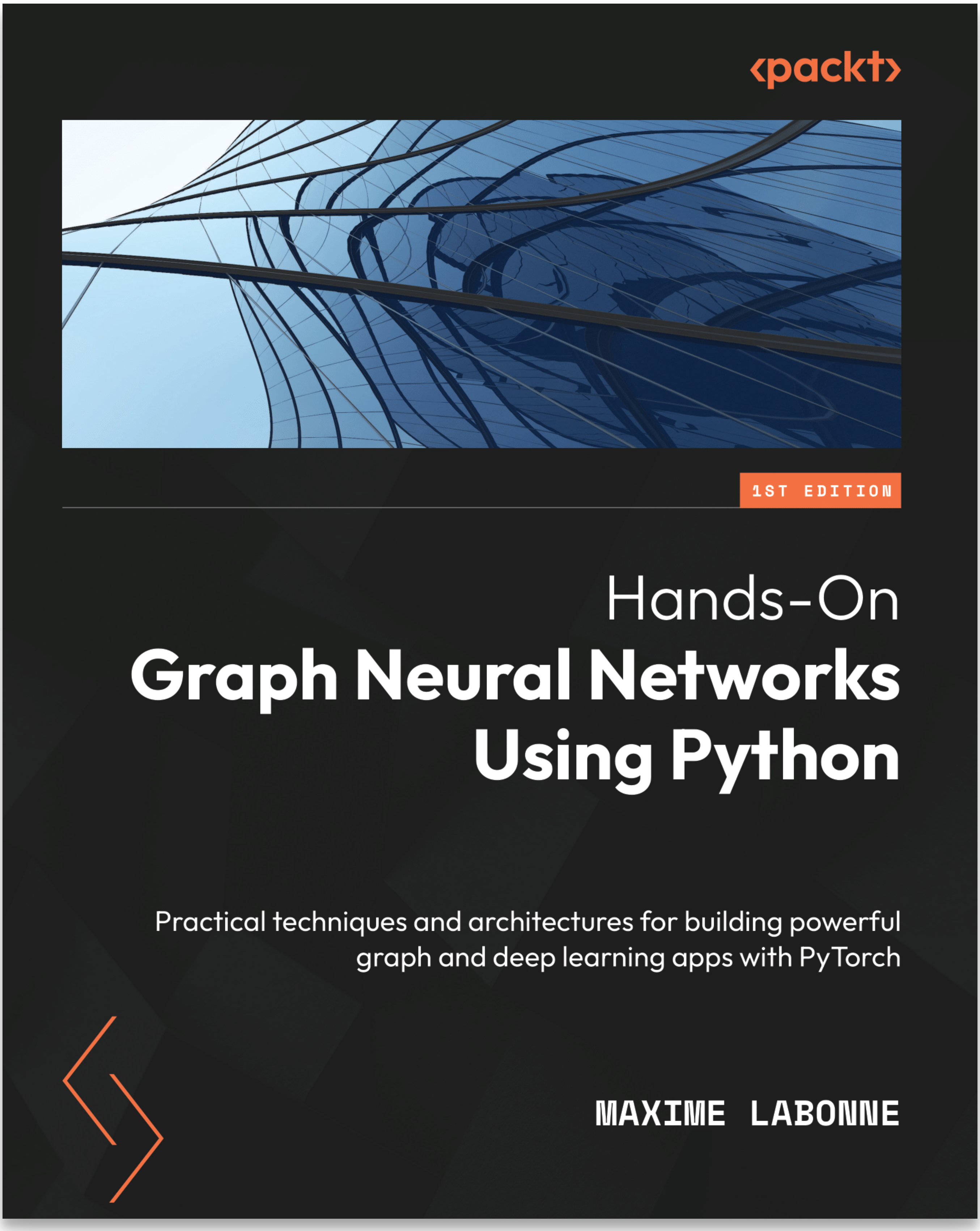
ML Blog

Fine-Tune Your Own Llama 2 Model in a Colab Notebook

A practical introduction to LLM fine-tuning

LARGE LANGUAGE MODELS

Find more a lot more architectures and applications using graph neural networks in my book, **Hands-On Graph Neural Networks** 👇

