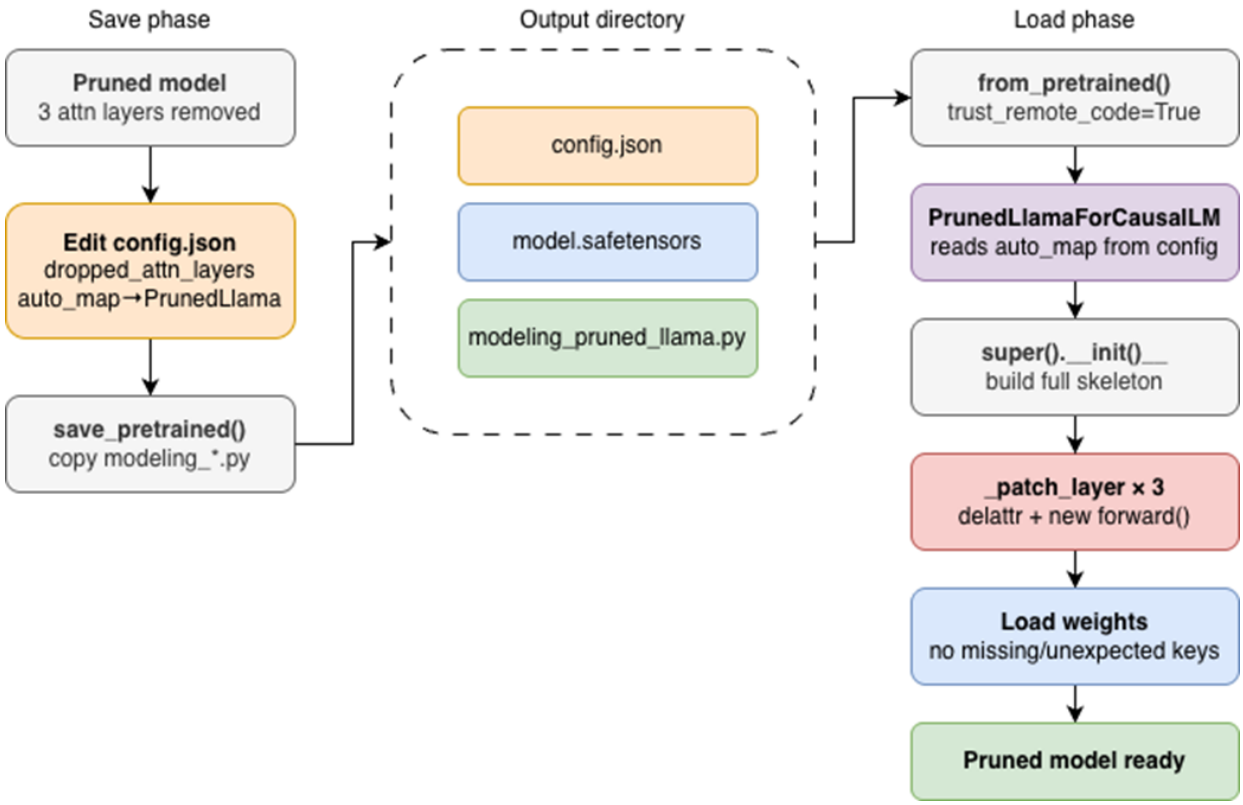


Figure 8.9 Save and reload pipeline for a structurally modified model. During the save phase, the pruned layer indices and the custom class reference are registered in config.json before serializing the weights. During the load phase, PrunedLlamaForCausalLM rebuilds the patched architecture before the weights are loaded, so attention keys with no target in the skeleton are skipped.

To be able to reuse our model, we need to track which layers have been modified. We'll use the `config.json` file for that.

```
model.config.dropped_attn_layers = sorted(layers_to_drop)
```

This parameter is custom; it doesn't exist in Hugging Face or the standard transformers library, so you can name it whatever you want, since the custom code we built to accompany the model is the one responsible for using it.

The class that interprets the parameter and acts on it accordingly is our own creation. It's called `PrunedLlamaForCausalLM`, and it wraps the logic needed for the pruned model to behave like any other Hugging Face model.

Listing 8.10 PrunedLlamaForCausalLM

```
modeling_code = '''
from typing import Optional, Tuple
import torch
from transformers import LlamaForCausalLM


class PrunedLlamaForCausalLM(LlamaForCausalLM):
    def __init__(self, config):
        super().__init__(config)
        dropped = getattr(config, "dropped_attn_layers", [])
        for idx in dropped:
            self._patch_layer(idx) #A

    def _patch_layer(self, layer_idx):
        layer = self.model.layers[layer_idx]

        if hasattr(layer, "self_attn"):
            delattr(layer, "self_attn") #B
        if hasattr(layer, "input_layernorm"):
            delattr(layer, "input_layernorm") #B

        def forward_no_attn(
            self,
            hidden_states: torch.Tensor,
            attention_mask: Optional[torch.Tensor] = None,
            position_ids: Optional[torch.LongTensor] = None,
            past_key_value=None,
            use_cache: Optional[bool] = False,
            cache_position: Optional[torch.LongTensor] = None,
            position_embeddings:
➥Optional[Tuple[torch.Tensor, torch.Tensor]] = None,
            **kwargs,
        ):
            if isinstance(hidden_states, tuple):
                hidden_states = hidden_states[0] #C
            residual = hidden_states
            hidden_states = self.post_attention_layernorm(hidden_sta
tes)
            hidden_states = self.mlp(hidden_states)
            hidden_states = residual + hidden_states
```

```
            return hidden_states

        PrunedLayer = type(
            "PrunedDecoderLayer",
            (type(layer),),
            {"forward": forward_no_attn}
        )
        layer.__class__ = PrunedLayer #D
'''

with open("modeling_pruned_llama.py", "w") as f:
    f.write(modeling_code)
```

#A Reads dropped_attn_layers from the config and calls _patch_layer for each index, automatically rebuilding the pruned architecture when loading the model.
#B hasattr checks that self_attn and input_layernorm exist before deleting them, avoiding potential errors.
#C If the input arrives as a tuple, we extract the first element. It's a safety measure to ensure we always work with the data tensor.
#D The patch is applied at the class level for the same reason as in drop_attention_layer: to prevent transformers from overwriting it during generation.

This class needs to be associated with the model so it runs when it's loaded, we'll do that by modifying the `config.json` file before saving the model.

```
OUTPUT_DIR = “./llama-3.2-3b-attn-drop-3”
model.config.dropped_attn_layers = sorted(layers_to_drop)
model.config.auto_map = {
    "AutoModelForCausalLM": "modeling_pruned_llama.PrunedLlamaForCau
salLM"
}
model.save_pretrained(OUTPUT_DIR)
tokenizer.save_pretrained(OUTPUT_DIR)
shutil.copy("modeling_pruned_llama.py", OUTPUT_DIR) #A
```

#A The file containing the PrunedLlamaForCausalLM class must be copied alongside the weights so that trust_remote_code=True can find it when reloading.

Now the model is ready to be loaded and used:

```
reloaded_model = AutoModelForCausalLM.from_pretrained(
    OUTPUT_DIR,
    torch_dtype=TORCH_DTYPE,
    device_map=device,
    trust_remote_code=True,
)
```

When you call `AutoModelForCausalLM.from_pretrained` with `trust_remote_code=True`, Hugging Face reads the `auto_map` field from the `config.json` file and knows it should instantiate `PrunedLlamaForCausalLM` instead of `LlamaForCausalLM`.

During instantiation, `__init__` calls `super().__init__(config)`, which builds the original architecture skeleton in memory — all 28 blocks with their attention submodules intact. Immediately after, `__init__` reads the `dropped_attn_layers` list from the config (the same list we wrote before saving: `[21, 22, 23]`) and invokes `_patch_layer` for each of those indices. This method removes the attention submodules from those three blocks using `delattr` and replaces the layer's `forward()` with the version that skips the attention phase.

When the constructor finishes, the skeleton in memory matches the architecture we saved: 25 complete blocks and three patched blocks. Only then does `from_pretrained` load the weights from the `.safetensors` file. The checkpoint contains weights only for the submodules that existed at save time; the pruned attention weights were never serialized in the first place, so every key in the checkpoint finds its target in the patched skeleton, and every submodule in the patched skeleton finds its weights in the checkpoint. No missing keys, no unexpected keys.

8.4.2 Memory savings, quality trade-offs, and pruning limits

You've removed three attention modules from the Llama-3.2-3B model, each accounting for roughly 25 million parameters; that is, about 75 million parameters in total, which corresponds to 150 MB in FP16. To understand the full impact of attention pruning and explore its limits, we ran two additional experiments removing 6 and 15 layers respectively. The code for these experiments is available in CH08_NB03_Remove_Attention_15L.ipynb in the book's repository. In this section we'll analyze the results across all three configurations, looking at memory consumption, benchmark performance, and generation quality.

The throughput measurements in this section were obtained on a Colab T4 GPU, where runtime conditions — shared hardware, memory state, thermal load — introduce enough variability to make precise speed comparisons unreliable. Treat them as directional indicators rather than definitive benchmarks. The quantitative analysis of inference speed impact under controlled conditions will be covered in section 8.5, when we look at the results from the original paper, which used dedicated hardware and larger models. For now, we'll focus on what our experiments reveal about memory consumption, benchmark performance, and generation quality across the three configurations.

Figure 8.10 shows the benchmark scores across the four configurations. The pattern is immediate: removing 3 and 6 layers produces contained variations across most benchmarks, with Lambada as the exception. At 15 layers the collapse is total, with scores dropping dramatically across all tasks.

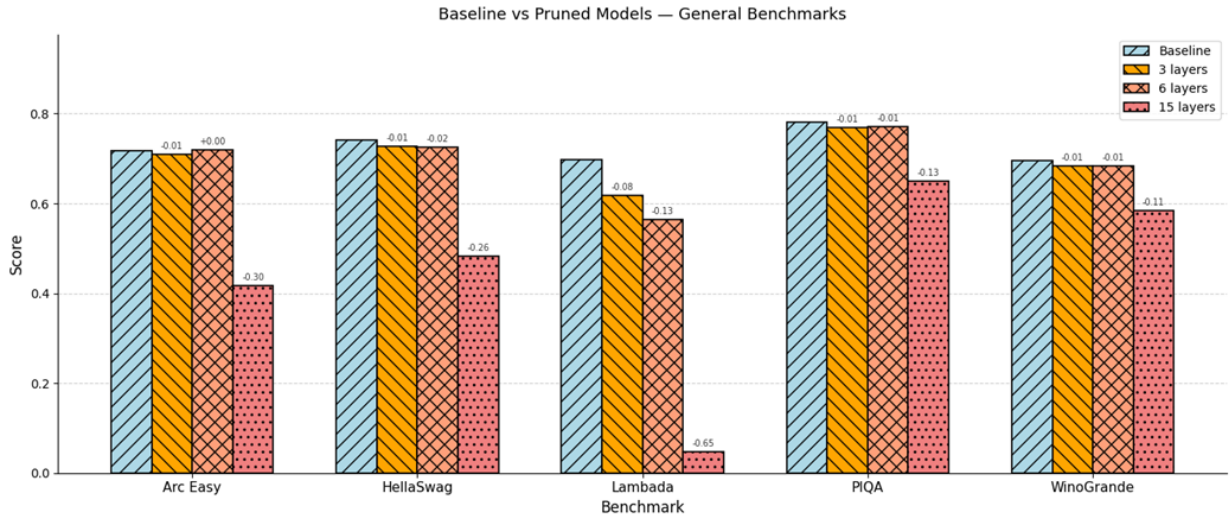


Figure 8.10 Benchmark scores for Llama-3.2-3B across four configurations. Degradation between 3 and 6 layers removed is negligible across all benchmarks, but Lambada shows progressive sensitivity to attention pruning at all ratios, and collapses almost completely at 15 layers removed, along with the rest of the tasks.

With 3 layers removed, four of the five benchmarks show negligible variations, ranging from a slight gain of +0.9% on ARC-Easy to a drop of -1.4% on HellaSwag. At 6 layers the picture is almost identical for these four tasks, with drops remaining below -2%. Lambada tells a different story at both ratios: -9.2% at 3 layers and -19.1% at 6 layers. At 15 layers, all benchmarks collapse, with Lambada falling -93.3%.

ARC-Easy, HellaSwag, PIQA, and WinoGrande are multiple-choice or binary-selection benchmarks: the model receives a context and the evaluation harness compares the probability it assigns to each candidate continuation, picking the most likely one as the answer. These tasks probe commonsense reasoning, physical plausibility, and semantic coherence — capabilities that rely on local discrimination rather than precise token-level generation. A 1-2% degradation here is consistent with having removed layers that the importance scores themselves identified as redundant. The model is still

discriminating correctly between plausible and implausible continuations.

Lambada is a different story. The model must predict the exact last word of a short passage, designed so that the answer is impossible to guess from the last sentence alone — it requires integrating the entire paragraph. Unlike the other four, this is not a closed-set selection task but an open-vocabulary generation task: there are no distractors to rule out, only one correct token among tens of thousands. Its progressive sensitivity across all pruning ratios suggests that long-range contextual integration is the first capability to degrade when attention layers are removed.

As the benchmarks show, removing attention layers affects the model's ability to understand context, although it maintains other capabilities very well. One of its biggest advantages is the reduction of dynamic VRAM consumption, as shown in Table 8.2.

