**6. Finetuning: Hands on**

Aarushi Kansal[**1**](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#Aff2)

(1)

Melbourne, Australia

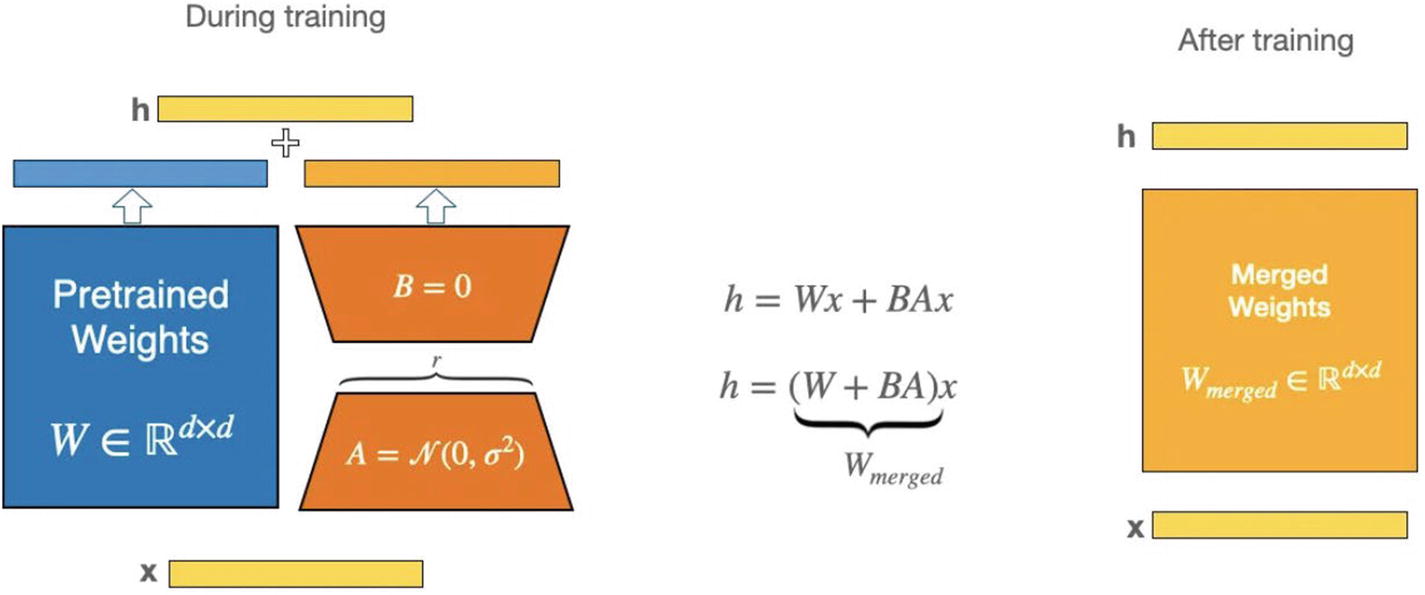
In Chapter [5](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_5_Chapter.xhtml), you learned about fine-tuning and model alignment in a very theoretical manner. It was the foundation to being able to fine-tune your own models. You learned about the whys, whats, and hows of fine-tuning. You learned that fine-tuning can be less resource and time consuming than building and training a model from scratch. The previous chapter talked to you about what happens to the neural network during the fine-tuning process – specifically that most layers are “frozen” and the final few layers are updated to adapt the model to a new task. The focus was on Reinforcement Learning with Human Feedback (RLHF) and Parameter-Efficient Fine-Tuning (PEFT).

In this chapter, you’re going to practice fine-tuning yourself. In particular this chapter will focus on using Llama 2 and PEFT for fine-tuning.

**Refresher**

Let’s quickly go through a little refresher on LoRA before you begin – if you remember everything, feel free to skip this section.

Recall from Chapter [5](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_5_Chapter.xhtml) – model training can be very resource heavy, and PEFT techniques such as LoRA attempt to minimize the amount of GPU and infra needed. Specifically, with LoRA, you can freeze most weights and only update or fine-tune the later few layers or weights needed for your specific needs. Training fewer weights allows you to fine-tune large models on a lower amount of GPUs – often only needing one. In Figure [6-1](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#Fig1), you can see with LoRA you only train A and matrices; the other weights remain frozen. After training, these are merged, leaving you with an adapted model for your specific use case.