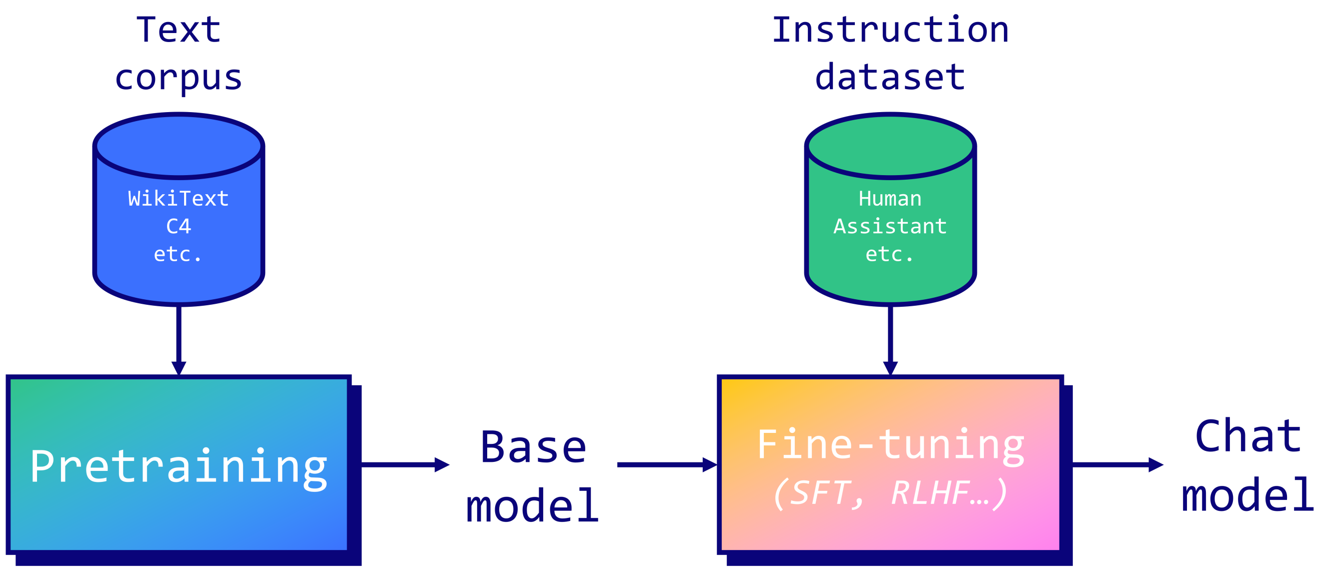

‹packt›
1ST EDITION
Hands-On
Graph Neural Networks
Using Python
Practical techniques and architectures for building powerful graph and deep learning apps with PyTorch
MAXIME LABONNE

With the release of LLaMA v1, we saw a Cambrian explosion of fine-tuned models, including Alpaca, Vicuna, WizardLM, among others. This trend encouraged different businesses to launch their own base models with licenses suitable for commercial use, such as OpenLLaMA, Falcon, XGen, etc. The release of Llama 2 now combines the best elements from both sides: it offers a **highly efficient base model along with a more permissive license**.

During the first half of 2023, the software landscape was significantly shaped by the **widespread use of APIs** (like OpenAI API) to create infrastructures based on Large Language Models (LLMs). Libraries

such as LangChain and LlamaIndex played a critical role in this trend. Moving into the latter half of the year, the process of **fine-tuning (or instruction tuning) these models is set to become a standard procedure** in the LLMOps workflow. This trend is driven by various factors: the potential for cost savings, the ability to process confidential data, and even the potential to develop models that exceed the performance of prominent models like ChatGPT and GPT-4 in certain specific tasks.
In this article, we will see why instruction tuning works and how to implement it in a Google Colab notebook to create your own Llama 2 model. As usual, the code is available on Colab and GitHub.

🔧 Background on fine-tuning LLMs



LLMs are pretrained on an extensive corpus of text. In the case of Llama 2, we know very little about the composition of the training set, besides its length of 2 trillion tokens. In comparison, BERT (2018) was "only" trained on the BookCorpus (800M words) and English Wikipedia (2,500M words). From experience, this is a **very costly and long process** with a lot of hardware issues. If you want to know more about it, I recommend reading Meta's logbook about the pretraining of the OPT-175B model.