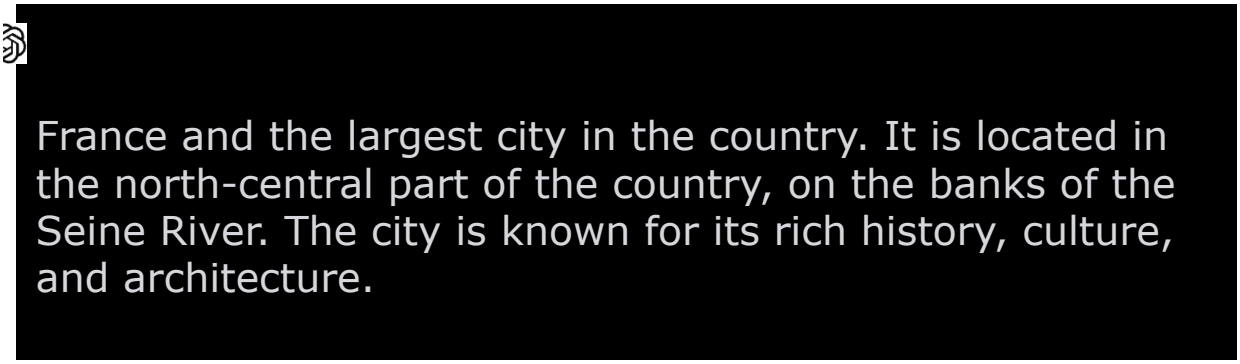
Table 8.2 Dynamic VRAM delta across pruning configurations

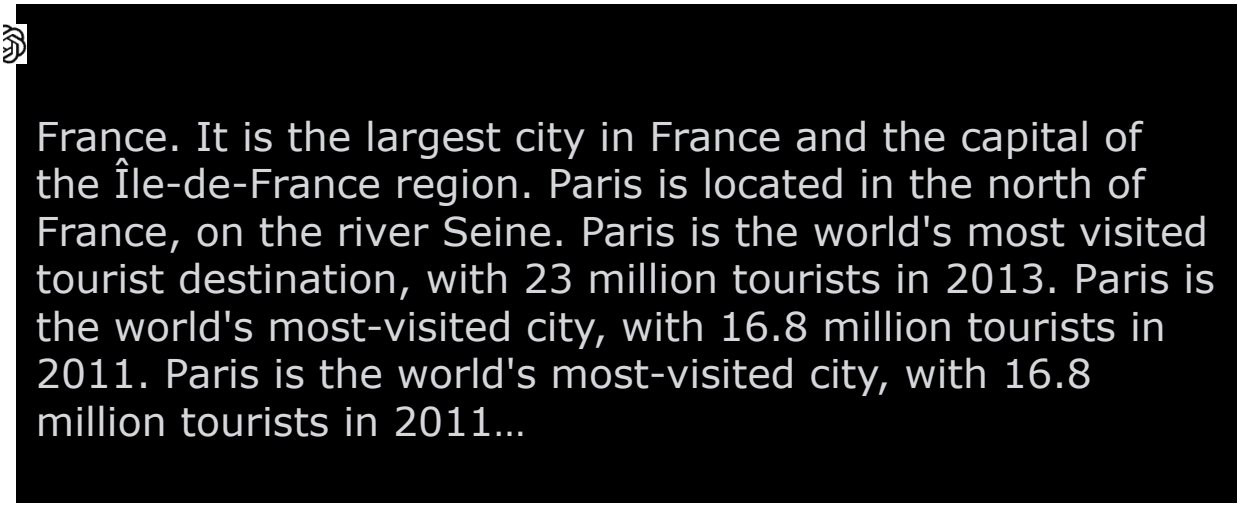
Configuration	Dynamic VRAM	Vs Baseline
Baseline	109.57 MB	_
3 layers removed	96.06 MB	-12.3%
6 layers removed	78.72 MB	-28.2%
15 layers removed	58.63 MB	-46.5%

These figures were obtained with the `measure_memory_allocation` function from the book's utility module, which measures the peak GPU memory growth during a 500-token generation relative to the memory occupied by the loaded model.

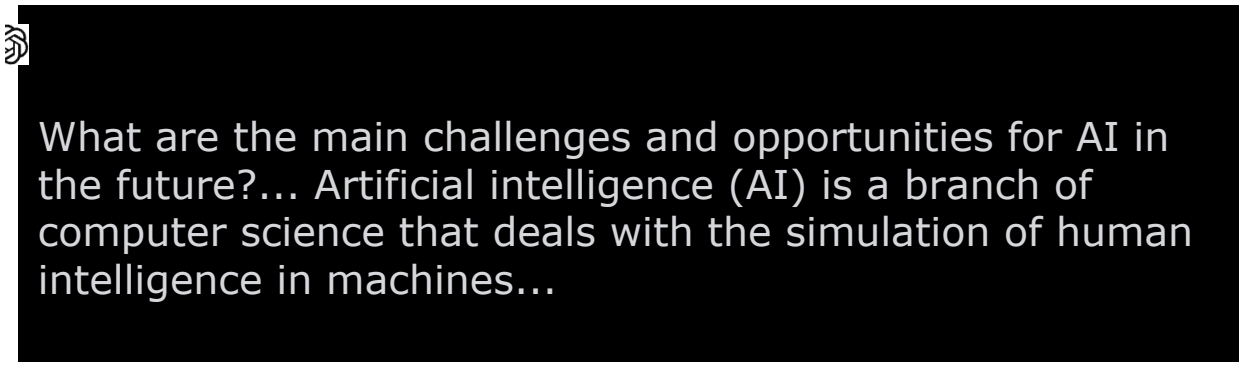
Each attention layer removed means fewer KV cache entries to allocate per generated token, and that translates directly into lower dynamic memory growth. With 3 layers removed the saving is modest. At 6 layers it approaches 30%, and at 15 layers it nearly halves the dynamic memory footprint of the baseline. On a 3B model running on constrained hardware, this is the primary practical benefit of attention pruning.

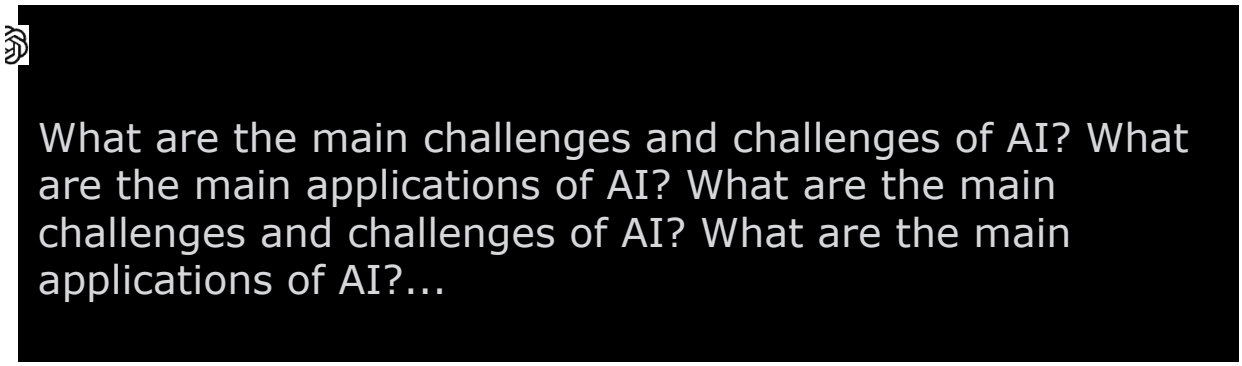
When we check text generation quality, the picture is more nuanced than a simple pass or fail. With a short prompt, the difference between 3 and 6 layers removed is already visible:



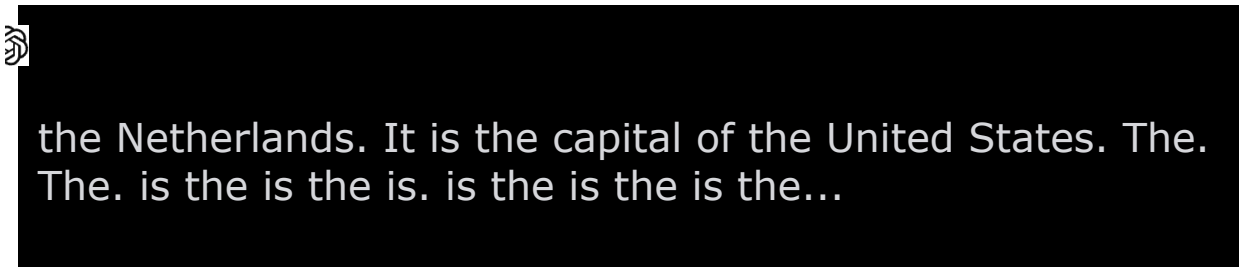


The 6-layer model starts coherently but immediately falls into a repetition loop. With long-form generation the contrast between 3 and 6 layers becomes even sharper:





At 15 layers, the collapse is immediate even with a short prompt:



The benchmarks used in the paper capture the result of a single static pass. The model only has to pick one option, which is why the authors present Attention Drop as a training-free technique. In text generation, where tokens are generated autoregressively, the picture is more nuanced. With three layers removed, the degradation doesn't manifest as an immediate collapse but as subtle factual inconsistencies that emerge in sustained long-form generation. A cost that static benchmarks are not designed to detect.

The paper formalizes this trade-off with a metric called the SDR (Speedup Degradation Ratio). The percentage of performance drop divided by the percentage of speedup gained. A low SDR means the model gets faster while losing little. Section 8.5.1 examines the speed trade-off in more detail alongside our own throughput measurements on the 3B model: removing 11% of attention layers produces no measurable speedup, while removing 54% yields a modest throughput gain at the cost of immediate generation collapse.

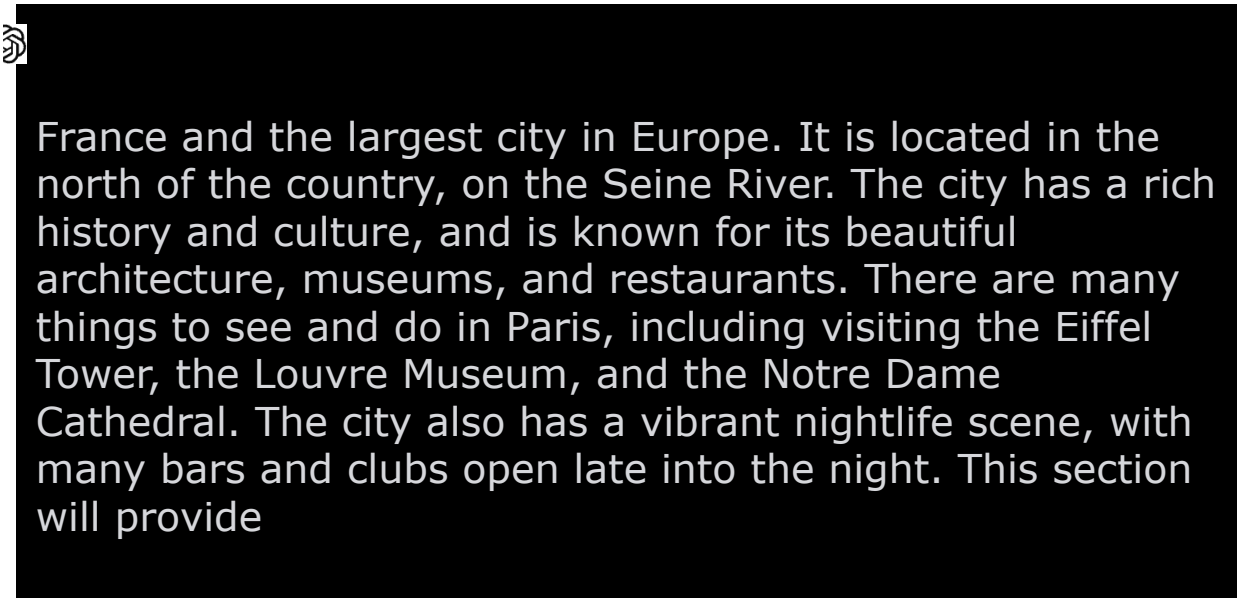
To re-adjust the weights, you can run a short Knowledge Distillation process with OptiPFair, the author's open-source library introduced in Chapter 1. The code implemented in the library for KD is the same we studied in Chapter 6.

```
trained_student, stats = opf.distill_model(
    student_model=student_to_train,
    teacher_model=teacher_model,
    dataloader=train_dataloader,
    alpha=0.6,
    beta=0.40,
)
```

The alpha and beta values control the balance between the two loss components. With alpha=0.6, the student gives more weight to learning directly from the ground truth labels; beta=0.40 keeps the student aligned with the teacher's output distribution through logit-based distillation. These are the default values that produced stable recovery in our experiments.

The recovery process took ten minutes on an A100 GPU using 10,000 records from the Cosmopedia dataset. You can find the complete process in the notebook: CH08_NB04_KD.ipynb (https://github.com/peremartra/Rearchitecting-LLMs/blob/main/CH08/CH08_NB04_KD.ipynb)

After recovery, the model is back to generating coherent text and demonstrating its knowledge.



Maybe due to the influence of the dataset used in the KD process (curated by Hugging Face), the model has gained notable knowledge about Paris attractions. It has compensated for the absence of the six removed attention layers, and the repetition loop disappears. This confirms that the degradation we observed wasn't an irreversible loss of capacity, but a temporary misalignment between architecture and weights that recovery corrects.

8.5 From paper to practice

The second part of this chapter has been based almost exclusively on the paper "What Matters in Transformers? Not All Attention is Needed" (https://arxiv.org/abs/2406.15786), published by Shwai He, Guoheng Sun, Zhenyu Shen, and Ang Li from the University of Maryland in 2024. But it has also been grounded in all the prior work we did in previous chapters.

The paper starts from the premise that not all components within a transformer block have the same importance, and that therefore we could remove the parts that contribute the least instead of the entire block.

The central instrument of the paper is the cosine similarity between the input and output of each module, measured including the residual connection. It's the same metric Chapter 4 uses to identify redundant blocks following the ShortGPT work (Men et al., 2024).

What the paper adds, and this has important consequences, is the insistence on including the residual connection in the measurement. If you only measure the internal output of the attention layer, without adding the residual contribution, you get a misleading estimate. The paper's ablation study demonstrates this clearly: without the residual, performance drops significantly even when removing just a few layers. With it, the metric becomes reliable. In our notebook, the hooks are anchored exactly at the `input_layernorm` points to capture X_A, and `post_attention_layernorm` to capture Y_A, which already includes the residual.

8.5.1 Pruning strategies and evaluation at scale

The authors explore three independent pruning strategies before proposing a fourth that integrates them.

Block Drop eliminates complete Transformer blocks, attention and MLP together, by identifying the blocks whose output is most similar to their input. It's the same technique implemented in chapter 4, with the same cosine similarity metric that ShortGPT calls Block Influence. They include it here as a starting point and reference, specifically to show its limitations: removing four blocks in Llama-2-13B already produces an average drop of 2.4% across benchmarks, and

eight blocks push that to 7.5%. It treats attention and MLP as an indivisible unit, without considering that within each block the two modules can have very different levels of redundancy.

MLP Drop targets exclusively the MLP layers. The result is similar to the previous one: even moderate removals produce significant degradation, especially in Mistral-7B, where removing eight MLP layers causes a near-total collapse in HellaSwag performance.

Attention Drop is the central finding of the paper. Removing eight attention layers from Llama-2-13B, average performance barely moves (68.1 versus 68.2 for the baseline). Removing sixteen, the drop is just 2.1 points on the multiple-choice benchmarks the paper uses as reference. The same pattern shows up in Mistral-7B. Attention layers, despite being the defining element of the Transformer architecture, exhibit a degree of redundancy far greater than usually attributed to them.

Finally, the Joint Layer Drop strategy removes both attention layers and MLP layers, regardless of whether they belong to the same block.

To compare these strategies, the authors introduce the SDR (Speedup Degradation Ratio) metric, defined as the percentage of performance drop divided by the percentage of speedup gained. A low SDR means the model speeds up a lot while losing little. The SDR values for Attention Drop are consistently ten to twenty times lower than those of Block Drop or MLP Drop, making it by far the most efficient strategy. Figure 8.11 visually summarizes the four strategies and their relative SDR.