***Figure 6-1***

Diagram from a LoRA paper, only A and B are fine-tuned (source: [https://​arxiv.​org/​abs/​2106.​09685](https://arxiv.org/abs/2106.09685))

While LoRA is already a significant improvement – in this chapter, you’re going to use a technique that goes one step further: QLoRA.

The concept of fine-tuning is the same as in LoRA, but QLoRA reduces the size of the model and speeds up inference.

Here’s how QLoRA does this:

1. 1.

**Uses Less Memory**: It changes the model slightly so that it uses less memory. Think of it like compressing a huge video into a smaller file so it’s easier to watch on your phone.

1. 2.

**4-Bit Inference**: Using 4-bit inference enhances speed and efficiency of the model, without degrading quality or performance.

**4-Bit NormalFloat (NF4) Data Type**

* **What It Is**: 4-bit NormalFloat (NF4) is a new type of data format. In typical machine learning models, weights (the parameters that get adjusted during training) are usually stored in a format that takes up a lot of memory. NF4, however, represents these weights in a way that requires much less space.
* **How It Works**: NF4 efficiently compresses the model’s weights without losing important information. It’s especially effective for weights that follow a normal (bell-curve) distribution, which is common in AI models. This is like taking a detailed picture and compressing it into a smaller file size while keeping all the important details intact.
* **Impact**: By using NF4, QLoRA drastically reduces the amount of memory needed to store the model’s weights. This is key in enabling the fine-tuning of massive models on less powerful hardware.

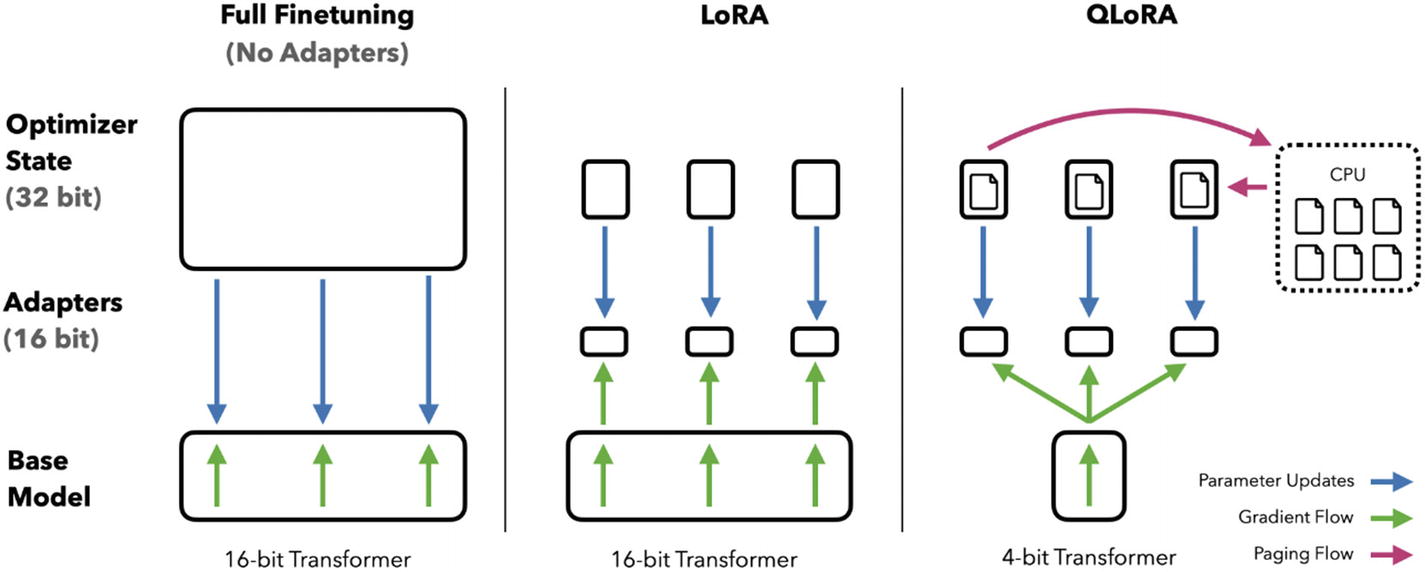
**Double Quantization**

* **What It Is**: Quantization is a process of simplifying the weights in a neural network to reduce their precision. Normally, this is done once, but QLoRA uses a technique called double quantization.
* **How It Works**: Imagine you first simplify a set of numbers, and then you find a way to simplify those simplified numbers even further. That’s what double quantization does – it compresses already compressed data, making it more compact.
* **Impact**: This further reduces the model’s memory footprint, allowing for efficient use of available memory and enabling the fine-tuning of very large models that would otherwise be unmanageable.