When the pretraining is complete, auto-regressive models like Llama 2 can **predict the next token** in a sequence. However, this does not make them particularly useful assistants since they don't reply to instructions. This is why we employ instruction tuning to align their answers with what humans expect. There are two main fine-tuning techniques:

- **Supervised Fine-Tuning** (SFT): Models are trained on a dataset of instructions and responses. It adjusts the weights in the LLM to minimize the difference between the generated answers and ground-truth responses, acting as labels.
- **Reinforcement Learning from Human Feedback** (RLHF): Models learn by interacting with their environment and receiving feedback. They are trained to maximize a reward signal (using PPO), which is often derived from human evaluations of model outputs.

In general, RLHF is shown to capture **more complex and nuanced** human preferences, but is also more challenging to implement effectively. Indeed, it requires careful design of the reward system and can be sensitive to the quality and consistency of human feedback. A possible alternative in the future is the Direct Preference Optimization (DPO) algorithm, which directly runs preference learning on the SFT model.

In our case, we will perform SFT, but this raises a question: why does fine-tuning work in the first place? As highlighted in the Orca paper, our understanding is that fine-tuning **leverages knowledge learned during the pretraining** process. In other words, fine-tuning will be of little help if the model has never seen the kind of data you're interested in. However, if that's the case, SFT can be extremely performant.

For example, the LIMA paper showed how you could outperform GPT-3 (DaVinci003) by fine-tuning a LLaMA (v1) model with 65 billion parameters on only 1,000 high-quality samples. The **quality of the instruction dataset is essential** to reach this level of performance, which is why a lot of work is focused on this issue (like evol-instruct, Orca, or phi-1). Note that the size of the LLM (65b, not 13b or 7b) is also fundamental to leverage pre-existing knowledge efficiently.

Another important point related to the data quality is the **prompt template**. Prompts are comprised of similar elements: system prompt (optional) to guide the model, user prompt (required) to give the instruction, additional inputs (optional) to take into consideration, and the model's answer (required). In the case of Llama 2, the authors used the following template for the **chat models**:

```
<s>[INST] <<SYS>>
System prompt
<</SYS>>

User prompt [/INST] Model answer </s>
```

There are other templates, like the ones from Alpaca and Vicuna, and their impact is not very clear. In this example, we will reformat our instruction dataset to follow Llama 2's template. For the purpose of this tutorial, I've already done it using the excellent `timdettmers/openassistant-guanaco` dataset. You can find it on Hugging Face under the name `mlabonne/guanaco-llama2-1k`. In the following, we will use a base model instead of a chat model, so this step is optional. Note that you don't need to follow a specific prompt template if you're using the base Llama 2 model instead of the chat version.

🦙 How to fine-tune Llama 2

In this section, we will fine-tune a Llama 2 model with 7 billion parameters on a T4 GPU with high RAM using Google Colab (2.21 credits/hour). Note that a T4 only has 16 GB of VRAM, which is barely enough to **store Llama 2-7b's weights** (7b × 2 bytes = 14 GB in FP16). In addition, we need to consider the overhead due to optimizer states, gradients, and forward activations (see this excellent article for more information). This means that a full fine-tuning is not possible here: we need parameter-efficient fine-tuning (PEFT) techniques like LoRA or QLoRA.

To drastically reduce the VRAM usage, we must **fine-tune the model in 4-bit precision**, which is why we'll use QLoRA here. The good thing is that we can leverage the Hugging Face ecosystem with the `transformers`, `accelerate`, `peft`, `trl`, and `bitsandbytes` libraries. This is what we'll do in the following code, based on Younes Belkada's GitHub Gist. First, we install and load these libraries.

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

```
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
```

Let's talk a bit about the parameters we can tune here. First, we want to load a `llama-2-7b-chat-hf` model and train it on the `mlabonne/guanaco-llama2-1k` (1,000 samples), which will produce our fine-tuned model `llama-2-7b-miniguanaco`. If you're interested in how this dataset was created, you can check this notebook. Feel free to change it: there are many good datasets on the Hugging Face Hub, like `databricks/databricks-dolly-15k`.