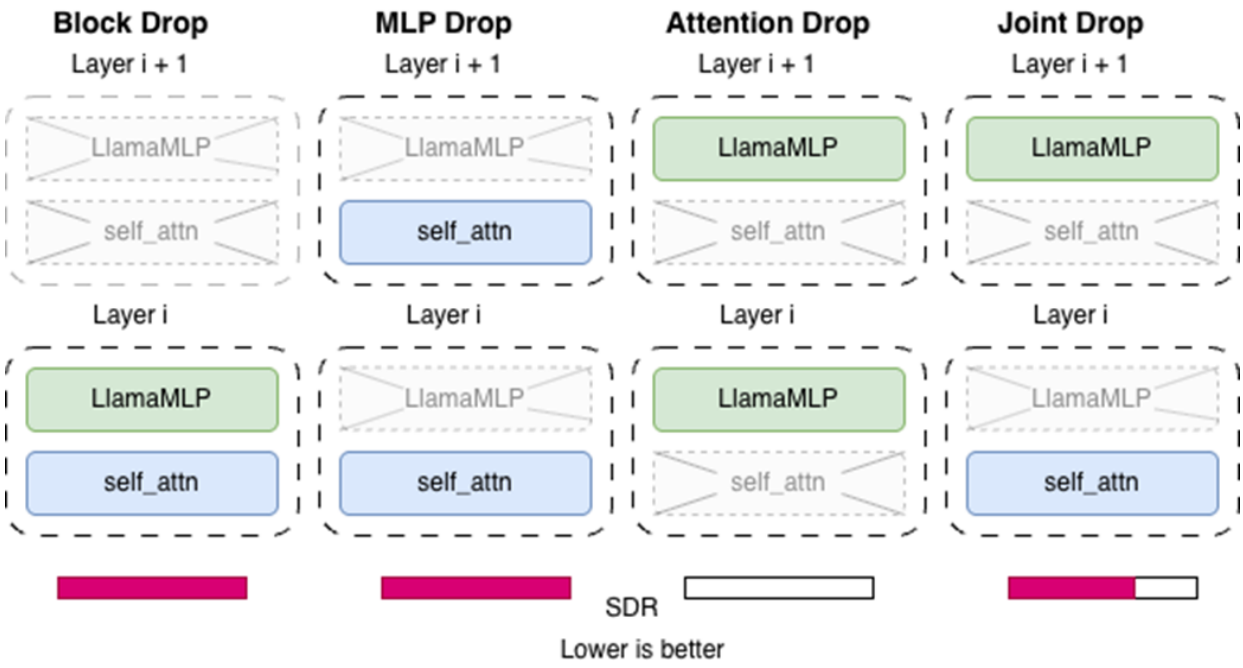


Figure 8.11 The four pruning strategies explored in He et al. (2024). Each column shows which modules are removed from a Transformer block (crossed out) and which are preserved. The bar at the bottom represents the SDR (Speedup Degradation Ratio). A shorter bar indicates a better trade-off between speed gain and quality loss. Attention Drop achieves the lowest SDR by a wide margin, making it the most efficient strategy despite leaving MLP layers intact.

One of the most relevant findings for those working with models in production is behavior at scale. Larger models tolerate attention removal with less impact. In Llama-2-70B, removing 32 of its 80 attention layers, 40% of them, produces an SDR of 0.00. Practically no measurable degradation, with a 1.35× speedup. Removing 40 layers, exactly half, pushes the speedup to 1.48× with an SDR of just 0.05.

With Llama-3.2-3B, the model we used in the notebook, the margin is smaller. The first and last layers are noticeably more active than the middle ones, and the redundancy space is concentrated in a narrower region. That's why we chose to remove only three layers in the experiment. Our throughput measurements put this in concrete terms:

removing 11% of attention layers (3 out of 28) produces no measurable speedup (25.0 vs 25.4 tok/s), while removing 54% (15 out of 28) yields a ~6% throughput gain at the cost of immediate generation collapse — a repetition loop from the very first token. Larger models tolerate significantly higher pruning ratios, which is exactly what the paper demonstrates with its SDR analysis at scale.

8.5.2 Beyond the benchmarks

The authors evaluate their pruned models on a standard set of tasks: ARC-C, BoolQ, HellaSwag, MMLU, OBQA, PIQA, RTE, and WinoGrande. All of them are multiple-choice or binary classification benchmarks. In these tasks the model doesn't generate text freely. It receives several options and must assign a higher probability to the correct one. It's a static evaluation that measures reasoning competence well, but doesn't test autoregressive token-by-token generation. Meaning, the model only needs to generate one token to get the right answer.

In our experiment, we added a text generation test and an additional benchmark, Lambada, not used in the paper. Both reveal a different story from the paper's benchmarks and expose model degradation that would otherwise remain hidden. We only tested two models (we're not writing a paper), but when removing three layers, Lambada showed a ~9% drop in both the 3B and 8B models — though in the latter case the tests were run on the quantized model. The full results for Llama-3.1-8B are available in CH08_NB03_L4_Llama-3.1-8B.ipynb in the book's repository.

In the 3B model, with three layers removed, text generation remained coherent for short prompts but showed factual inconsistencies in sustained long-form generation, a

degradation that static benchmarks are not designed to detect.

These results don't invalidate the technique. We need to consider the task we want to assign to the model. If your use case is classification, ranking, or any task where the model chooses between predefined options, Attention Drop without recovery is perfectly viable, just as the authors claim. If

the use case involves text generation, a recovery process, even a brief one, is necessary to restore autoregressive coherence.

The paper verifies that Attention Drop is fully compatible with quantization. The authors integrate the method with AWQ at 4-bit precision, and the performance difference compared to the pruned model without quantization is less than 1% across all benchmarks tested. This means the two strategies covered in this chapter, KV cache quantization and attention layer removal, are not alternative techniques but cumulative ones. You can apply both to the same model and get the benefits of each without them interfering with each other.

One last thing, the University of Maryland itself has published a more recent paper, *Making Large Language Models Efficient Dense Retrievers* (https://arxiv.org/abs/2512.20612), where they analyzed what happens when you adapt an LLM as a dense retrieval model (for example, to generate embeddings in a RAG system). Their results showed that the architecture's behavior is completely reversed. The attention layers are critical for performance, while the MLPs show high redundancy and can be pruned massively.

This is another example that the optimal structure of an LLM depends directly on the nature of the production task it's going to face. In "What Matters in Transformers," the conclusion is that removing MLP layers causes a significant drop in performance, while in the second one, the conclusion is the opposite. The reason the authors reached such different conclusions comes down to the metrics used; metrics that aren't arbitrary but are directly tailored to what each task demands from the model.

8.6 Hands-on lab

This chapter has combined two complementary approaches: KV cache optimization, which acts on dynamic memory without modifying the architecture, and the physical removal of Attention Modules, which reduces the model permanently. The exercises in this section are designed to consolidate both perspectives. The first validates the chapter's central metric from the opposite side. The second combines the two techniques covered. The latter corrects the degradation that, as we've seen, pruning introduces.

- Validating the importance metric with inverse pruning:
    - What to do: In notebook NB03, locate the `select_layers_to_drop` function from Listing 8.8. Modify it to select the layers with the highest importance scores instead of the lowest, inverting the selection criterion. Drop the three most important layers from the model and run the same benchmarks from the notebook on the resulting model.
    - Questions to answer: How much does the score drop on each benchmark compared to the base model? Is the degradation uniform across all benchmarks, or does it hit some harder than others? Compare these results with those from the correctly pruned model

in section 8.4.2. Does the difference between both experiments justify the use of cosine similarity as an importance metric? What happens with the text generation test?

- Combining KV cache quantization and attention pruning:
  - What to do: The model oopere/llama-3.2-3b-attn-drop-6, available on Hugging Face Hub, is the Llama-3.2-3B resulting from notebook NB03, with the Attention Modules from layers 18, 20, 21, 22, 23 and 24 physically removed. Load it and apply 4-bit KV cache quantization with HQQ, following the same configuration from notebook NB01. Measure the three metrics you already know: static VRAM, dynamic VRAM delta, and throughput. Compare the results with those obtained in section 8.1 on the full base model.
  - Questions to answer: Does the dynamic VRAM reduction offered by KV cache quantization hold, amplify, or shrink on a model that already has fewer attention layers? Does throughput change compared to the base model with quantized KV cache? What happens to the quality of the generated text when combining both techniques?
- Recovering the pruned model with knowledge distillation:
  - What to do: The model oopere/llama-3.2-3b-attn-drop-6 shows repetition loops in autoregressive generation, as we saw in section 8.4.2. Use the notebook CH08_NB04_KD.ipynb from the book's repository as a reference to run the recovery process with OptiPFair. Your task is to experiment with at least two different configurations, varying one of these parameters: the balance between alpha and beta, the number of samples in the calibration dataset, or the training duration. For each

configuration, evaluate the recovered model with the Paris prompt from section 8.4.2 and with the benchmarks from notebook NB03.
    - Questions to answer: What's the minimum configuration needed for the repetition loop to disappear? Is it enough for Lambada to improve to confirm that autoregressive generation has recovered, or do you also need to verify the quality of the generated text directly?

You can share your results, models you publish on the Hugging Face Hub, and any questions about the exercises in the Discussions space of the book's GitHub repository, where you'll find a dedicated thread for this chapter.

8.7 Summary

- The dynamic memory consumption of the attention mechanism scales with context length. Quantizing the KV cache reduces this memory footprint without altering the model weights, allowing longer prompts to be processed and batch sizes to be increased in production engines like vLLM.
- Weight quantization and KV cache quantization are independent levers that act on different areas of GPU memory. Applying one does not interfere with the other, and both can be combined in the same inference pipeline.
- The effectiveness of KV cache quantization depends on both the inference engine and the GPU architecture. On GPUs with native FP8 support, vLLM doubles the available KV token capacity with no throughput penalty; on older architectures like the T4, the same operation introduces a software overhead that penalizes generation speed.

- Attention layers are significantly more redundant than MLP layers. Anchoring PyTorch hooks at `input_layernorm` and `post_attention_layernorm` gives finer-grained importance scores than block-level measurement.
- You can modify a model's execution flow without retraining it by creating a custom class that inherits from the original architecture and overrides its `forward()` method. This technique is reusable across any structural modification that requires bypassing or replacing internal modules.
- Saving and reloading a structurally modified model requires registering the pruned layer indices in `config.json` and shipping a custom class alongside the weights. Without this mechanism, `from_pretrained` reconstructs the original architecture and fails at the first forward pass.
- Multiple-choice benchmarks do not reveal the full cost of attention pruning. Benchmarks like Lambada and free-text generation expose a degradation that static, closed-set evaluation hides entirely.
- The optimal pruning strategy depends on the production task. The cosine similarity metric that identifies attention layers as redundant in generative models points to MLP layers as the expendable component in dense retrieval architectures, where the conclusions are precisely inverted.

9 Dynamic routing with Mixture of Experts

This chapter covers

- Understanding dynamic routing for adaptive inference
- The core components of a Mixture of Experts architecture (MoE)
- Transforming an open source model into a MoE
- Comparing routing strategies for inference
- Reusing a specialized model as a MoE expert

Mixture of Experts has quickly moved from being a lab experiment to one of the most important architectures in frontier LLMs and, inevitably, it has made its way to SLMs. Mistral proved it could work outside the lab with the successful Mixtral 8x7B, and DeepSeek took the idea a step further, with fine-grained architectures: many small experts instead of a few large ones, each covering a very specific portion of the model's knowledge.

In this chapter, we apply the same logic but at a smaller scale. We will add highly specialized experts to a dense model, incorporating domain expertise without degrading, or even improving, its general capabilities, and you will get firsthand experience with an architecture already proven at the frontier, and one you'll be seeing much more of in SLMs.

We will explore MoE with two complementary projects. First, we will build a two-expert system on SmolLM2-1.7B-Instruct using sparse upcycling, initializing Expert 1 as a copy of the base model's MLP Module weights and training it alongside

the router so it specializes in clinical extraction. We will also compare two routing strategies: *soft routing,* where each token receives a weighted mix of both experts, and *hard routing,* where the router makes a binary decision. Then we will go a step further: instead of starting from the base model, we will take the clinical expert we already trained in Chapter 7 and transplant it directly as Expert 1 in our MoE architecture, fundamentally changing the training strategy.

But MoE is not the only dynamic routing technique. Before we start building our model, let us take a brief look at the different approaches that exist.

9.1 Dynamic inference techniques

The computational cost per forward pass of an LLM is, by default, constant. It varies depending on the number of tokens it has to generate, but to generate each of those tokens, it uses its entire structure. It does not matter if the token being generated answers a trivial question or is part of a fifty-page report analysis. To produce that token, the model executes exactly the same operations, activates the same parameters, and traverses all its layers from beginning to end. It is an architecture designed for the hardest case, even though most inputs do not require it.

Dynamic inference techniques make decisions at runtime about

what to compute, what to skip, or where to route each token. It is a paradigm shift that, as we will see, has implications for both architecture and training.

The three types of approaches are early exiting, token dropping, and Mixture of Experts (which is the one we focus on in this chapter).

The most intuitive of these approaches is early exiting. Why go through all the layers of the model if you already have enough confidence in the answer? The idea is to add intermediate exit points in the model, where a small classification layer evaluates whether the representation accumulated up to that point is good enough to generate an answer. If the confidence exceeds a threshold, the model exits before completing the forward pass.

The evaluation is based on the logits generated at each block. The more concentrated the probability distribution over the vocabulary is on a single candidate, the higher the model's confidence. This concentration is what determines whether the model exits at that point or continues processing the remaining blocks.

A different approach is token dropping, which instead of deciding which parts of the model to execute, discriminates which tokens to process. Prepositions, articles, or filler tokens can be uninformative for the middle layers of the model, so they are excluded from processing. To do this, the importance of each token is evaluated layer by layer, using attention weights as an indicator.

A token that receives little attention from the rest of the sequence is considered dispensable and is discarded from the calculation in subsequent layers. This discarding translates directly into a reduction of the KV cache, where the key-value pairs of the removed tokens are no longer stored or looked up. The result is that the sequence keeps shrinking as the model goes deeper, concentrating the computation on the tokens that really matter. It is especially useful in long contexts, where the savings can be significant.

The technique we'll cover in this chapter is Mixture of Experts. Instead of reducing work by dropping tokens or exiting early, it keeps the full forward pass but routes it

toward the most suitable expert for each token. In a MoE architecture, dense MLP Modules are replaced by a set of specialized experts and a router that decides, token by token and at runtime, which of them should process each element of the sequence. The result is not a model that does less, but a model that does the right thing.

The three techniques share a limitation worth knowing before considering adoption. The main one is compatibility with standard inference libraries like vLLM or TGI. These libraries are optimized for conventional architectures and assume a uniform forward pass. Any modification that introduces conditional decisions at runtime — whether exiting early, discarding tokens, or routing to experts — breaks those assumptions and requires custom implementations or specific adaptations. This is not usually a problem for large labs that control both the model and the inference engine, but it is worth keeping in mind when building SLMs.

The lack of compatibility is partly because all three techniques are still under active research. Results are promising, but production adoption is still limited. With one exception: MoE is the most mature of the three, and it is present in some of the models supported by libraries like vLLM, though its integration into standard inference tools is still a work in progress.

Table 9.1 summarizes the three techniques, their decision mechanism, where in the model they act, and their main objective.

Table 9.1 Dynamic inference techniques

Technique	What it decides	Decision mechanism	Where it acts	Main objective
Early Exiting	When to stop	Logit concentration on a single token	Model blocks	Reduce the number of processed blocks
Token Dropping	Which tokens to process	Attention weights	KV cache	Reduce the active sequence length
Mixture of Experts	Which expert to route to	Trained linear router	MLP module	Direct each token to the most suitable expert

Of the three techniques, we'll focus on the most mature and most widely used one: Mixture of Experts. Not only is it beginning to appear in the creation of SLMs, but it's also an architecture used by the labs that build today's state-of-the-art models.

