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 1.1

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
- 2025-10-05: v1.1

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	2.8.0+cu126	safetensors	0.6.2
transformers	4.56.1	trl	0.23.1
peft	0.17.0	bitsandbytes	0.47.0
pandas	2.2.2	datasets	4.0.0
huggingface-hub	0.34.4	ollama	0.9.6
numpy	2.0.2	unsloth	2025.8.9
matplotlib	3.10.0	xformers	0.0.32.post2
accelerate	1.10.0	gguf	0.17.1
mistral-common	1.8.4	protobuf	3.20.1

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 contextlib import nullcontext
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 in **conversational** format.

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

In previous versions, it would suffice to have two columns named `prompt` and `completion`, and the dataset would be automatically converted to the conversational format. This approach isn't supported anymore, so we're better off doing the conversion ourselves. We'll be using the `format_dataset()` function below, which was adapted from an earlier version of the `trl` package.

```
# Adapted from trl.extras.dataset_formatting.instructions_formatting_function
# Converts dataset from prompt/completion format to the conversational 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(converted_sample)
        return {'messages': output_texts}
    else:
        converted_sample = [
            {"role": "user", "content": examples["prompt"]},
            {"role": "assistant", "content": examples["completion"]},
        ]
        return {'messages': converted_sample}
```

Notice that the function expects examples with both `prompt` and `completion` columns, so we need to rename some columns before applying it to our dataset.

```
dataset = dataset.rename_column("sentence", "prompt")
dataset = dataset.rename_column("translation_extra", "completion")
dataset = dataset.map(format_dataset)
dataset = dataset.remove_columns(["prompt", "completion", "translation"])
messages = dataset[0]["messages"]
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.'}]
```

That's it for formatting!

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)

It is important to keep the `EOS` token unique. In some models, such as Phi-3, the `EOS` token also serves as the `PAD` token, which can cause it to be masked during training and result in endless token generation. To prevent this, we map the `PAD` token to `UNK` instead.

```
tokenizer = AutoTokenizer.from_pretrained(repo_id)
# Assign the UNK token to the PAD token to make EOS token unique
tokenizer.pad_token = tokenizer.unk_token
tokenizer.pad_token_id = tokenizer.unk_token_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_length=64,  # renamed in v0.20
    # Dataset
    # packing a dataset means no padding is needed
    packing=True,
    packing_strategy='wrapped', # added in v0.20

    ## 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',
    ## EXTRA
    # ensures that training in bf16 happens only when it is supported
    bf16=torch.cuda.is_bf16_supported(including_emulation=False)
)
```

At the time of writing, the current version of `trl` (0.23.1) enables mixed-precision training using the `bf16` data type by default, regardless of whether the environment supports it. To prevent issues, we explicitly set the appropriate `bf16` argument in the configuration.