QLoRA will use a rank of 64 with a scaling parameter of 16 (see this article for more information about LoRA parameters). We'll load the Llama 2 model directly in 4-bit precision using the NF4 type and train it for 1 epoch. To get more information about the other parameters, check the TrainingArguments, PeftModel, and SFTTrainer documentation.

```
# The model that you want to train from the Hugging Face hub
model_name = "NousResearch/llama-2-7b-chat-hf"

# The instruction dataset to use
dataset_name = "mlabonne/guanaco-llama2-1k"

# Fine-tuned model name
new_model = "llama-2-7b-miniguanaco"

################################################################################
# QLoRA parameters
################################################################################

# LoRA attention dimension
lora_r = 64

# Alpha parameter for LoRA scaling
lora_alpha = 16

# Dropout probability for LoRA layers
lora_dropout = 0.1
```

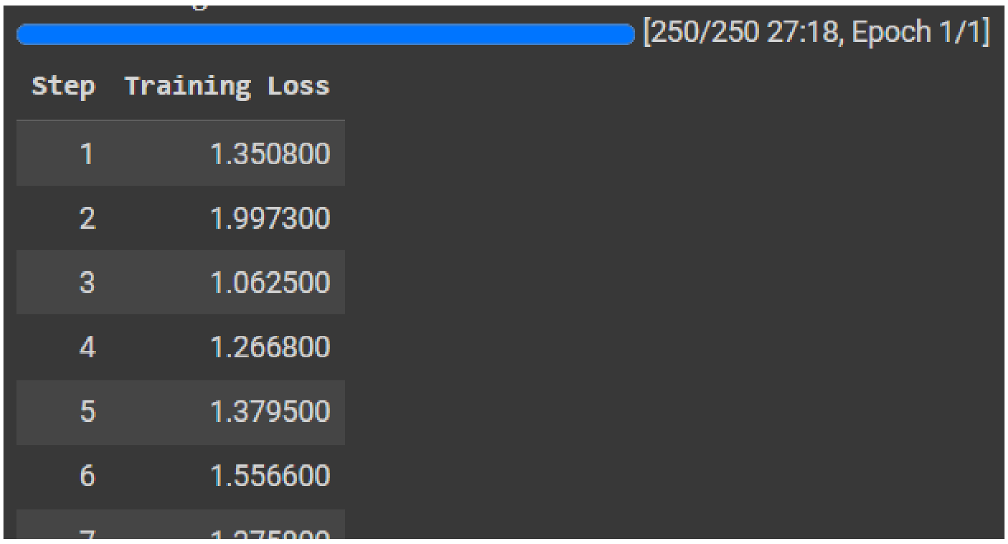
```python
################################################################################
# bitsandbytes parameters
################################################################################

# Activate 4-bit precision base model loading
use_4bit = True

# Compute dtype for 4-bit base models
bnb_4bit_compute_dtype = "float16"

# Quantization type (fp4 or nf4)
bnb_4bit_quant_type = "nf4"

# Activate nested quantization for 4-bit base models (double quantization)
use_nested_quant = False

################################################################################
# TrainingArguments parameters
################################################################################

# Output directory where the model predictions and checkpoints will be stored
output_dir = "./results"

# Number of training epochs
num_train_epochs = 1

# Enable fp16/bf16 training (set bf16 to True with an A100)
fp16 = False
bf16 = False

# Batch size per GPU for training
per_device_train_batch_size = 4

# Batch size per GPU for evaluation
per_device_eval_batch_size = 4

# Number of update steps to accumulate the gradients for
gradient_accumulation_steps = 1

# Enable gradient checkpointing
gradient_checkpointing = True

# Maximum gradient normal (gradient clipping)
max_grad_norm = 0.3

# Initial learning rate (AdamW optimizer)
learning_rate = 2e-4

# Weight decay to apply to all layers except bias/LayerNorm weights
weight_decay = 0.001

# Optimizer to use
optim = "paged_adamw_32bit"
```

```
# Learning rate schedule (constant a bit better than cosine)
lr_scheduler_type = "constant"

# Number of training steps (overrides num_train_epochs)
max_steps = -1

# Ratio of steps for a linear warmup (from 0 to learning rate)
warmup_ratio = 0.03

# Group sequences into batches with same length
# Saves memory and speeds up training considerably
group_by_length = True

# Save checkpoint every X updates steps
save_steps = 25

# Log every X updates steps
logging_steps = 25

################################################################################
# SFT parameters
################################################################################

# Maximum sequence length to use
max_seq_length = None

# Pack multiple short examples in the same input sequence to increase efficiency
packing = False

# Load the entire model on the GPU 0
device_map = {"": 0}
```

We can now load everything and start the fine-tuning process. We're relying on multiple wrappers, so bear with me.