9.2 Implementing and evaluating a two-expert MoE

Before we start building the model, let's analyze the main components of an MoE, which is made up of three components:

- **The experts:** They process information independently, each one potentially specialized in a specific type of input.
- **The router:** A linear layer that decides how to distribute computation among the available experts.
- **The combined forward pass:** The mechanism that takes the experts' outputs and mixes them, in proportions determined by the router, to produce a single output.

In this chapter we'll build a two-expert system on SmolLM2-1.7B-Instruct (https://huggingface.co/HuggingFaceTB/SmolLM2-1.7B-Instruct). Hugging Face built this model on the Llama architecture, so the layer names you'll see in the code, like LlamaMLP or LlamaDecoderLayer, keep Meta's original naming. The design is deliberately simple. We'll have Expert 0, formed by the original model weights, which will remain frozen during training to preserve the general behavior. Alongside it, Expert 1, a copy of the original model weights, which we'll specialize for clinical extraction.

The intervention affects exclusively the MLP module of each Transformer block. The attention is not modified. We also don't replace the entire model — we'll limit ourselves to layers 13 through 23, the deepest ones, where representations are more abstract and specialization is more effective.

The router, which we'll train from scratch, will learn to decide how much weight to give each expert based on the token being processed. This router, together with the experts, forms the MoE Block that replaces the MLP layer inside the Transformer block, as we can see in Figure 9.1.

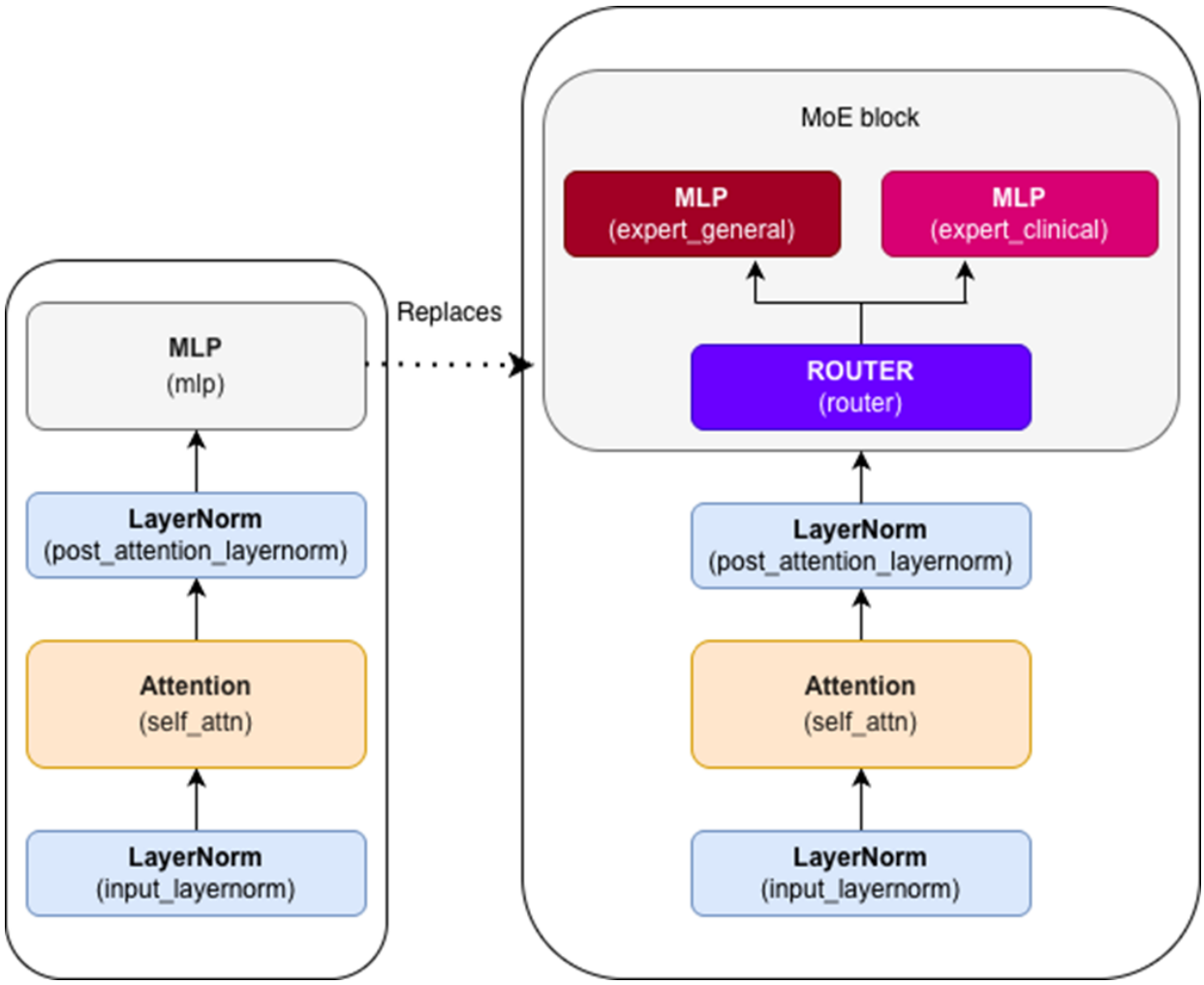


Figure 9.1 The MoE block replaces the MLP module in selected transformer layers. A router receives the hidden states and assigns routing weights to two experts that process in parallel — `expert_general`, which holds the original MLP weights (frozen), and `expert_clinical`, the trainable copy. Attribute names in parentheses correspond directly to the model's code structure.

We start from a model we already know, duplicating its MLP layers from 13 to 23 and adding a router inside each new MoE block. This duplication is known as *sparse upcycling.* Rather than initializing the new experts randomly, we start them from already-trained weights, giving the model a head start on the specialization task.

The only weights we will train are the new expert and the routers. The rest of the model remains frozen.

The code for this section is available in the notebook CH09_NB01_MoE.ipynb in the book repository.

9.3 Building the MoE

To build this first MoE, we'll follow four steps:

1. Define the MoE block that will replace each MLP Module.
2. Replace the selected MLPs with the new block.
3. Freeze the weights that will not be trained, leaving the newly created experts and the router as trainable.
4. Train the model.

DEFINING THE MOE BLOCK

The MoE block is the core component of our architecture. It will replace the MLP Module in each of the selected blocks, and contains the two experts and the router. The `Trainable2ExpertMoE` class defines this new block by creating the necessary layers and defining the `__init__` and `forward` methods of the MoE block.

Listing 9.1 Trainable2ExpertMoE

```
class Trainable2ExpertMoE(nn.Module):
    def __init__(self, original_mlp, hidden_size):
        super().__init__()
        target_dtype = next(original_mlp.parameters()).dtype
        target_device = next(original_mlp.parameters()).device

        self.expert_general = original_mlp #A
        self.expert_clinical = copy.deepcopy(original_mlp) #B
        self.router = nn.Linear(hidden_size, 2,
                                bias=False,
                                dtype=target_dtype,
                                device=target_device) #C
        self.last_routing = None

    def forward(self, hidden_states):
        router_logits = self.router(hidden_states)

        if self.training:
            noise = torch.randn_like(router_logits) * 0.35  #1
            router_logits = router_logits + noise

        routing_weights = F.softmax(
            router_logits.float(), dim=-1
        ).to(hidden_states.dtype)  #2

        out_general = self.expert_general(hidden_states)
        out_clinical = self.expert_clinical(hidden_states)

        # Soft routing: each token receives a weighted blend of both
experts
        output = out_general * routing_weights[..., 0].unsqueeze(-1)
[CA]+ out_clinical * routing_weights[..., 1].unsqueeze(-1) #3
        self.last_routing = routing_weights.detach() #D

        return output
```

#A Expert 0 receives the original MLP from the Transformer block.
#B Expert 1 is a deep copy of the original MLP.
#C The router is a linear layer that projects the hidden state of each token to two logits, one per expert.
#D We save the routing weights so we can inspect them later in the routing analysis.

The `Trainable2ExpertMoE` class is a new `nn.Module` that acts as a wrapper for `LlamaMLP`, the original MLP module from our model. It receives it as an argument and stores it as an attribute, converting it into Expert 0. The original `LlamaMLP` remains exactly what it was, but now lives inside a larger block that also contains Expert 1 and the router.

It has two methods: `__init__`, which runs once at construction time and defines the block's structure, and `forward`, which runs at every model step, both during training and inference.

In `__init__`, the block receives the original MLP from the layer it's going to replace and the model's hidden state size. With those two elements it builds the three components of the MoE block. The original MLP becomes Expert 0 directly, preserving its weights intact. Expert 1 is initialized as a deep copy of those same weights — since Expert 1 is not frozen, its weights will drift during training. The router is a simple linear layer that projects each token's hidden state, of dimension `hidden_size`, to one value per expert — in this case two values. Finally, `last_routing` is initialized to `None` and will act as a store for the routing weights from the last forward pass, which will let us inspect them later.

In the forward pass, the block receives the hidden states of the tokens and processes them in three phases. First, the router calculates its logits by projecting those hidden states.

During training, before converting those logits into probabilities, we add Gaussian noise (#1). Without this noise, the router can collapse toward a single expert from the very first iterations. At the start of training, Expert 0 and Expert 1 have identical weights, so any random fluctuation can create an early preference in the router toward one of them. Once that preference is established, the favored

expert receives more gradient and improves faster. The more the router prefers it , the more the other expert gets sidelined with barely any learning signal. Expert 1 is the only one that can adapt, since Expert 0 is frozen, but for the router to learn to direct clinical tokens to it, it needs time to observe that it produces better outputs in that context. The noise breaks that initial dynamic by forcing the router to explore both experts during the first epochs, giving Expert 1 time to specialize before the router starts making consistent decisions.

Next, we compute the routing weights with a softmax (#2), which converts the two logits into probabilities that sum to 1. This operation is performed in FP32 even when the rest of the model runs in float16, because at reduced precision softmax can produce degenerate values that completely zero out one expert's contribution. It's a minimal cost that avoids numerical instabilities during training.

NOTE

An MoE model can have an indeterminate number of experts and not all of them need to be selected to contribute to the response. For example, it could have eight experts and select the output of only two.

With those weights, both experts process the hidden states in parallel and their outputs are combined (#3) in a weighted sum where the weights assigned by the router always sum to 1. This guarantees that the output magnitude is consistent with what the rest of the model expects to receive.

REPLACING THE SELECTED MLP LAYERS

With the MoE block defined, the next step is to transplant it into the model. The process is straightforward. We iterate over the selected layers, replacing the original `LlamaMLP` with an instance of `Trainable2ExpertMoE`, passing it the original MLP and the hidden state size.

Listing 9.2 Replacing layers

```
LAYERS_TO_MODIFY = range(13, 24)
hidden_size = model.config.hidden_size

for layer_idx in LAYERS_TO_MODIFY:
    original_mlp = model.model.layers[layer_idx].mlp #A
    model.model.layers[layer_idx].mlp = Trainable2ExpertMoE(
                                                original_mlp,
                                                hidden_size) #B
```

#A Extract the original LlamaMLP before replacing it and passing it as Expert 0.
#B Replace the MLP with the new MoE block, which already contains Expert 0, Expert 1, and the router.

The choice of which blocks to modify does not have a single answer. In the literature we find very different approaches. While Mixtral applies MoE to all blocks of the model, others limit it to the deepest ones, or alternate blocks, and there are even proposals that concentrate hundreds of experts in a single block.

In our case, we modified blocks 13 to 23, the deeper half of the model, for a practical reason. Working in Colab with a T4, replacing all blocks would not only have slowed down training, but would likely have caused an out-of-memory error when training. Deeper blocks are also the most abstract and specialized, which makes them natural candidates for domain specialization.

We can inspect the result by printing the model structure, where the difference between the two groups of blocks is clearly visible.

Listing 9.3 MoE structure

```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(49152, 2048, padding_idx=2)
    (layers): ModuleList(
      (0-12): 13 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
          (q_proj): Linear(in_features=2048, out_features=2048, bias
=False)
          (k_proj): Linear(in_features=2048, out_features=2048, bias
=False)
          (v_proj): Linear(in_features=2048, out_features=2048, bias
=False)
          (o_proj): Linear(in_features=2048, out_features=2048, bias
=False)
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=2048, out_features=8192, b
ias=False)
          (up_proj): Linear(in_features=2048, out_features=8192, bia
s=False)
          (down_proj): Linear(in_features=8192, out_features=2048, b
ias=False)
          (act_fn): SiLUActivation()
        )
        (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
        (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
      )
      (13-23): 11 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
          (q_proj): Linear(in_features=2048, out_features=2048, bias
=False)
          (k_proj): Linear(in_features=2048, out_features=2048, bias
=False)
          (v_proj): Linear(in_features=2048, out_features=2048, bias
=False)
          (o_proj): Linear(in_features=2048, out_features=2048, bias
=False)
        )
        (mlp): Trainable2ExpertMoE(
          (expert_general): LlamaMLP(
            (gate_proj): Linear(in_features=2048, out_features=8192,
bias=False)
```

```
            (up_proj): Linear(in_features=2048, out_features=8192, b
ias=False)
            (down_proj): Linear(in_features=8192, out_features=2048,
bias=False)
            (act_fn): SiLUActivation()
          )
          (expert_clinical): LlamaMLP(
            (gate_proj): Linear(in_features=2048, out_features=8192,
bias=False)
            (up_proj): Linear(in_features=2048, out_features=8192, b
ias=False)
            (down_proj): Linear(in_features=8192, out_features=2048,
bias=False)
            (act_fn): SiLUActivation()
          )
          (router): Linear(in_features=2048, out_features=2, bias=Fa
lse)
        )
        (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
        (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
      )
    )
    (norm): LlamaRMSNorm((2048,), eps=1e-05)
    (rotary_emb): LlamaRotaryEmbedding()
  )
  (lm_head): Linear(in_features=2048, out_features=49152, bias=Fals
e)
)
```

In listing 9.3, we can see how blocks 0 to 12 keep their original `LlamaMLP` form, but blocks 13 to 23 have been replaced by `Trainable2ExpertMoE`. Inside it, we find the original `LlamaMLP` block, which we named `expert_general`, and another `LlamaMLP` block named `expert_clinical`. The attention and normalization layers remain unchanged.

FREEZING STRATEGY

With the architecture ready, it's time to define which parameters are going to be trained. We start by freezing the

entire model and then unfreeze only Expert 1 and the router of each MoE block. Expert 0 remains frozen to preserve the general behavior of the base model.

Listing 9.4 Preparing model weights

```
for param in model.parameters():
    param.requires_grad_(False) #A

for layer_idx in LAYERS_TO_MODIFY:
    for param in model.model.layers[layer_idx].mlp.expert_clinical.p
arameters():
        param.requires_grad_(True) #B
    for param in model.model.layers[layer_idx].mlp.router.parameters
():
        param.requires_grad_(True) #B

trainable = sum(p.numel() for p in model.parameters() if p.requires_
grad)
total = sum(p.numel() for p in model.parameters())
print(f"Total parameters              : {total:,}")
print(f"Trainable (Expert 1 + router) : {trainable:,}  ({trainable/t
otal:.2%})")
```

#A Freeze all model parameters as a starting point.
#B Unfreeze only Expert 1 and the router in each modified MoE block.