**Paged Optimizers**

* **What They Are**: Optimizers in machine learning are algorithms that adjust the weights of the model to reduce errors in predictions. Paged Optimizers are a special kind of optimizer used in QLoRA.
* **How They Work**: These optimizers manage memory more efficiently during the training process. Think of it as having a smart system that only pulls out the tools (weights) you need at the moment and puts them back when they’re not needed, preventing the workbench (memory) from getting cluttered.
* **Impact**: Paged Optimizers help to manage and reduce sudden increases in memory use (called spikes) that typically occur during training. This makes it feasible to train large models on hardware with limited memory.

In Figure [6-2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#Fig2), you can see a comparison of fine-tuning techniques and a visual representation of how QLoRA uses Paged Optimizers to manage memory more efficiently.



***Figure 6-2***

Diagram comparing full fine-tuning, LoRA, and QLoRA (source: [https://​arxiv.​org/​pdf/​2305.​14314.​pdf](https://arxiv.org/pdf/2305.14314.pdf))

**What Is Llama 2?**

In July 2023, Meta released their latest (almost) open source, pre-trained, transformer-based LLM: Llama 2. I say almost because there are some restrictions and requirements to the license for Llama 2. You can check them out on their website: <https://ai.meta.com/llama/>.

It’s notable for being a contender to challenging proprietary LLMs – models that were once considered only for the tech giants, meaning almost anyone can run, host, and fine-tune a large model with similar if not better capabilities. The model comes in varying parameter sizes, from 7 billion up to 70 billion.

In terms of training, according to Meta, Llama 2 has been pre-trained on a wide array of publicly available online data, and they claim to not train on any Meta data. Diversity of the dataset helps the model in effectively understanding and generating human-like text across various topics and styles.

One of the key improvements in Llama 2 is its increased context length, which is double that of its predecessor. This enhancement enables the model to consider more information from the input text, leading to outputs that are more coherent and contextually relevant.

The model also includes a version fine-tuned for dialogue, known as LLaMA-2-Chat, making it particularly useful for applications in conversational AI, such as chatbots and virtual assistants.

And in this chapter, you’re going to learn how to fine-tune your own version of Llama 2. Let’s get started with some coding.

**Fine-Tuning**

**Setup**

1. 1)

**Google Colab Notebook**: I’m going to use an A100, but you can also use a T4 as well for this book.

1. 2)

**Llama 2**: 7B parameter chat model.

1. 3)

**Python 3**

**Llama 2 Model**

You can either request access to the model from Meta here: <https://ai.meta.com/resources/models-and-libraries/llama-downloads/>, or you can use one of the Llama models already on Hugging Face, such as <https://huggingface.co/NousResearch/Llama-2-7b-chat-hf>. It’s the same model, but you don’t have to wait for access. For the purpose of this exercise, I’m going to use the one from Nous Research.

First, go ahead and download the libraries you’ll need as shown in Listing [6-1](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC1).

!pip install -q accelerate==0.21.0 peft==0.4.0 bitsandbytes==0.40.2 transformers==4.31.0 trl==0.4.7

***Listing 6-1***

Installing all required libraries and versions

Next, you’ll import all the modules and functions, which you can see in Listing [6-2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC2).

import os

import torch

from datasets import load\_dataset

from transformers import (

   AutoModelForCausalLM,

   AutoTokenizer,

   BitsAndBytesConfig,

   HfArgumentParser,

   TrainingArguments,

   pipeline,

   logging,

)

from peft import LoraConfig, PeftModel

from trl import SFTTrainer

***Listing 6-2***

Module imports for fine-tuning

That’s your general setup and now to the more fun and configurable parts. First, you’re going to decide on and load a few things:

1. 1)

The base model you want to fine-tune

1. 2)

The dataset you want to fine-tune with

1. 3)

The name of your new fine-tuned model

All of which you can see in Listing [6-3](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC3).

# Base model to finetune - using NousResearch so you don't have to wait for access req

model\_name = "NousResearch/llama-2-7b-chat-hf"

# Dataset to use - find more on HuggingFace

dataset\_name = "mlabonne/guanaco-llama2-1k"

# Newly fine-tuned model name

new\_model = "llama-2-7b-gen-ai-book"

***Listing 6-3***

Model and dataset names

Notice for the dataset, I’ve chosen an existing one called mlabonne/guanaco-llama2-1k. Let’s talk a little about datasets.