- First of all, we want to **load the dataset** we defined. Here, our dataset is already preprocessed but, usually, this is where you would reformat the prompt, filter out bad text, combine multiple datasets, etc.
- Then, we're configuring `bitsandbytes` for 4-bit quantization.
- Next, we're loading the Llama 2 model in 4-bit precision on a GPU with the corresponding tokenizer.
- Finally, we're loading configurations for QLoRA, regular training parameters, and passing everything to the `SFTTrainer`. The training can finally start!

```
# Load dataset (you can process it here)
dataset = load_dataset(dataset_name, split="train")

# Load tokenizer and model with QLoRA configuration
compute_dtype = getattr(torch, bnb_4bit_compute_dtype)

bnb_config = BitsAndBytesConfig(
    load_in_4bit=use_4bit,
    bnb_4bit_quant_type=bnb_4bit_quant_type,
    bnb_4bit_compute_dtype=compute_dtype,
    bnb_4bit_use_double_quant=use_nested_quant,
)

# Check GPU compatibility with bfloat16
if compute_dtype == torch.float16 and use_4bit:
    major, _ = torch.cuda.get_device_capability()
    if major >= 8:
        print("=" * 80)
        print("Your GPU supports bfloat16: accelerate training with bf16=True")
        print("=" * 80)

# Load base model
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map=device_map
)
model.config.use_cache = False
model.config.pretraining_tp = 1

# Load LLaMA tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right" # Fix weird overflow issue with fp16 training

# Load LoRA configuration
peft_config = LoraConfig(
    lora_alpha=lora_alpha,
    lora_dropout=lora_dropout,
    r=lora_r,
    bias="none",
    task_type="CAUSAL_LM",
)

# Set training parameters
training_arguments = TrainingArguments(
    output_dir=output_dir,
    num_train_epochs=num_train_epochs,
    per_device_train_batch_size=per_device_train_batch_size,
    gradient_accumulation_steps=gradient_accumulation_steps,
    optim=optim,
    save_steps=save_steps,
    logging_steps=logging_steps,
    learning_rate=learning_rate,
    weight_decay=weight_decay,
    fp16=fp16,
    bf16=bf16,
    max_grad_norm=max_grad_norm,
    max_steps=max_steps,
    warmup_ratio=warmup_ratio,
    group_by_length=group_by_length,
    lr_scheduler_type=lr_scheduler_type,
    report_to="tensorboard"
)

# Set supervised fine-tuning parameters
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    peft_config=peft_config,
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    tokenizer=tokenizer,
    args=training_arguments,
    packing=packing,
)

# Train model
trainer.train()

# Save trained model
trainer.model.save_pretrained(new_model)
```