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,
)
```

Up to version 0.23 of `trl`, there was a known issue where training failed if the LoRA configuration had already been applied to the model, as the trainer froze the whole model, including the adapters.

If the model already contained the adapters, as in our case, training would only work if the underlying original model (`model.base_model.model`) was used together with the `peft_config` argument.

This issue was fixed in version 0.23.1 of `trl`, released in October 2025.

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
50	1.362300
100	0.607900
150	0.353600
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).
    - If the model was trained using mixed-precision, we wrap the generation in the `autocast()` context manager, which automatically handles conversion between data types.
- 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")
    tokenized_input = tokenized_input.to(model.device)
    model.eval()
    # if it was trained using mixed precision, uses autocast context
    ctx = torch.autocast(device_type=model.device.type, dtype=model.dtype) \
          if model.dtype in [torch.float16, torch.bfloat16] else nullcontext()
    with ctx:
      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:

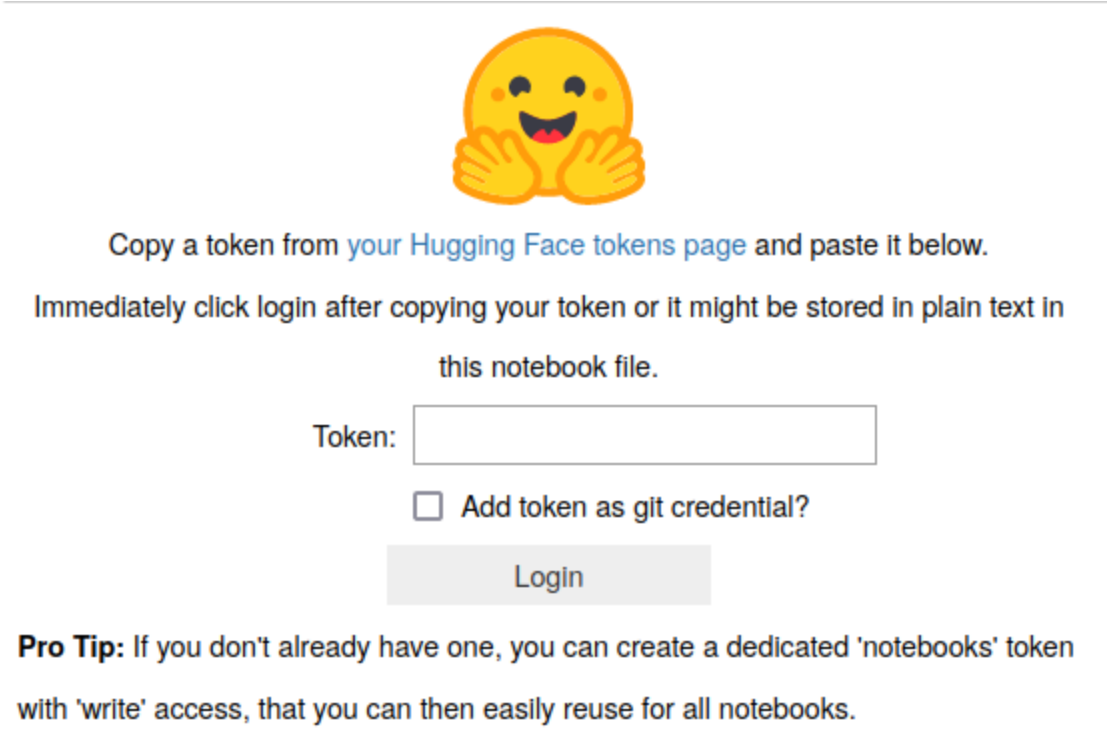

Figure 0.1 - Logging into the Hugging Face Hub

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

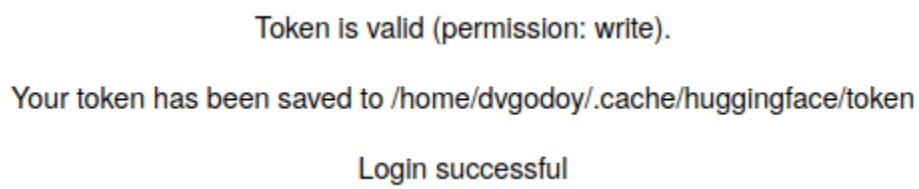

Figure 0.2 - Successful Login