Firstly, you can find a range of different datasets on both Hugging Face and Kaggle – so really, take your pick. The reason I’ve chosen this one for this book is because its small (only 1k) and also already formatted for Llama. The other dataset I like is <https://huggingface.co/datasets/Photolens/oasst1-langchain-llama-2-formatted>, also formatted perfectly – but a lot bigger, so choose this if you have a lot of time for fine-tuning.

**Formatting**

From Meta’s paper on Llama, the required template for prompting is as shown in Listing [6-4](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC4).

<s>[INST] <<SYS>>

{{ system\_prompt }}

<</SYS>>

{{ user\_message }} [/INST]

***Listing 6-4***

Llama 2 prompting template

This template follows the training dataset, and it’s the format you’re going to need your own dataset in as well for fine-tuning. So you can either use one of the ones already in the right format or choose your own and format it.

The content in between <<SYS>> <</SYS>> is the model’s context. For example, it could be some kind of role the system is playing.

Also, one note on prompt template, since you’re fine-tuning, you could in theory also update the actual expected prompt template, so your new model would actually be fine-tuned to understand a different model. You won’t do that in this book – but it could be an exercise for you to try out yourself.

So now you can go ahead and load the dataset, as shown in Listing [6-5](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC5).

dataset = load\_dataset(dataset\_name, split="train")

***Listing 6-5***

Load the dataset of your choice, name defined earlier

Now you’re going to do the quantization configuration using the BitsAndBytesConfig – remember, quantizing basically means converting the weights in a way that reduces the memory used by the model, and in QLoRA, this is done twice, all of which you can see in Listing [6-6](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC6).

bnb\_config = BitsAndBytesConfig(

   load\_in\_4bit=True,

   bnb\_4bit\_quant\_type="nf4",

   bnb\_4bit\_compute\_dtype=compute\_dtype,

   bnb\_4bit\_use\_double\_quant=True,

)

***Listing 6-6***

4-bit double quantization by BitsAndBytesConfig

This is just setting up the configuration; you still have to actually load a quantized model. And to do that, you’re going to use AutoModelForCausalLM from the same Hugging Face transformer library as shown in Listing [6-7](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC7). Specifically you tell it the model (which you defined earlier) and the BitsAndBytesConfig configuration that you just set up.

model = AutoModelForCausalLM.from\_pretrained(

   model\_name,

   quantization\_config=bnb\_config,

   device\_map={"": 0}

)

model.config.use\_cache = False

model.config.pretraining\_tp = 1

***Listing 6-7***

4-bit double quantization by BitsAndBytesConfig

When you run this code snippet, the library infers the model architecture based on the path you provide it (the path being where it lives on Hugging Face). It then loads the model, with the quantization applied – meaning the model’s weights are converted from their original precision (typically 32-bit floating point) to the 4-bit format as defined.

So by now you’ve loaded up a quantized model – meaning it’s memory footprint is significantly smaller. That’s “Q” in QLoRA. You still have to do the actual LoRA setup.

Listing [6-8](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC8) shows you how to set up a configuration for LoRA. Let’s dive into each parameter in the LoraConfig:

1. 1.

lora\_alpha=16: This parameter specifies the scaling factor (α) for the LoRA layers. In the context of LoRA, α is a hyperparameter that controls the scaling of the low-rank updates applied to the model’s weights. A higher value of α typically leads to more significant updates during fine-tuning.

1. 2.

lora\_dropout=0.1: This sets the dropout rate for the LoRA layers. Dropout is a regularization technique used to prevent overfitting in neural networks. A dropout rate of 0.1 means that during the training process, each parameter in the LoRA layers has a 10% chance of being temporarily “dropped,” that is, set to zero, which helps in making the model less sensitive to specific features and promotes generalization.

1. 3.

r=64: This parameter defines the rank of the low-rank matrices used in LoRA. The rank (r) here is a crucial part of LoRA’s approach to reducing the number of trainable parameters. By using low-rank matrices (matrices with reduced rank), LoRA allows for a more memory-efficient way of fine-tuning large models. A rank of 64 means that the low-rank matrices will have 64 columns (or rows, depending on the implementation), which is significantly smaller than the size of the original weight matrices in large language models.