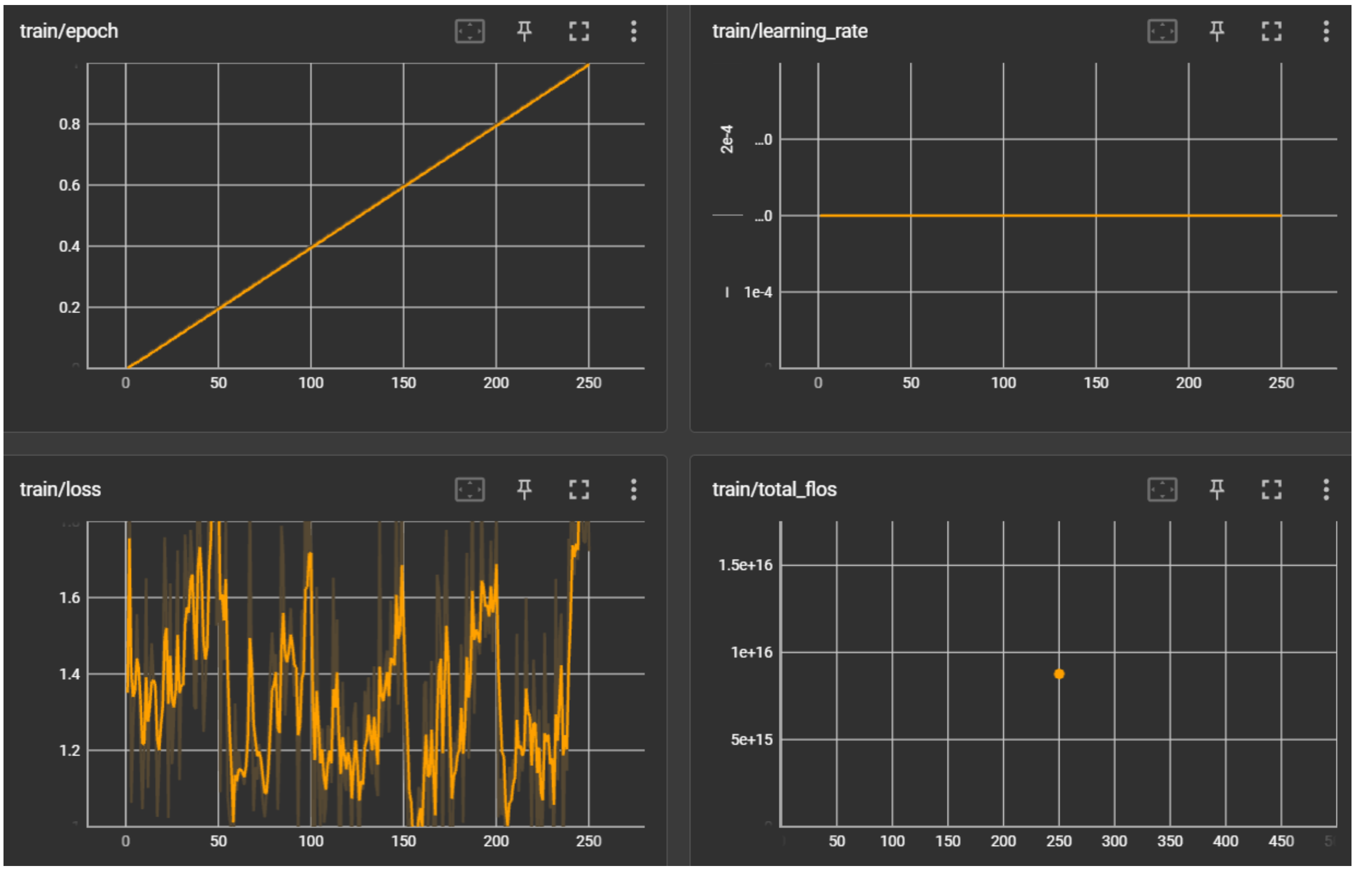


[250/250 27:18, Epoch 1/1]

Step	Training Loss
1	1.350800
2	1.997300
3	1.062500
4	1.266800
5	1.379500
6	1.556600

The training can be very long, depending on the size of your dataset. Here, it took less than an hour on a T4 GPU. We can check the plots on tensorboard, as follows:

```
%load_ext tensorboard
%tensorboard --logdir results/runs
```



Let’s make sure that the model is behaving correctly. It would require a more exhaustive evaluation, but we can use the **text generation pipeline** to ask questions like “What is a large language model?” Note that I’m formatting the input to match Llama 2’s prompt template.

```
# Ignore warnings
logging.set_verbosity(logging.CRITICAL)

# Run text generation pipeline with our next model
prompt = "What is a large language model?"
pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer,
        max_length=200)
result = pipe(f"<s>[INST] {prompt} [/INST]")
print(result[0]['generated_text'])
```

```
/usr/local/lib/python3.10/dist-packages/transformers/generation/utils.py:1270:
UserWarning: You have modified the pretrained model configuration to control
generation. This is a deprecated strategy to control generation and will be removed
soon, in a future version. Please use a generation configuration file (see
https://huggingface.co/docs/transformers/main_classes/text_generation )
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torch/utils/checkpoint.py:31: UserWarning:
None of the inputs have requires_grad=True. Gradients will be None
  warnings.warn("None of the inputs have requires_grad=True. Gradients will be
None")
```

```
<s>[INST] What is a large language model? [/INST] A large language model is a type
of artificial intelligence that is trained on a large dataset of text to generate
human-like language. It is typically trained on a dataset of text that is much
larger than the dataset used for smaller language models. The large dataset allows
the model to learn more complex patterns in language, which can result in more
accurate and natural-sounding language generation.

Large language models are often used for tasks such as text summarization, language
translation, and chatbots. They are also used for more complex tasks such as
writing articles, generating creative content, and even composing music.

Large language models are trained using a technique called deep learning, which
involves using many layers of artificial neural networks to learn complex patterns
in the data. The model is trained on a large dataset of text, and the neural
networks are trained to predict the next word in a sequence of text given
```