Which returns:

```
Total parameters              : 2,265,069,568
Trainable (Expert 1 + router) : 553,693,184  (24.44%)
```

Since most of the blocks are frozen, we only need to train 24.44% of the model weights, which correspond to the MLP layers (`expert_clinical`) of the new MoE blocks (`Trainable2ExpertMoE`) with their corresponding router layers.

TRAINING

Before training, we need to prepare the data. We will use a mix of two types of examples: clinical prompts from the

`oopere/clinical-ner-qdora dataset` that we already used in Chapter 7, and general conversations from the `HuggingFaceTB/smoltalk` dataset (https://huggingface.co/datasets/HuggingFaceTB/smoltalk). The mix is deliberate; if we trained only on clinical data, the router wouldn't have any signal to learn to discriminate between contexts. The smoltalk dataset was part of the base model's own training data, but this is not a problem here. We are not trying to teach the model new knowledge; we want the router to learn which expert to send each token to. If anything, this can work in our favor. Because the model already recognizes this style of text, the router may find it easier to associate it with the general expert rather than the clinical one, where only the medical dataset appears.

Listing 9.5 Dataset preparation

```
clinical_prompts = []
for item in train_data:
    label = json.loads(item["label"]) if isinstance(item["label"], s
tr) else item["label"]
    messages = [
        {"role": "system", "content": "Extract:"},
        {"role": "user", "content": item["note"]},
        {"role": "assistant", "content": json.dumps(label, indent=
2)},
    ]
    clinical_prompts.append(
        tokenizer.apply_chat_template(messages, tokenize=False,
                                      add_generation_prompt=False)
    )

smoltalk = load_dataset("HuggingFaceTB/smoltalk", "all", split="trai
n")
smoltalk_single = smoltalk.filter(
    lambda x: len(x["messages"]) == 2 and len(x["messages"][0]["cont
ent"]) < 300
) #1

general_subset = smoltalk_single.select(range(GENERAL_TRAIN_SAMPLE
S))

general_prompts = []
for item in general_subset:
    general_prompts.append(
        tokenizer.apply_chat_template(
            item["messages"],
            tokenize=False,
            add_generation_prompt=False,
        )
    )

training_prompts = clinical_prompts + general_prompts
random.shuffle(training_prompts) #2
```

The clinical prompts follow exactly the same format we used in Chapter 7: a system message with "Extract:", the clinical

note as the user message, and the structured JSON as the assistant response.

For general prompts, we filter SmolTalk to keep only single-turn conversations and short questions (#1). Long or multi-turn conversations would introduce unnecessary complexity in the general training signal, and could make it harder for the router to learn to distinguish between both contexts in a limited training process like ours.

Once both sets are built, we mix them and shuffle them randomly (#2) before training. This ensures that the router sees clinical and general examples interleaved throughout the entire training, instead of learning them sequentially. If we didn't do this, the router would first learn to route everything to one expert and then would have to unlearn that behavior, which would make convergence harder.

With the data ready, the modified model, and the weights in the required state, we can now start training.

Listing 9.6 Training loop

```
EPOCHS = 3
def train_MoE_router_expert(model):
    model.config.use_cache = False  #A
    model.to(DEVICE).train()

    for param in model.parameters():
        if param.requires_grad:
            param.data = param.data.float()  #1

    router_params = []
    expert_params = []

    for layer_idx in LAYERS_TO_MODIFY:
        router_params.extend(
            model.model.layers[layer_idx].mlp.router.parameters())
        expert_params.extend(
            model.model.layers[layer_idx].mlp.expert_clinical.parame
ters())

    optimizer = optim.AdamW([
        {"params": router_params, "lr": 1e-6},
        {"params": expert_params, "lr": 1e-5},
    ])  #2

    scaler = GradScaler("cuda")

    for epoch in range(EPOCHS):
        total_loss = 0
        random.shuffle(training_prompts) #B

        for idx, text in enumerate(training_prompts):
            inputs = tokenizer(
                text, return_tensors="pt",
                truncation=True, max_length=MAX_TRAIN_LENGTH
            ).to(DEVICE)

            optimizer.zero_grad()

            with autocast("cuda", dtype=torch.float16):
                loss = model(**inputs, labels=inputs["input_ids"]).l
oss
```

```
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()

            total_loss += loss.item()
            if (idx + 1) % 100 == 0:

    model.config.use_cache = True
```

#A Deactivate the KV cache during training.
#B Shuffle the prompts at the start of each epoch to prevent the model from memorizing the order of the examples.

The `train_MoE_router_expert` function orchestrates the training of Expert 1, the medical data expert, and the routers.

It starts by disabling the KV cache and putting the model in training mode. The trainable parameters are promoted to FP32 #1. Even though the model operates in float16 to reduce memory consumption, optimizers like AdamW are numerically unstable at reduced precision. Weight updates can be so small that float16 rounds them to zero. Promoting only the trainable parameters to FP32 is a reasonable tradeoff — it keeps memory consumption low for the rest of the model and guarantees numerical stability where it's needed.

You configure the optimizer with two different learning rates #2. The router learns at 1e-6 and Expert 1 at 1e-5. The router is a small linear layer that starts from scratch and needs very fine adjustments to avoid overfitting in the early iterations. Expert 1, on the other hand, starts from already pretrained weights and needs a stronger signal to specialize in the clinical domain without taking too much time to converge.

The training loop is standard. For each epoch, we shuffle the prompts, run the forward pass with `autocast` in float16 to make the most of the T4 memory, compute the gradient

with the `GradScaler` to compensate for the numerical scaling of mixed precision, and update the weights.

With the model trained, we have our first functional MoE. The next step is to evaluate the model's behavior in the two domains we care about and analyze whether the tokens are actually being routed to different experts.

9.4 Testing the MoE

The first test is to check that the model still responds correctly in both domains. Using a general knowledge prompt and a clinical extraction one, which is what the expert is trained for:

```
messages_general = [{"role": "user",
                     "content": "What's the most significant buildin
g in Barcelona?"}]

messages_clinical = [{"role": "system", "content": "Extract:"},
                     {"role": "user", "content": "Patient is a 50 yo
F c/o severe
                      chest pain and SOB since yesterday. Vitals: HR
110, BP 150/90.
                      Started on Aspirin 81mg."}]
```

The evaluation functions and the schema compliance metric are the same ones we developed in Chapter 7, so we won't reproduce them here. The results are as follows:

```
[Test A - General knowledge]
Response: The most significant building in Barcelona is the Sagrada
Familia, a large-scale Roman Catholic church designed by Antoni Gaud
í.
It is one of the most famous buildings in the world and a major tour
ist
attraction.

[Test B - Clinical NER]
Response: {
  "patient_age": 50,
  "symptoms": [
    "chest pain",
    "shortness of breath"
  ],
  "vital_signs": {
    "temperature": null,
    "heart_rate": 110,
    "blood_pressure": "150/90"
  },
  "medications": [
    {
      "name": "Aspirin",
      "dose": "81mg",
      "frequency": "not specified"
    }
  ],
  "duration_days": 1
}
```

Analyzing these two specific responses, we can see that the general question about Barcelona receives a natural and coherent answer, a sign that Expert 0 and the overall behavior of the model have not been compromised by the training. The clinical note, written with typical medical practice abbreviations like "yo" for years old or "SOB" for shortness of breath, is correctly transformed into a structured JSON that respects the schema we defined in Chapter 7.

Let us now look at the compliance rate across the full dataset:

```
                samples  valid_json_pct  schema_ok_pct
category
abbreviations         6           100.0          100.0
clean                 8           100.0          100.0
implicit              7           100.0          100.0
irrelevant           12           100.0           91.7
typos                 7           100.0          100.0

Overall schema compliance (soft): 97.5%
```

The model achieves 97.5% schema compliance on the test set, maintaining 100% valid JSON across virtually all categories. The only drop appears in the "irrelevant" category — notes that contain no extractable clinical information — where we go from 58.3% accuracy with the base model without fine-tuning, measured in Chapter 7, to 91.7% with Expert 1 inside the MoE architecture. It is clear that the MoE has contributed its knowledge, but let us verify this by inspecting how tokens are distributed among the experts in one of the model's blocks.

We define a function that runs a forward pass and, for a specific layer, shows how much weight each expert receives for each token in the sequence:

Listing 9.7 Analyze routing

```
def analyze_routing(text, layer_index=0):
    inputs = tokenizer(text, return_tensors="pt").to(DEVICE)
    with torch.no_grad():
        with torch.amp.autocast("cuda", dtype=torch.float16):
            _ = model(**inputs)

    moe_layer      = model.model.layers[layer_index].mlp #A
    routing_weights = moe_layer.last_routing.squeeze(0) #B
    tokens          = tokenizer.convert_ids_to_tokens(inputs["input_
ids"][0])

    print(f"Routing analysis (layer {layer_index}):")
    print(f"{'Token':<18} | {'Expert 0 (General)':<20} | {'Expert 1
(Clinical)'}")
    print("-" * 65)
    for token, weights in zip(tokens, routing_weights):
        clean = token.replace("Ġ", "").replace(" ", "")
        print(f"{clean:<18} | {weights[0].item():<20.4f} | {weights
[1].item():.4f}")
```

#A Access the MoE block of the layer we want to inspect.
#B Retrieve the routing weights saved in last_routing during the last forward pass.

The function runs a normal forward pass on the input text and, instead of keeping the final output of the model, accesses the MoE Block of the specified layer and retrieves the routing weights that this block stored internally in `last_routing` during that same forward pass.

Let's run it with both prompts:

```
analyze_routing("What's the most significant building in Barcelon
a?",
                layer_index=23)
```

We obtain:

```
Routing analysis (layer 23):
Token               | Expert 0 (General)   | Expert 1 (Clinical)
--------------------------------------------------------------------
What                | 0.5142               | 0.4858
's                  | 0.5996               | 0.4004
the                 | 0.5796               | 0.4207
most                | 0.5566               | 0.4436
significant         | 0.6035               | 0.3965
building            | 0.5894               | 0.4104
in                  | 0.5430               | 0.4570
Barcelona           | 0.6060               | 0.3943
?                   | 0.5239               | 0.4758
```

With the call:

```
analyze_routing("Patient is a 67-year-old male with chronic pain", l
ayer_index=23)
```

We obtain:

```
Routing analysis (layer 23):
Token               | Expert 0 (General)   | Expert 1 (Clinical)
--------------------------------------------------------------------
Patient             | 0.5171               | 0.4829
is                  | 0.5049               | 0.4949
a                   | 0.5288               | 0.4712
                    | 0.5039               | 0.4958
6                   | 0.5117               | 0.4880
7                   | 0.5264               | 0.4734
-                   | 0.5234               | 0.4766
year                | 0.6006               | 0.3997
-                   | 0.5728               | 0.4275
old                 | 0.4648               | 0.5352
male                | 0.4609               | 0.5391
with                | 0.3804               | 0.6196
chronic             | 0.4685               | 0.5312
pain                | 0.5483               | 0.4514
```

In the question about Barcelona, all tokens receive a slightly higher weight toward Expert 0, hovering around 0.55–0.60. There is nothing in that text to suggest clinical content, and

the router reflects this with a consistent, though mild, preference toward the general expert.

In the clinical note, the pattern shifts in a recognizable way. The first tokens, "Patient is a 67", maintain weights close to equilibrium, still without enough context to decide. But starting from "old", "male", "with", and especially "chronic", the weight leans toward Expert 1, reaching its highest point on "with" at 0.62.

The execution we are performing is called *soft routing,* where two or more experts combine their outputs. In our small model with only two experts, we are running both of them, which achieves specialization, but we are activating all available model weights.

The structure we have defined is not the most common one because you usually work with a larger number of experts, often dozens, activating only the ones the router considers most relevant for that token on each forward pass. This selective activation is what makes MoE attractive from an efficiency standpoint, since the model can have a very large total capacity without inference cost growing in the same proportion. In our case, with just two experts, we will see that logic in its simplest form in the next section, implementing *hard routing,* where each token is processed by only one of the two experts.

9.5 Implementing hard routing

To implement *hard routing* in our MoE, we just need to change how it behaves at inference time. The weights of both experts and the router are the same ones we just trained — no second training required.

During inference, instead of combining the outputs of both experts, the router makes a binary decision per token, and

that token is processed only by the chosen expert. That is the core idea that makes MoE attractive at scale: activating only the experts considered necessary. Figure 9.2 shows the difference between soft and hard routing on the same token.

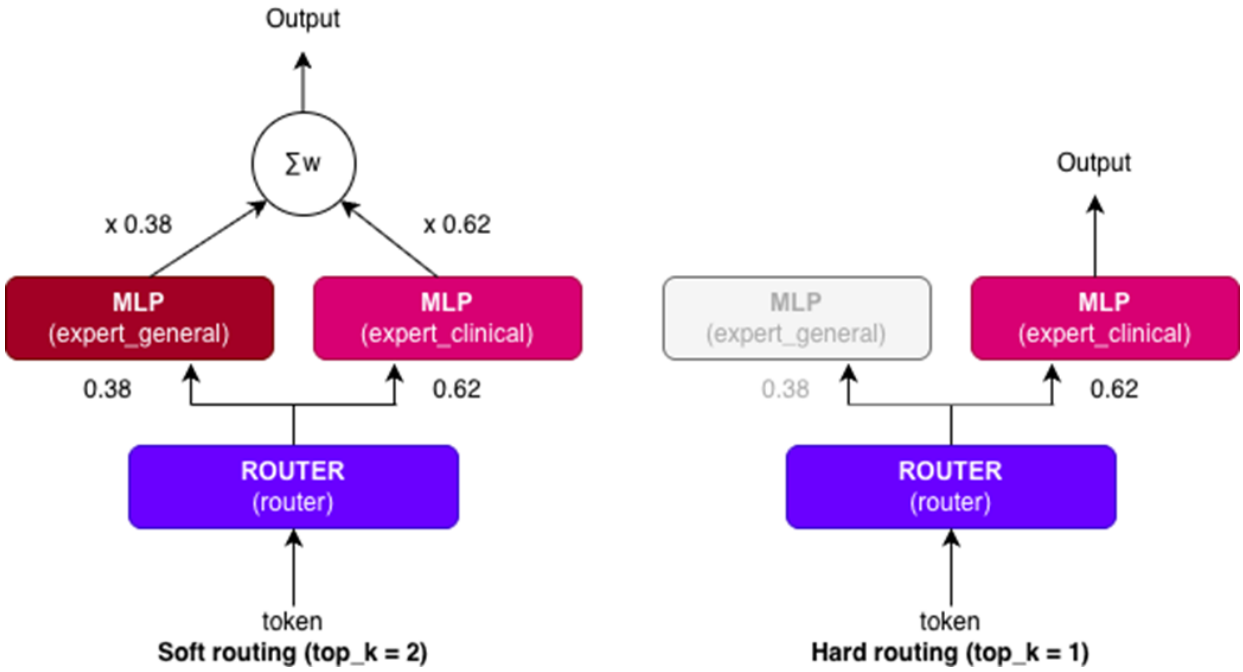


Figure 9.2 Two routing behaviors applied to the same token. Soft routing (left, top-k = 2) runs both experts in parallel and combines their outputs in a weighted sum — × 0.38 for `expert_general`, × 0.62 for `expert_clinical`. Hard routing (right, top-k = 1) selects `expert_clinical` via argmax and passes its output through directly.