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.

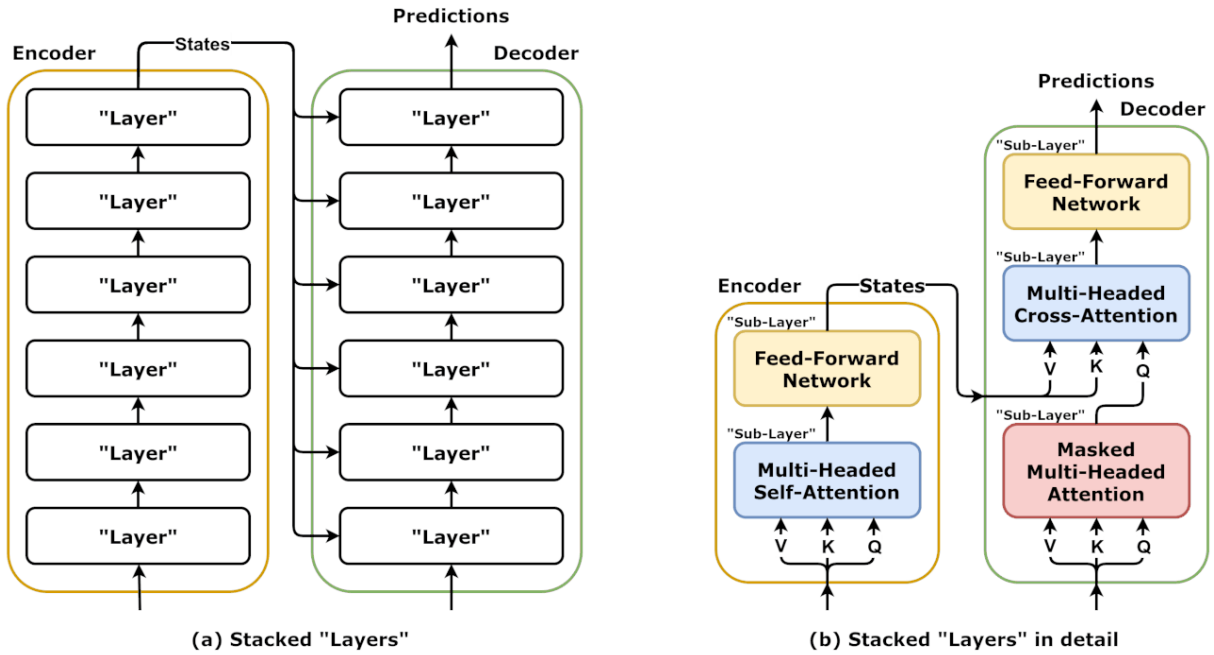

Figure 1.1 - Transformer's stacked "layers"

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.

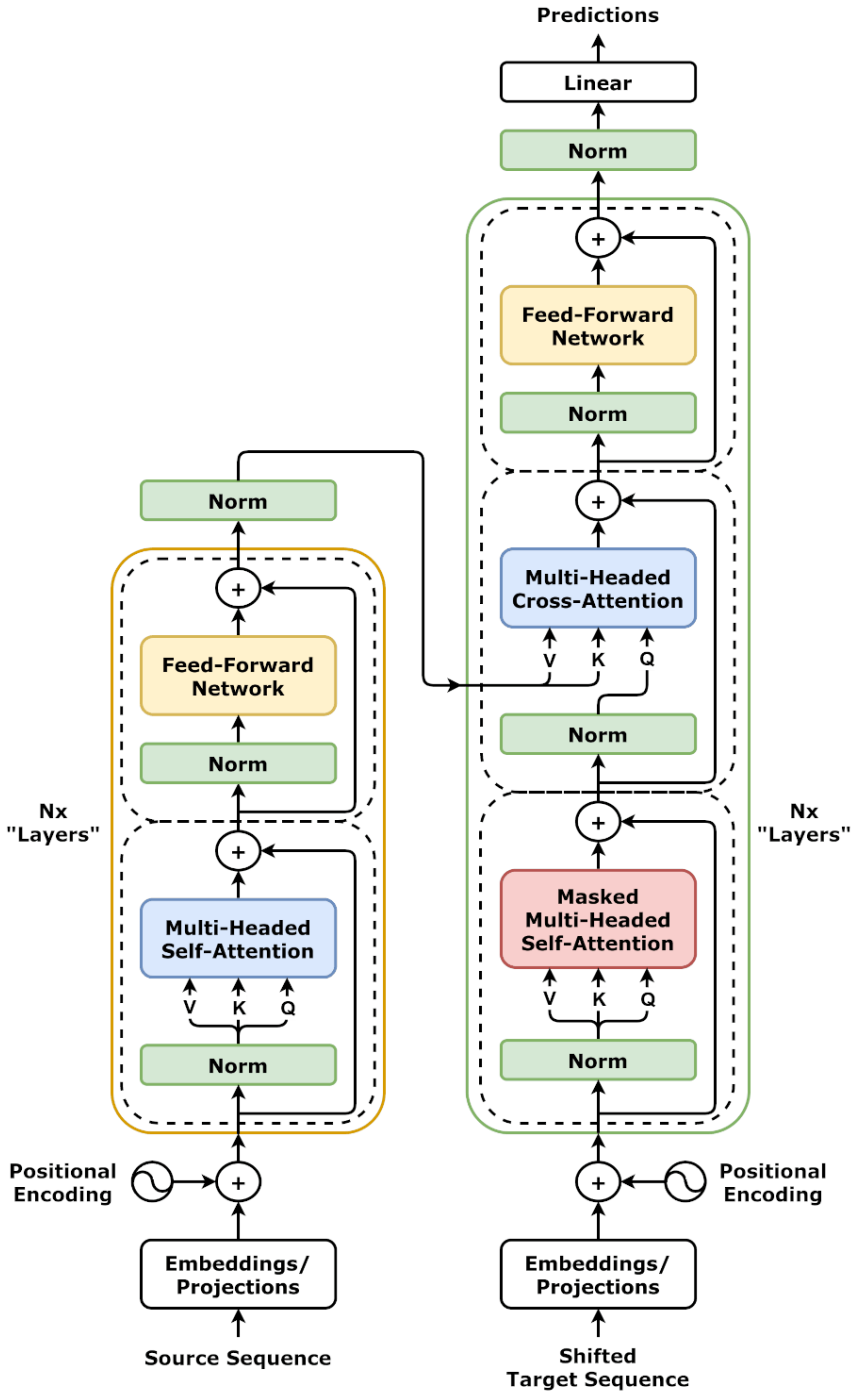

Figure 1.2 - Transformer architecture in detail

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.

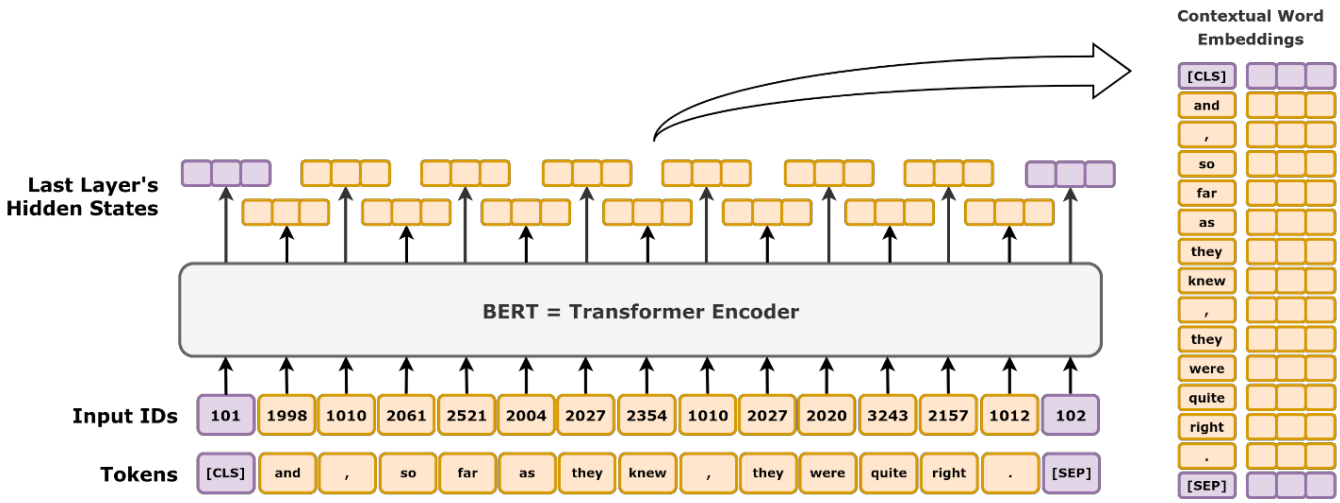

Figure 1.3 - Contextual word embeddings from BERT

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).

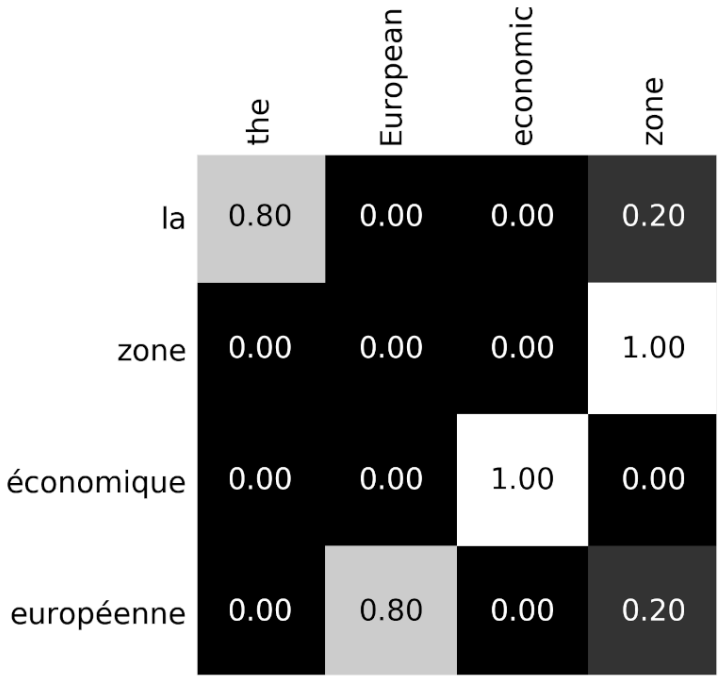

Figure 1.4 - Attention scores

"*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.

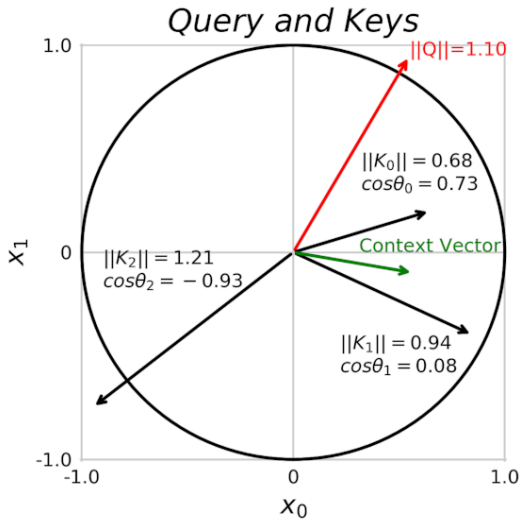

Figure 1.5 - Querying two-dimensional keys

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