1. 4.

bias=”none”: This indicates that no bias term is added in the LoRA layers. In neural networks, a bias term is often added to the output of each neuron to help the model fit the data better. By setting it to “none,” this configuration opts not to use such bias terms in the LoRA layers.

1. 5.

task\_type=”CAUSAL\_LM”: This specifies the type of task the model is being fine-tuned for. In this case, “CAUSAL\_LM” indicates a causal language modeling task, where the model generates text based on a given context, predicting each next token based on the previous ones (as opposed to, for example, masked language modeling).

peft\_config = LoraConfig(

   lora\_alpha=16,

   lora\_dropout=0.1,

   r=64,

   bias="none",

   task\_type="CAUSAL\_LM",

)

***Listing 6-8***

LoRA config

These are configurable values, and you should tweak them and go through a bit of a trial-and-error process for your own use cases.

Next is configuring the actual training or fine-tuning parameters, shown in Listing [6-9](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC9).

There are quite a few hyperparameters you can deal with here. Let’s dive into some of them.

1. 1.

num\_train\_epochs=1: The number of training epochs, that is, how many times the entire training dataset will be passed through the model. Here, it’s set to 1, meaning the dataset will be used once for training.

1. 2.

per\_device\_train\_batch\_size=4: The batch size per device during training. Batch size is the number of training examples utilized in one iteration. A size of 4 means that the model will process four examples at a time on each device (like a GPU).

1. 3.

gradient\_accumulation\_steps=1: This sets the number of steps to accumulate gradients before performing a backward/update pass. A value of 1 means the model will update weights after every forward-backward pass.

1. 4.

optim=”paged\_adamw\_32bit”: Specifies the optimizer to use for training. “paged\_adamw\_32bit” refers to a variant of the AdamW optimizer with 32-bit precision, with modifications for efficient memory management (“paged”).

1. 5.

save\_steps=25: The model will save a checkpoint every 25 training steps.

1. 6.

logging\_steps=25: Logging metrics will happen every 25 steps of training.

1. 7.

learning\_rate=2e-4: The learning rate for the optimizer. This is a crucial hyperparameter that affects how much the model weights are updated during training.

1. 8.

weight\_decay=0.001: This sets the weight decay rate, a regularization technique to prevent overfitting by penalizing large weights.

1. 9.

fp16=False, bf16=False: These parameters indicate that neither 16-bit floating-point (FP16) or bfloat16 precision is used during training, which can be methods for reducing memory usage.

1. 10.

max\_grad\_norm=0.3: This is for gradient clipping to avoid exploding gradients. Gradients will be clipped if their norm exceeds 0.3.

1. 11.

max\_steps=-1: This implies that training will not be bounded by a maximum number of steps (it will rely on the number of epochs instead).

1. 12.

warmup\_ratio=0.03: This defines the warmup phase of training, where the learning rate gradually ramps up to the full specified rate. A ratio of 0.03 means that 3% of the total training steps will be used for warmup.

1. 13.

group\_by\_length=True: This indicates that training examples will be grouped by their lengths for more efficient batching.

1. 14.

lr\_scheduler\_type=‘constant’: The learning rate scheduler type. Here, ‘constant’ means the learning rate does not change during training.

training\_arguments = TrainingArguments(

   output\_dir=output\_dir,

   num\_train\_epochs=1,

   per\_device\_train\_batch\_size=4,

   gradient\_accumulation\_steps=1,

   optim="paged\_adamw\_32bit",

   save\_steps=25,

   logging\_steps=25,

   learning\_rate=2e-4,

   weight\_decay=0.001,

   fp16=False,

   bf16=False,

   max\_grad\_norm=0.3,

   max\_steps=-1,

   warmup\_ratio=0.03,

   group\_by\_length=True,

   lr\_scheduler\_type='constant',

   report\_to="tensorboard"

)

***Listing 6-9***

Training config

Finally, the actual fine-tuning happens with an SFTTrainer from Hugging Face. You installed the TRL library, which provides an interface for you to do supervised fine-tuning, by just providing your model, dataset, LoRA config, and training params (among a few others) and then running the training by calling .train(), all of which you can see in Listing [6-10](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC10).

trainer = SFTTrainer(

   model=model,

   train\_dataset=dataset,

   peft\_config=peft\_config,

   dataset\_text\_field="text",

   max\_seq\_length=None,

   tokenizer=tokenizer,

   args=training\_arguments,

   packing=False,

)