Let's see how the `Trainable2ExpertMoE` class looks with the new forward method.

Listing 9.8 Trainable2ExpertMoE hard inference

```
class Trainable2ExpertMoE(nn.Module):
    def __init__(self, original_mlp, hidden_size):
        ... #A

    def forward(self, hidden_states):
        router_logits = self.router(hidden_states)
        routing_weights = F.softmax(
            router_logits.float(),
            dim=-1).to(hidden_states.dtype)

        if self.training:
        .... #A
        else:
            expert_choice = torch.argmax(
                router_logits,
                dim=-1) #B

            output = torch.zeros_like(hidden_states)

            mask_general = (expert_choice == 0)
            mask_clinical = (expert_choice == 1)

            if mask_general.any():
                output[mask_general] = self.expert_general(
                    hidden_states[mask_general]) #C
            if mask_clinical.any():
                output[mask_clinical] = self.expert_clinical(
                    hidden_states[mask_clinical]) #C

            hard_decisions = F.one_hot(expert_choice, num_classes=
2).float()
            self.last_routing = hard_decisions.detach()

        return output
```

#A The __init__ function and the training branch don't change.
#B We take the index of the highest logit.
#C Each expert only processes the tokens the router has assigned to it.

The `__init__` method does not change compared to Listing 9.1 — it still creates Expert 0, Expert 1 as a deep copy, and

the router. The only new thing is in `forward`, and only in the branch that runs when the model is in evaluation mode, meaning when `self.training` is False.

We start by computing `expert_choice` using `torch.argmax` on the router logits. It returns the position with the highest value between the two logits. If the general expert's logit is higher, `expert_choice` is 0 for that token. If it is the clinical expert's logit, it is 1.

With that decision made, we create `output` as an empty tensor, filled with zeros, with the same shape as `hidden_states`. It is just a container where we will place results as each expert processes its portion.

Then we build two boolean masks: `mask_general`, which is `True` for every token that the router assigned to expert 0, and `mask_clinical`, which is `True` for every token assigned to expert 1. A boolean mask is simply a list of true/false values the same size as the token sequence, marking which positions meet a condition. Because the router picks exactly one expert per token, each token appears in only one mask.

Instead of passing all tokens through both experts as we did with *soft routing*, we use those masks to select which tokens are sent to each expert. Each expert receives only its portion of the sequence.

The result of each expert is placed back in its original position using the zero-initialized tensor we preallocated earlier (`output = torch.zeros_like(hidden_states)`). We use the same boolean mask as a write index: `output[mask_general] = self.expert_general(hidden_states[mask_general])`. This is crucial to avoid losing the order. Even though the tokens have been separated to be processed by different experts, the mask

acts as a map that guarantees each token's output goes back to exactly its original position within the output tensor.

At the end, the output tensor contains the complete and perfectly ordered sequence, with the guarantee that each token has been processed by a single expert.

Finally, we save `hard_decisions` using `F.one_hot`, which converts the `expert_choice` integer (0 or 1) into a two-position vector with a 1 in the chosen position and a 0 in the other. We save it in `last_routing` with the same format we used in soft routing, so `analyze_routing` can keep working without changes. It will just see ones and zeros instead of continuous weights.

Now we need to attach this new `forward` to the layers that contain the MoE block:

Listing 9.9 Patching blocks with the hard routing forward

```
for layer_idx in LAYERS_TO_MODIFY:
    old_block = model.model.layers[layer_idx].mlp
    new_block = Trainable2ExpertMoE(
                    old_block.expert_general,
                    hidden_size)

    new_block.expert_clinical.load_state_dict(
                     old_block.expert_clinical.state_dict())
    new_block.router.load_state_dict(
                     old_block.router.state_dict())
    model.model.layers[layer_idx].mlp = new_block
```

We iterate over the same eleven layers we modified in Listing 9.2. We store the block to be replaced in `old_block`, so we can access its weights later.

Then we create a new block with `Trainable2ExpertMoE`. Because its `__init__` method creates a copy of the MLP weights from the original model, we need to restore the

router weights and the clinical expert weights we had saved in `old_block`.

The result is that `new_block` has exactly the same knowledge as `old_block`; the only thing that changes is the `forward` method it executes, which is now the hard routing version.

We will run the same analysis we did with soft routing, but now with the model in evaluation mode. The `analyze_routing` function does not change, since `last_routing` still stores the same thing, now with 0 or 1 values instead of continuous weights.

```
analyze_routing("What's the most significant building in Barcelon
a?",
                layer_index=23)
```

The response is:

```
Routing analysis (layer 23):
Token              | Expert 0 (General)   | Expert 1 (Clinical)
-------------------------------------------------------------
What               | 1.0000               | 0.0000
's                 | 1.0000               | 0.0000
the                | 1.0000               | 0.0000
most               | 1.0000               | 0.0000
significant        | 1.0000               | 0.0000
building           | 1.0000               | 0.0000
in                 | 1.0000               | 0.0000
Barcelona          | 1.0000               | 0.0000
?                  | 1.0000               | 0.0000
```

For the question about Barcelona, all tokens go to Expert 0. There is no ambiguity; there are no intermediate weights.

Let's see the result with the clinical prompt:

```
analyze_routing("Patient is a 67-year-old male with chronic pain”)
```

We obtain:

```
Routing analysis (layer 23):
Token               | Expert 0 (General)   | Expert 1 (Clinical)
------------------------------------------------------------------
Patient             | 1.0000               | 0.0000
is                  | 0.0000               | 1.0000
a                   | 1.0000               | 0.0000
                    | 0.0000               | 1.0000
6                   | 0.0000               | 1.0000
7                   | 1.0000               | 0.0000
-                   | 1.0000               | 0.0000
year                | 1.0000               | 0.0000
-                   | 1.0000               | 0.0000
old                 | 0.0000               | 1.0000
male                | 0.0000               | 1.0000
with                | 0.0000               | 1.0000
chronic             | 0.0000               | 1.0000
pain                | 1.0000               | 0.0000
```

We can see how tokens are distributed between the two experts. Since it's an all-or-nothing decision, we lose the gradation we had in soft routing, but there's a clear tendency: tokens with explicit clinical load like "old", "male", or "chronic" go to the Clinical Expert, while more neutral ones stay with the General Expert.

The routing doesn't work token by token, though. Function words like "with" are sometimes sent to the Clinical Expert, while an unambiguous clinical term like "Patient" stays with the General Expert. Which expert a token gets depends on what the router learned during training, not on any inherent property of the token itself. The "pain" illustrates this well because it stays with the General Expert, but if we modify the prompt to say "chronic back pain" the result changes, "back" goes to the General Expert and "pain" moves to the Clinical Expert. This is because the router works with the hidden states that accumulate context from previous tokens.

NOTE

We've been calling the new expert Clinical Expert for convenience and clarity, but we have no direct evidence that its weights have developed a specialized internal representation for clinical data. What we do know is that it was trained on clinical data and that the router tends to send it tokens with clinical load. The specialization, if it exists, is a result of training, not something we can directly observe in the weights.

As for schema compliance, the result is identical to soft routing, 97.5%. Meaning the model achieves the same performance but running only one of the two experts instead of combining the output of both.

9.6 How routing shapes model behavior

In the previous sections we have seen two models, the base and the MoE, and we have tested the latter with two different forward functions. This last part is important, because the model has exactly the same weights, but we get different results due to the processing of the responses by its forward methods.

Let's look at the outputs of the control prompt about Barcelona:

PM

What's the most significant building in Barcelona?

The most significant building in Barcelona is the Sagrada Familia, a large-scale Roman Catholic church designed by Antoni Gaudí. It is one of the most famous buildings in the world and a major tourist attraction. The Sagrada Familia is a UNESCO World Heritage Site and is considered a masterpiece of Catalan Modernism.

The most significant building in Barcelona is the Sagrada Familia, a large-scale Roman Catholic church designed by Antoni Gaudí. It is one of the most famous buildings in the world and a major tourist attraction.

The most significant building in Barcelona is the Sagrada Familia, a large-scale Roman Catholic church designed by Antoni Gaudí.

The base model describes the Sagrada Familia, its architecture, its tourist relevance, and its UNESCO World Heritage status. The soft model trims the response, we lose the UNESCO detail, and the hard model reduces it to a single sentence.

Curiously, in the matching text the responses are identical. In other words, the only thing that changes is when the model decides to stop generating tokens. This behavior may be due to the new expert having learned to generate

structured JSON, a closed-ended generation task where the response is concise and ends as soon as the schema is complete, which may have led it to close generation earlier.

This is a control test based on a single prompt, intended to give a general sense of how it affects model behavior. In Figure 9.3 we find the results of the five general benchmarks, which we already used in Chapter 7, comparing the base model with the two MoE models.

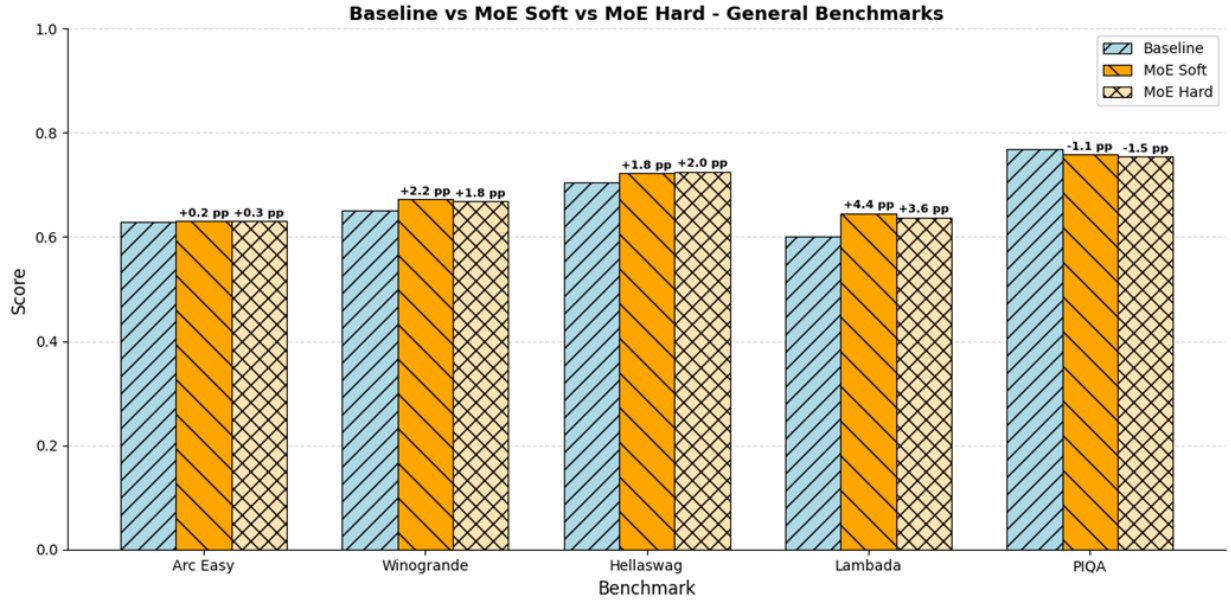


Figure 9.3 Benchmark comparison across five general-knowledge tasks for the base model, MoE Soft, and MoE Hard. Delta values expressed in percentage points (pp) relative to the baseline.

In four of the five benchmarks, the two MoE models improve over the baseline, with gains ranging from +0.17 pp in ARC-Easy to +4.37 pp in Lambada. The exception is PIQA, where both models drop slightly, -1.14 pp in soft and -1.52 pp in hard. PIQA evaluates commonsense reasoning about real-world physical actions, a type of knowledge very far from the clinical domain in which we specialized Expert 1. The movements are small in all cases and don't allow drawing definitive conclusions, but the general pattern suggests that clinical specialization hasn't damaged the model's general

capabilities. If we had to venture a conclusion, MoE improved the model's overall behavior at the same time it specialized it.

In the two previous experiments, we trained the new expert starting from a copy of the base model weights. But if you remember, in Chapter 7 we already trained a model specialized in clinical extraction. In the next section, we'll use the work already done to create the new expert starting from the previously fine-tuned model.

9.7 Reusing a fine-tuned model as an expert

In this section, instead of starting from a copy of the base model, we'll use the MLP Module of the model created in Chapter 7 as Expert 1 within the MoE architecture. Since we already have the expert, we can limit ourselves to training the router, as the MLP Module is in principle already adapted to the task of extracting clinical entities. But we'll run into the problem that our previously fine-tuned model wasn't designed to be part of a MoE, it was meant to work autonomously, and in addition to the MLP layers, the attention layers were also modified. To solve this, we'll subject the expert to some light additional training that lets it adapt to working with the base model's attention layers, which, as you'll recall, haven't been fine-tuned for the clinical entity extraction task.

We'll also take the opportunity to update the class that controls the MoE, generalizing it to accept a configurable number of experts and to specify how many we want active for each response.

The code for this section is available in the notebook CH09_NB02_MOE_Upcycling_TopK.ipynb in the book's

repository.

9.8 Creating a new MoE class

We're going to start from the `Trainable2ExpertMoE` class and create a new one called `TrainableTopKMoE`. This new class will be able to receive two additional variables that let us specify the number of experts the model can have and the number of experts it should consider when generating the response.

Listing 9.10 TrainableTopKMoE signature

```
class TrainableTopKMoE(nn.Module):
    def __init__(self, original_mlp, hidden_size, num_experts=2, top
_k=2,
                 extra_experts=None):
        ...

    def forward(self, hidden_states):
        ...
```

As you can see, the `__init__` method signature is different and reflects the main changes. The parameters `num_experts` and `top_k` appear. The first contains the number of experts the MoE will need to manage, and the second the number of experts activated per token during inference (this is the parameter that replaces the soft/hard distinction we used in the previous section). The `extra_experts` parameter lets us pass different MLP modules that will be used when creating the experts. We will use it to pass the MLP module from the pretrained model in chapter 7. The `forward` method keeps the same signature, receiving the hidden states and returning the combined output. What changes is its internal behavior.

With the skeleton clear, let us look at the `__init__` method. This is where the most delicate part of the transplant happens: deciding where each expert comes from.

Listing 9.11 TrainableTopKMoE.__init__

```
def __init__(self, original_mlp, hidden_size, num_experts=2, top_k=
2,
             extra_experts=None):
    super().__init__()
    target_dtype  = next(original_mlp.parameters()).dtype #1
    target_device = next(original_mlp.parameters()).device #1

    self.num_experts = num_experts
    self.top_k = top_k

    experts_list = [original_mlp] #A
    extra_experts = extra_experts or []
    for e in extra_experts:
        experts_list.append(
            copy.deepcopy(e).to(device=target_device, dtype=target_d
type)
        ) #B

    while len(experts_list) < num_experts:
        experts_list.append(
            copy.deepcopy(original_mlp).to(
                device=target_device,
                dtype=target_dtype)
        ) #C

    self.experts = nn.ModuleList(experts_list) #D
    self.router = nn.Linear(hidden_size, num_experts, bias=False,
                            dtype=target_dtype,
                            device=target_device) #E
    self.last_routing = None
```

#A Expert 0 is still the original MLP module of the Transformer block.
#B Pre-trained experts passed in extra_experts, are copied directly as Expert 1, Expert 2, and so on.
#C If experts are left to define, they are filled with copies of the original MLP.
#D Experts are stored in a module list indexable by position.
#E The router projects the hidden state to num_experts logits.