The model outputs the following response:

```
A large language model is a type of artificial intelligence that is trained on a
large dataset of text to generate human-like language. It is typically trained on a
dataset of text that is much larger than the dataset used for smaller language
models. The large dataset allows the model to learn more complex patterns in
language, which can result in more accurate and natural-sounding language generation.

Large language models are often used for tasks such as text summarization, language
translation, and chatbots. They are also used for more complex tasks such as writing
articles, generating creative content, and even composing music.

Large language models are trained using a technique called deep learning, which
involves using many layers of artificial neural networks to learn complex patterns in
the data. The model is trained on a large dataset of text, and the neural networks
are trained to predict the next word in a sequence of text given
```

From experience, it is **very coherent** for a model with only 7 billion parameters. You can play with it and ask harder questions from evaluation datasets like BigBench-Hard. Guanaco is an excellent dataset that has produced high-quality models in the past. You can train a Llama 2 model on the entire dataset using `mlabonne/guanaco-llama2`.

How can we store our new `llama-2-7b-miniguanaco` model now? We need to merge the weights from LoRA with the base model. Unfortunately, as far as I know, there is no straightforward way to do it: we need to reload the base model in FP16 precision and use the `peft` library to merge everything. Alas, it also creates a problem with the VRAM (despite emptying it), so I recommend **restarting the notebook**, re-executing the three first cells, and then executing the next one. Please contact me if you know a fix!

```python
# Reload model in FP16 and merge it with LoRA weights
base_model = AutoModelForCausalLM.from_pretrained(
    model_name,
    low_cpu_mem_usage=True,
    return_dict=True,
    torch_dtype=torch.float16,
    device_map=device_map,
)
model = PeftModel.from_pretrained(base_model, new_model)
model = model.merge_and_unload()

# Reload tokenizer to save it
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"
```

Our weights are merged and we reloaded the tokenizer. We can now push everything to the Hugging Face Hub to save our model.

```
!huggingface-cli login

model.push_to_hub(new_model, use_temp_dir=False)
tokenizer.push_to_hub(new_model, use_temp_dir=False)
```

```
CommitInfo(commit_url='https://huggingface.co/mlabonne/llama-2-7b-
guanaco/commit/0f5ed9581b805b659aec68484820edb5e3e6c3f5', commit_message='Upload
tokenizer', commit_description='', oid='0f5ed9581b805b659aec68484820edb5e3e6c3f5',
pr_url=None, pr_revision=None, pr_num=None)
```

You can now use this model for inference by loading it like any other Llama 2 model from the Hub. It is also possible to reload it for more fine-tuning – perhaps with another dataset?

If you're serious about fine-tuning models, **using a script** instead of a notebook is recommended. You can easily rent GPUs on Lambda Labs, Runpod, Vast.ai, for less than 0.3$/h. Once you're connected, you can install libraries, import your script, log in to Hugging Face and other tools (like Weights & Biases for logging your experiments), and start your fine-tuning.

The `trl` script is currently very limited, so I made my own based on the previous notebook. You can **find it here on GitHub Gist**. If you're looking for a comprehensive solution, check out Axolotl from the OpenAccess AI Collective, which natively handles multiple datasets, Deepspeed, Flash Attention, etc.

Conclusion

In this article, we saw how to fine-tune a Llama 2 7b model using a Colab notebook. We introduced some necessary background on LLM training and fine-tuning, as well as important considerations related to instruction datasets. In the second section, we **successfully fine-tuned the Llama 2 model** with its native prompt template and custom parameters.

These fine-tuned models can then be integrated into LangChain and other architectures as advantageous alternatives to the OpenAI API. Remember, in this new paradigm, instruction datasets are the new gold, and the quality of your model heavily depends on the data on which it's been fine-tuned. So, good luck with building high-quality datasets!

If you’re interested in more content about LLMs, follow me on Twitter @maximelabonne.

References

- Hugo Touvron, Thomas Scialom, et al. (2023). Llama 2: Open Foundation and Fine-Tuned Chat Models.
- Philipp Schmid, Omar Sanseviero, Pedro Cuenca, & Lewis Tunstall. Llama 2 is here - get it on Hugging Face. https://huggingface.co/blog/llama2
- Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, & Tatsunori B. Hashimoto. (2023). Stanford Alpaca: An Instruction-following LLaMA model.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, & Kristina Toutanova. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.
- Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, & Luke Zettlemoyer. (2023). QLoRA: Efficient Finetuning of Quantized LLMs.