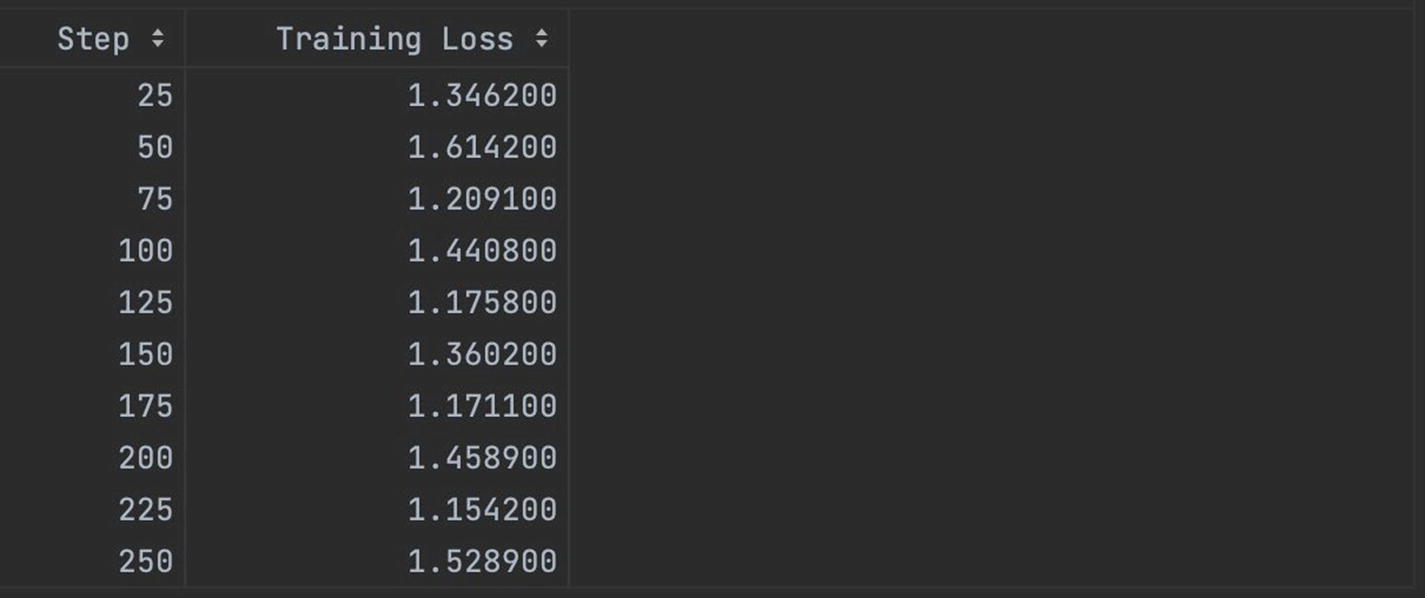
# training

trainer.train()

***Listing 6-10***

Supervised fine-tuning

Once you start the training, it’ll complete 1 epoch, and depending on the colab settings you’re using, timing might range from 0.5 to 1.5 hrs. You’ll see the steps and training loss, as shown in Figure [6-3](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#Fig3).



***Figure 6-3***

Example of fine-tuning running

Now that your model is fine-tuned, you need to save it, as shown in Listing [6-11](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC11). Once you save it, you’ll see the new model and related files in the path you specified earlier.

trainer.model.save\_pretrained(new\_model)

***Listing 6-11***

Saving your new model

From here, you can immediately run inference as shown in Listing [6-12](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_6_Chapter.xhtml#PC12). Notice the template is the same as you learned about earlier in this chapter. If you fine-tuned your model to efficiently work with another prompt template, then you can update it here too.

# Run inference immediately after training on model

prompt = "YOUR QUERY HERE"

pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer, max\_length=800)

result = pipe(f"<s>[INST] {prompt} [/INST]")

print(result[0]['generated\_text'])

***Listing 6-12***

Example of running inference on fine-tuned Llama 2

Also notice here you’re actually calling the base model name – that’s because you’re running it in the same script, meaning the model object holds the updated weights. And you can just call this object without reloading the new model.

If, however, you run it in a new session, you will need to reload from the new\_model directory to make sure you are using the model with the updated weights.

**Summary**

In this chapter, you learned how to fine-tune an open source model (Llama 2) using just one GPU, all thanks to a technique called QLoRA. QLoRA incorporates two aspects: quantization and LoRA. The combination of the two ensures the model consumes less memory, fine-tuning is faster (while remaining accurate), and inference is faster on a smaller model.