The method receives the original MLP module (which will become the first expert), the hidden size for the router, how

many total experts you want, how many are activated per token (top_k), and a list of already trained experts, in our case the clinical expert.

We then retrieve the precision and device from the original MLP module, so all new experts are created with the same precision and on the same device. The following lines store the hyperparameters as class attributes so they can be accessed later in the `forward` method.

We save the original MLP as the first expert with `experts_list = [original_mlp]`. We do not copy it; we reuse the same instance since it is the model's MLP. Then we continue with the loop that copies the extra experts with a `deepcopy` for each expert received, this is where the transplanted clinical MLP module from Chapter 7 will land once we build that model in the next section. If we need to create more experts than those received in `extra_experts`, copies of the original MLP Module are made.

With the experts already created and stored in `self.experts`, we create the routing layer. This is a simple linear projection from `hidden_size` to `num_experts`, which, given the hidden state of a token, produces one logit per expert. This is what will be used later, with softmax and top-k, to decide which experts process each token.

Let us look at the new implementation of the `forward` method, which is capable of managing N experts and returning the response formed by K experts.

Listing 9.12 TrainableTopKMoE.forward

```
def forward(self, hidden_states):
    router_logits = self.router(hidden_states)
    if self.training:
        routing_weights = F.softmax(
            router_logits.float(), dim=-1).to(
                    hidden_states.dtype) #A

        output = torch.zeros_like(hidden_states)
        for i in range(self.num_experts):
            output = output + self.experts[i](hidden_states) * \
                routing_weights[..., i].unsqueeze(-1) #B

        self.last_routing = routing_weights.detach()
    else:
        topk_values, topk_indices = torch.topk(
            router_logits, k=self.top_k, dim=-1) #1
        topk_weights = F.softmax(
            topk_values.float(), dim=-1).to(
                    hidden_states.dtype) #C

        output = torch.zeros_like(hidden_states)
        full_weights = torch.zeros_like(router_logits)

        for expert_idx in range(self.num_experts):
            is_selected = (topk_indices == expert_idx) #2
            if not is_selected.any():
                continue
            token_mask = is_selected.any(dim=-1)
            weight_per_token = (topk_weights * is_selected).sum(dim=
-1)

            expert_out = self.experts[expert_idx](
                                    hidden_states[token_mask]) #3
            output[token_mask] += expert_out * \
                weight_per_token[token_mask].unsqueeze(-1) #D
            full_weights[..., expert_idx] = weight_per_token

        self.last_routing = full_weights.detach()

    return output
```

#A During training, softmax is calculated over all experts, without prior selection.
#B Each expert processes the entire sequence and its output is weighted by the weight assigned by the router.
#C Softmax is renormalized only over the selected experts, their weights add up to 1 among themselves again.
#D The result is written back to its original position using the same mask, and it's scaled by the corresponding weight.

As we can see, the method is divided into two sections: one that runs during model training and one that runs at inference time. In both cases, we start by calling the router created in the `__init__` function, which returns, for each token, a vector of size `num_experts` indicating how much weight is assigned to each expert for that token.

When the model is in training mode, the forward computes the softmax over the logits of all experts without filtering anything first. Each expert processes the full sequence, and its output is weighted by the weight the router assigned to it for that token.

The weighted sum of all those outputs is the final output. It is the same mechanics you already know from `Trainable2ExpertMoE`, just now generalized to `num_experts` instead of two, and without the Gaussian noise we were adding before. There is no need for it, because in this chapter the expert we transplant already arrives trained and there is no risk of the router collapsing toward it from the first step. It already has a sufficiently differentiated behavior for some tokens to prefer one expert over another.

The result is stored in `self.last_routing` so we can explore it later with the `analyze_routing` function, but in production we could just discard that value.

In inference mode, the logic is different because we no longer use all the experts. We use `torch.topk` on `router_logits`

to select, for each token, the `top_k` highest logits, storing both their values and the corresponding expert indices #1. This is where, and only here, the method decides which experts each token will use. The rest of the method does not revisit that decision, it just executes it. The softmax is recalculated only over those selected logits, so the weights sum to 1 across the chosen experts, not across all available ones.

With the selection already made, the loop that follows has a different job. It does not select, it reorganizes. It goes through all the experts in the block, one by one, and for each one, builds a boolean mask, `is_selected` #2, that answers the question "which tokens belong to this expert." In Figure 9.4, you can see the selection and reorganization process.

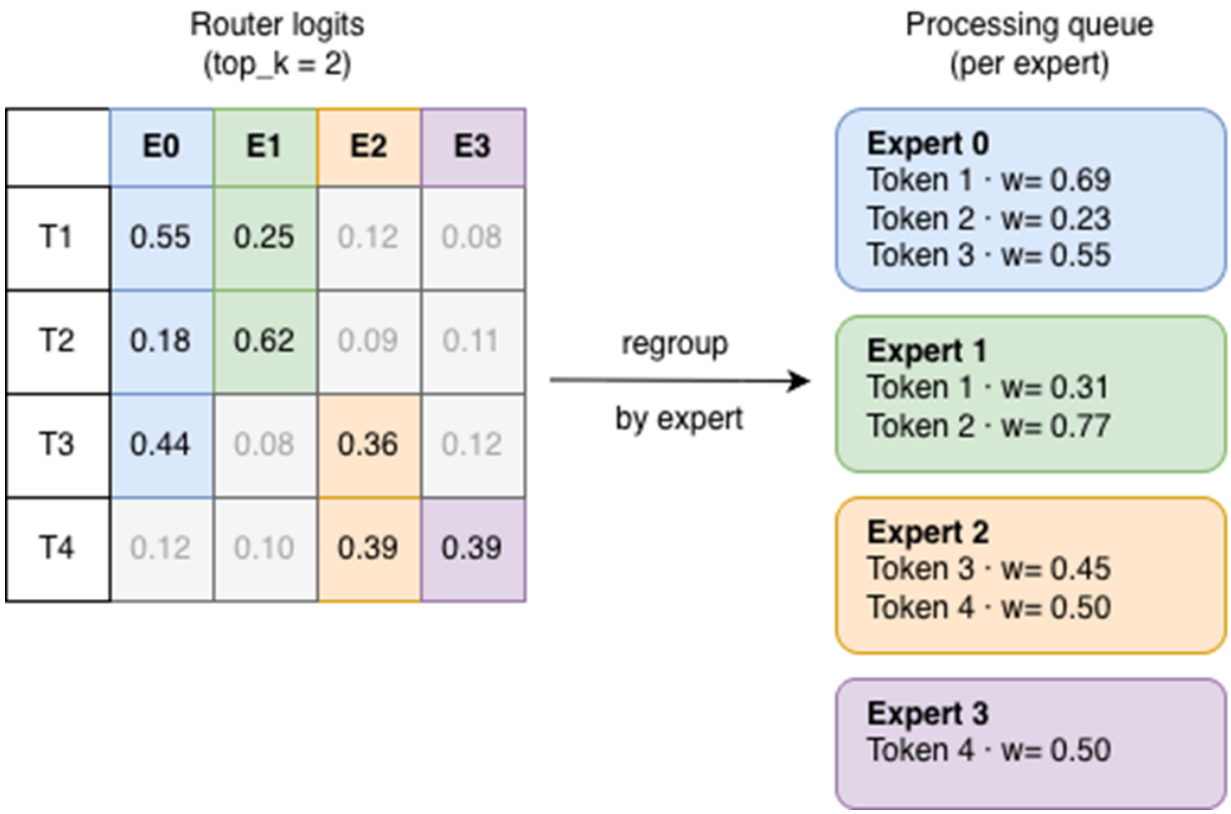


Figure 9.4 Two views of the same top-k routing decision. Left: the router produces one logit score per expert for each token; the two highest scores per row (highlighted) are selected. Right: the same assignments reorganized by expert, showing which tokens each expert will process and their renormalized weights — for each token, those weights sum to 1 across its assigned experts. The example uses four experts rather than two so that top_k = 2 leaves half the experts idle per token. The loop in `TrainableTopKMoE.forward` performs exactly this regrouping.

It is possible that an expert has no tokens to process, in which case the loop moves on to the next one without doing extra work; this is what makes the approach sparse, as not all experts process data on every pass. If there are tokens assigned, that expert processes only its portion of the sequence, `hidden_states[token_mask]` #3, not the full batch. This line is where compute is actually saved; the rest of the loop is just logistics to get here. The result is written back into output using the same mask as an index, and scaled by the weight that corresponds to each token within that expert.

At the end, `last_routing` stores these sparse weights, with zeros for the experts that were not selected, so `analyze_routing` can keep working without changes, just as you did in 9.2.2. Outside the notebook, and without needing to inspect expert traffic, we could discard the `last_routing` variable entirely.

9.9 Transplant the expert and router-only training

Now that the MoE class is ready, we just need to create the model. To do that, you first extract the already trained MLPs from the Chapter 7 model and transplant them directly as Expert 1. Then you train only the router, keeping both Expert 0 and Expert 1 frozen, to see if it's enough to teach it how to distribute traffic between two experts that already know what they're doing. Finally, we'll look at the results to decide the next steps.

TRANSPLANTING THE EXPERT

The Chapter 7 model is available on Hugging Face, so we just need to download it and grab its MLP module.

Listing 9.13 Extracting the clinical MLPs

```
LAYERS_TO_MODIFY = range(model.config.num_hidden_layers)
hidden_size = model.config.hidden_size

clinical_mlps = {}
clinical_model = AutoModelForCausalLM.from_pretrained(
    "oopere/SmolLM2-1.7B-ClinicalNER",
    torch_dtype=torch.float16,
    device_map="cpu"
) #A

for layer_idx in LAYERS_TO_MODIFY:
    clinical_mlps[layer_idx] = copy.deepcopy(
        clinical_model.model.layers[layer_idx].mlp
    ) #B

del clinical_model #C
```

#A We load the chapter 7 model directly into the CPU.
#B We extract the MLP module from each of the 24 blocks and save it in a dictionary.
#C We remove the model from memory since it's no longer necessary.

Since we're working in a memory-constrained environment, we load the model on the CPU, retrieve the MLP layers we're interested in, and then remove the model from memory.

So now it's time to build the MoE using the two MLPs we have available: the one from the base model and the one we just recovered from the pre-trained model.

Listing 9.14 Transplanting into the MoE blocks

```
for layer_idx in LAYERS_TO_MODIFY:
    base_mlp = model.model.layers[layer_idx].mlp
    model.model.layers[layer_idx].mlp = TrainableTopKMoE(
        base_mlp, hidden_size, num_experts=2, top_k=2,
        extra_experts=[clinical_mlps[layer_idx]]
    ) #A

del clinical_mlps #B
```

#A Each MoE block is created with its original MLP as Expert 0 and its corresponding clinical MLP as Expert 1.
#B We don't need the MLP layers; they are already inside the MoE blocks.

With this loop, you replace the original MLP module of each Transformer Block with its corresponding MoE block. The result is 24 `TrainableTopKMoE` blocks, each with two experts: a general one inherited from the base model and a clinical one inherited from Chapter 7, and a router that doesn't know anything yet, because it's a freshly created linear layer.

The next step is to teach the router to choose between the two experts.

TRAINING THE ROUTER

With both experts already in place, the freezing of this phase is much simpler than in section 9.2.1. We only have to unfreeze the router.

Listing 9.15 Freezing everything except the router

```
for param in model.parameters():
    param.requires_grad_(False) #A

for layer_idx in LAYERS_TO_MODIFY:
    for param in model.model.layers[layer_idx].mlp.router.parameters
():
        param.requires_grad_(True) #B
```

#A We start by freezing the entire model, just like in 9.2.1.
#B We unfreeze only the router of each of the 24 MoE blocks. Expert 0 and Expert 1 remain intact.

In section 9.2.1 you trained the router and Expert 1 at the same time, which is 24.44% of the model. Here, with both experts frozen, the only trainable component is the router, which represents 0.0034% of the model weights.

That simplicity is also reflected in the learning rate. In notebook 1, the router was set to 1e-6, deliberately low because it had to synchronize with Expert 1's training without destabilizing it. Here there is no expert learning in parallel, so the router can afford a much more aggressive pace:

```
...
 optimizer = optim.AdamW([
      {"params": router_params, "lr": 1e-3},
  ])
...
```

The rest of the training loop is identical to the one you already saw in Listing 9.6, same forward with `autocast`, same language modeling loss, same `GradScaler`, so we don't repeat it here.

With the trained router we can evaluate it on the clinical entity extraction task and we get 90% compliance, whether

we use `top_k=2` or `top_k=1`.

```
set_top_k(model, 2)
moe_results_k2_faseA = evaluate_moe_on_test_set()
print(f"\n(top_k=2): {moe_results_k2_faseA_score:.1f}%")
```

We obtain:

```
              samples  valid_json_pct  schema_ok_pct
category
abbreviations        6           100.0          100.0
clean                8           100.0          100.0
implicit             7           100.0          100.0
irrelevant          12            66.7           66.7
typos                7           100.0          100.0
MoE overall schema compliance (top_k=2): 90.0%
```

The expert worked, but we did not achieve 100% compliance, which is what we got with the model in Chapter 7. We have to keep in mind that during the fine-tuning of the model, we did not just modify its MLP modules, we also modified the Attention modules. By extracting the MLP modules and making them work with the base model's Attention modules, without the fine-tuning a desynchronization occurs: the MLP module is used to receiving different information from the Attention module. To fix this, we are going to train the last six MLP modules so they get used to working with the Attention module that accompanies them.

9.10 Fine-tuning the expert tail

The transplanted MLP Module learned to extract clinical entities working side by side with a specific Attention Module, the one fine-tuned with QLoRA in Chapter 7. Now that same MLP Module receives tensors from a different

Attention Module, the base model's, which has not been fine-tuned for the entity extraction task.

The solution is not to retrain the expert from scratch; that would waste all the work from Chapter 7. A small, localized nudge is enough. We target the layers closest to the output, as Figure 9.5 shows; those are the most task-specific, and therefore the most sensitive to the representational shift introduced by swapping the Attention Module.

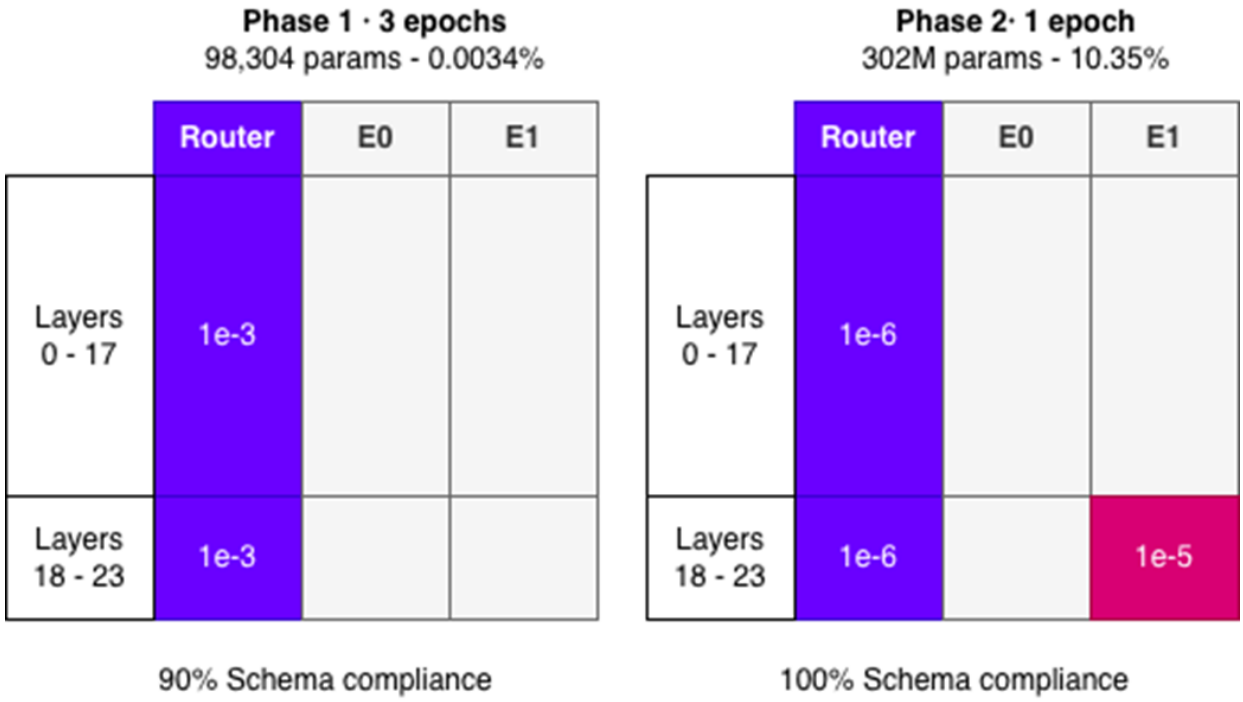


Figure 9.5 Two-phase training strategy for the upcycled MoE. Phase 1 trains only the router across all 24 layers (lr = 1e-3) for three epochs — 98,304 parameters, 0.0034% of the model — reaching 90% schema compliance. Phase 2 reduces the router learning rate to lr = 1e-6 and unfreezes the `expert_clinical` tail layers 18–23 at lr = 1e-5 for one epoch, bringing trainable parameters to 10.35% and compliance to 100%.

UNFREEZING AND TRAINING THE EXPERT TAIL

You start from the same point as in phase 1, with the router already trained and its weights unfrozen, but now you add

something else to the set of trainable parameters: the last six MLP layers of the clinical expert.

Listing 9.16 Unfreezing the expert tail

```
EXPERT_LAYERS_TO_UNFREEZE = range(18, 24)
for layer_idx in EXPERT_LAYERS_TO_UNFREEZE:
    for param in model.model.layers[layer_idx].mlp.experts[1].parame
ters():
        param.requires_grad_(True)
```

Doing an exercise in saving trainable parameters, we will train the weights of the router + layers, which represent 10% of the model's weights, for a single epoch.

```
...
 optimizer = optim.AdamW([
      {"params": router_params,  "lr": 1e-6},
      {"params": expert_params,  "lr": 1e-5},
  ])
...
```

On this occasion, since we only want to make small adjustments, the learning rates are especially low to allow for small tuning steps. The router is already trained and the MLP already knows its task. It only needs to adapt to the Attention layers it works with.

The rest of the training loop is the same as in listing 9.6. Let's see the results with this new training.

RESULTS

If we run the schema compliance again, this time the results are 100%, both with `top_k=1` and `top_k=2`.

```
               samples  valid_json_pct  schema_ok_pct
category
abbreviations        6           100.0          100.0
clean                8           100.0          100.0
implicit             7           100.0          100.0
irrelevant          12           100.0          100.0
typos                7           100.0          100.0
MoE overall schema compliance (top_k=1): 100.0%
```

Let's analyze the path to reach this result. In the first phase, we trained only the router for three epochs, a tiny number of parameters. In the second phase, we added six layers, training 10% of the model's weights for one epoch. We should also consider that if during the fine-tuning in Chapter 7 we had known we wanted to use the model for an MoE, we could have trained only its MLP Modules. But this never happens in real life; we all know that if we find a model we can leverage, there's a 99.99% chance it has been trained on more than just the MLP Module.

Before closing out the MoE construction, it's worth checking that this specialization hasn't taken a toll on the model's general behavior.

As you can see in Figure 9.6, the pattern is practically the same as in notebook 1. Improvements were made in WinoGrande, HellaSwag, and especially Lambada, with a small drop in PIQA and, in this model, also in ARC-Easy. Nothing particularly concerning: as with the previous model, we get a specialist that performs as well as or better than the base model on general benchmarks.

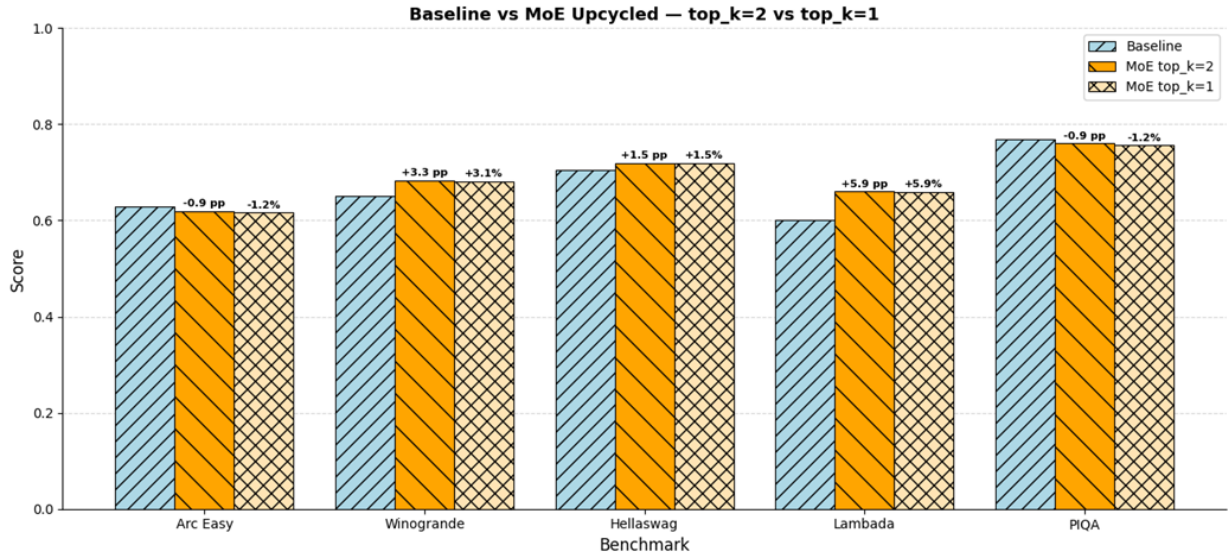


Figure 9.6 Benchmark comparison across five general-knowledge tasks for the upcycled MoE, baseline vs top_k=2 vs top_k=1.

We've built a MoE at a very modest scale. The labs that popularized MoE work with a similar logic but in a different order of magnitude, dozens or hundreds of experts, models with hundreds of billions of parameters, where the technique was born to solve a large-scale computational cost problem. What you just built is that same idea adapted to a different setting: an SLM that can afford few experts, focused on specific tasks, without needing the distributed training machinery that a frontier model requires.

9.11 From paper to practice

What you have built in the two previous sections has a very strong foundation in the literature. MoE model creation is a field where research is advancing rapidly, driven by the big labs that have adopted it as the structure for many of their models (some we know for sure, others we can only suspect). Each decision, from duplicating the original MLP to transplanting an already specialized expert, has its origin in specific research papers, usually applied at a scale far beyond a Colab with a T4. We will review two papers, one

for each approach you followed: Sparse Upcycling (Komatsuzaki et al., 2022), which formalizes the mechanism behind 9.2, and Mixtral of Experts (Jiang et al., 2024), which puts the architectural decisions of 9.3 into context.

9.12 Sparse upcycling

Komatsuzaki et al. proposed sparse upcycling in 2022 (https://arxiv.org/abs/2212.05055) on T5, an encoder-decoder model with a different architecture from what we find in today's LLMs, and on Vision Transformer, which shows that the idea is not tied to text, it works just as well initializing a vision MoE from a dense ViT. You applied it to SmolLM2-1.7B-Instruct, a decoder-only LLM, but the underlying mechanism is the same as in Listing 9.1: duplicating the already-trained dense module to populate the experts instead of initializing them randomly.

The results the authors report are compelling; their upcycled T5 and ViT models outperform both their dense versions and a sparse MoE trained from scratch with the same compute budget, and they do it using around 50% of the initial dense pretraining cost, though this varies by model.

That said, what you did in section 9.2 is not an exact replication of the paper. We introduced several modifications to adapt it to our reality: less data and less compute. They train all experts after initialization, which means we would have had to train the general expert as well, not just the clinical one. They let the training differences gradually diverge the weights over time, something viable with a large budget and a long training run.

You, on the other hand, froze Expert 0 to preserve the overall behavior of the base model, and added Gaussian noise to the router logits during training. The goal here is to

force the router to explore different assignments from the very first step, accelerating convergence and achieving useful specialization in just three epochs. Without that signal, two experts with identical weights could take much longer to differentiate, or might never do so within the available budget. The paper tested a similar approach: adding Gaussian noise to the parameters of both the router and the experts, but they concluded that if the noise was small it had no effect and if it was large it had a negative impact.

The training cost reduction figures the paper reports — between 40% and 60% depending on the model — do not apply to our case, since we have also frozen the general expert. We have not measured it and it will depend on the number of experts we want to train, but we have shown that we can keep one expert frozen while training the rest. Although that forces us to find a balance point, since a frozen expert does not adapt, and if you overtrain the router it might stop sending it information because it does not adapt like the others.

9.13 Mixtral of Experts

Jiang et al. (2024) introduced Mixtral 8x7B (https://arxiv.org/abs/2401.04088), a model with 32 Transformer blocks where each MLP is replaced by a MoE block with eight experts and top-k routing, selecting two per token. A structure very similar to what you built in section 9.3, but at a different scale. The Mixtral model has 46.7B total parameters, but by activating only two experts per token, the effective computation drops to 12.9B active parameters, which allows it to match or outperform Llama 2 70B on multiple benchmarks at a fraction of the inference cost.

Mistral modifies all the model's blocks to convert them into MoE blocks. That's exactly what we did with notebook NB02, although we transplanted a previously created clinical expert. The selection mechanism also matches in its essentials: the router applies softmax over the top-k logits to produce renormalized weights. The difference with Mixtral is one of scale. Two experts and ~2.9B total parameters versus eight experts and 47B, but the algorithm that decides which expert processes each token has the same underlying logic.

The paper makes a very interesting observation when analyzing the routing to the experts. They expected to find domain specialization — math going to one expert, code to another — and did not find it. The token distribution across experts is almost uniform regardless of the text domain. What they do observe is syntactic specialization. Tokens like `self` in Python or indentation tokens consistently route to the same expert. The router learns structure, not domain semantics. In our case, it has been different. With one frozen expert for general behavior and another already specialized in clinical extraction, the router had to learn to distinguish between domains to be useful.

9.14 Hands-on lab

The notebook `CH09_NB03_HandsOn_3Expert_MoE.ipynb` extends the upcycling design from section 9.3 to three experts: Expert 0 keeps the original base weights, Expert 1 carries the clinical specialization from Chapter 7, and Expert 2 is initialized from a copy of the base MLP and trained on Python code from `flytech/python-codes-25k`. Router and Expert 2 tail layers are trained jointly on a mixture of all three domains. Your primary tool for evaluating the result is `analyze_routing`, which you already used in section 9.2.2. The three exercises

that follow explore the main variables you can adjust once the baseline notebook is running.

- Training depth and routing stability:
  - What to do: Run the baseline notebook with `EPOCHS = 1`, then repeat with `EPOCHS = 2`. After each run, compare the `analyze_routing` output for all three prompts in both layer 18 and layer 23. Pay close attention to the clinical prompt in layer 23. Does Expert 1 remain visible at two epochs, or does it erode in the same direction you observe at one?
  - Questions to answer: Does extending training reinforce or destabilize the routing pattern you saw at one epoch? In which layer and for which prompt does the routing become most differentiated? Is there any expert that consistently receives near-zero weight across all prompts regardless of training length?
- Expert isolation through selective freezing:
  - What to do: Modify the freezing cell so that only Expert 2 tail layers and the router are trainable. Expert 1 stays fully frozen throughout, the same way Expert 0 does. Keep everything else identical: same EPOCHS, same datasets, same EXPERT2_TRAINABLE_LAYERS. After training, run analyze_routing on the three prompts in layers 18 and 23 and compare against the baseline run.
  - Questions to answer: Does keeping Expert 1 fully frozen recover its dominance in layer 23 for clinical tokens? Does Expert 2 receive more or less routing weight on the code prompt compared to the baseline? At which layer does the most visible change occur? What does this experiment tell you about the tradeoff between expert adaptation and expert identity preservation?

- Two-phase training for cleaner routing:
  - What to do: Restructure training into two phases following the same logic as Notebook 2. In Phase 1, freeze both Expert 1 and Expert 2 completely and train only the router for one epoch. In Phase 2, unfreeze Expert 2 tail layers, reduce the router learning rate to `1e-6`, and train for one additional epoch. Compare the `analyze_routing` output against the single-phase baseline from exercise one.
  - Questions to answer: Does a dedicated router-first phase change the starting point for Expert 2 in Phase 2? In layer 18, does the separation between Expert 2 and Expert 0 on the code prompt become sharper after the two-phase approach? Does the clinical routing in layer 23 behave differently compared to the baseline?

9.15 Summary

- Dynamic inference techniques, like early exiting, token dropping, and Mixture of Experts, allow the model to make runtime decisions about what to compute. Of the three, MoE is the most mature and the only one with growing support in standard inference libraries.
- An MoE architecture is built around three elements. Experts process information independently, the router decides which expert acts on each token, and the combined forward pass merges their outputs into a single response.
- Building an MoE starting from a copy of the base model weights allows initializing the experts with already-acquired knowledge, instead of starting from scratch. This strategy, known as sparse upcycling, speeds up convergence.

- Soft routing and hard routing are two inference behaviors that can be obtained on the same trained weights, by modifying only the forward method. The model's behavior depends both on its weights and on how the experts' outputs are combined.
- The router does not work on raw tokens, but on their hidden states, which accumulate the context of everything that has occurred before in the sequence. That is why the same token can be sent to different experts depending on the context in which it appears.
- The `TrainableTopKMoE` class generalizes the architecture to accept a configurable number of experts and a top-k parameter that controls how many of them are activated per token.
- An MLP Module from a pre-trained model can be transplanted directly as an expert into an MoE architecture, without the need to retrain it from scratch. This opens the door to reusing existing specialized models as components of a larger system.
- If the transplanted expert comes from a model with a different Attention Module than the base model, it might need a light adjustment to adapt to the tensors it receives. A light training of the layers closest to the output is usually enough to recover full performance.
- Specializing an expert in a specific domain does not necessarily penalize the general capabilities of the model. General benchmarks of the MoE model built in this chapter are maintained or improved compared to the base model.