



Fine-Tuning LLaMA 2 Models using a single GPU, QLoRA and AI Notebooks

By Mathieu Busquet / 2023-07-21 / AI, AI Notebooks, Artificial Intelligence, Deep learning, Fine-tuning, GPU, LLaMa 2, Machine learning, PyTorch, QLoRA

In this tutorial, we will walk you through the process of fine-tuning [LLaMA 2](#) models, providing step-by-step instructions.



Fine-tuning **LLaMA 2** models
using a single GPU and QLoRA



All the code related to this article is available in our dedicated [GitHub repository](#). You can reproduce all the experiments with [OVHcloud AI Notebooks](#).

Introduction

On July 18, 2023, [Meta](#) released [LLaMA 2](#), the latest version of their **Large Language Model** (LLM).

Trained between January 2023 and July 2023 on 2 trillion tokens, these new models outperforms other LLMs on many benchmarks, including reasoning, coding, proficiency, and knowledge tests. This release comes in different flavors, with parameter sizes of **7B**, **13B**, and a mind-blowing **70B**. Models are intended for free for both commercial and research use in English.

To suit every text generation needed and fine-tune these models, we will use [QLoRA \(Efficient Finetuning of Quantized LLMs\)](#), a highly efficient fine-tuning technique that involves quantizing a pretrained LLM to just 4 bits and adding small “Low-Rank Adapters”. This unique approach allows for fine-tuning LLMs **using just a single GPU!** This technique is supported by the [PEFT library](#).

To fine-tune our model, we will create a [OVHcloud AI Notebooks](#) with only 1 GPU.

Mandatory requirements

To successfully fine-tune LLaMA 2 models, you will need the following:

- Fill Meta's form to [request access to the next version of Llama](#). Indeed, the use of Llama 2 is governed by the Meta license, that you must accept in order to download the model weights and tokenizer.
- Have a [Hugging Face](#) account (with the same email address you entered in Meta's form).
- Have a [Hugging Face token](#).
- Visit the page of one of the LLaMA 2 available models (version [7B](#), [13B](#) or [70B](#)), and accept Hugging Face's license terms and acceptable use policy.
- Log in to the Hugging Face model Hub from your notebook's terminal by running the `huggingface-cli login` command, and enter your token. You will not need to add your token as git credential.
- Powerful Computing Resources: Fine-tuning the Llama 2 model requires substantial computational power. Ensure you are running code on GPU(s) when using [AI Notebooks](#) or [AI Training](#).

Set up your Python environment

Create the following `requirements.txt` file:

```
torch
accelerate @ git+https://github.com/huggingface/accelerate.git
bitsandbytes
datasets==2.13.1
transformers @ git+https://github.com/huggingface/transformers.git
peft @ git+https://github.com/huggingface/peft.git
trl @ git+https://github.com/lvwerra/trl.git
scipy
```

Then install and import the installed libraries:

```
pip install -r requirements.txt
```

```
import argparse
import bitsandbytes as bnb
from datasets import load_dataset
from functools import partial
import os
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training, AutoPeftModelF
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, set_seed, Trainer, TrainingArg
    DataCollatorForLanguageModeling, Trainer, TrainingArguments
from datasets import load_dataset
```

Download LLaMA 2 model

As mentioned before, LLaMA 2 models come in different flavors which are 7B, 13B, and 70B. Your choice can be influenced by your computational resources. Indeed, larger models require more resources, memory,

processing power, and training time.

To download the model you have been granted access to, **make sure you are logged in to the Hugging Face model hub**. As mentioned in the requirements step, you need to use the `huggingface-cli login` command.

The following function will help us to download the model and its tokenizer. It requires a bitsandbytes configuration that we will define later.

```
def load_model(model_name, bnb_config):
    n_gpus = torch.cuda.device_count()
    max_memory = f'{40960}MB'

    model = AutoModelForCausalLM.from_pretrained(
        model_name,
        quantization_config=bnb_config,
        device_map="auto", # dispatch efficiently the model on the available ressources
        max_memory = {i: max_memory for i in range(n_gpus)},
    )
    tokenizer = AutoTokenizer.from_pretrained(model_name, use_auth_token=True)

    # Needed for LLaMA tokenizer
    tokenizer.pad_token = tokenizer.eos_token

    return model, tokenizer
```

Download a Dataset

There are many datasets that can help you fine-tune your model. You can even use your own dataset!

In this tutorial, we are going to download and use the [Databricks Dolly 15k dataset](#), which contains **15,000 prompt/response pairs**. It was crafted by over 5,000 Databricks employees during March and April of 2023.

This dataset is designed specifically for fine-tuning large language models. Released under the [CC BY-SA 3.0 license](#), it can be used, modified, and extended by any individual or company, even for commercial applications. So it's a perfect fit for our use case!

However, like most datasets, this one has **its limitations**. Indeed, pay attention to the following points:

- It consists of content collected from the public internet, which means it may contain objectionable, incorrect or biased content and typo errors, which could influence the behavior of models fine-tuned using this dataset.
- Since the dataset has been created for Databricks by their own employees, it's worth noting that the dataset reflects the interests and semantic choices of Databricks employees, which may not be representative of the global population at large.
- We only have access to the **train** split of the dataset, which is its largest subset.

```
# Load the databricks dataset from Hugging Face
from datasets import load_dataset
```

```
dataset = load_dataset("databricks/databricks-dolly-15k", split="train")
```

Explore dataset

Once the dataset is downloaded, we can take a look at it to understand what it contains:

```
print(f'Number of prompts: {len(dataset)}')
print(f'Column names are: {dataset.column_names}')

*** OUTPUT ***
Number of prompts: 15011
Column Names are: ['instruction', 'context', 'response', 'category']
```

As we can see, each sample is a dictionary that contains:

- **An instruction:** What could be entered by the user, such as a question
- **A context:** Help to interpret the sample
- **A response:** Answer to the instruction
- **A category:** Classify the sample between Open Q&A, Closed Q&A, Extract information from Wikipedia, Summarize information from Wikipedia, Brainstorming, Classification, Creative writing

Pre-processing dataset

Instruction fine-tuning is a common technique used to fine-tune a base LLM for a specific downstream use-case.

It will help us to format our prompts as follows:

```
Below is an instruction that describes a task. Write a response that appropriately
completes the request.

### Instruction:
Sea or Mountain

### Response:
I believe Mountain are more attractive but Ocean has it's own beauty and this tropical
weather definitely turn you on! SO 50% 50%

### End
```

To delimit each prompt part by hashtags, we can use the following function:

```
def create_prompt_formats(sample):
    """
    Format various fields of the sample ('instruction', 'context', 'response')
    Then concatenate them using two newline characters
    :param sample: Sample dictionary
    """
```

```

INTRO_BLURB = "Below is an instruction that describes a task. Write a response that appr
INSTRUCTION_KEY = "### Instruction:"
INPUT_KEY = "Input:"
RESPONSE_KEY = "### Response:"
END_KEY = "### End"

blurb = f"{INTRO_BLURB}"
instruction = f"{INSTRUCTION_KEY}\n{sample['instruction']}"
input_context = f"{INPUT_KEY}\n{sample['context']}" if sample["context"] else None
response = f"{RESPONSE_KEY}\n{sample['response']}"
end = f"{END_KEY}"

parts = [part for part in [blurb, instruction, input_context, response, end] if part]

formatted_prompt = "\n\n".join(parts)

sample["text"] = formatted_prompt

return sample

```

Now, we will use our **model tokenizer to process these prompts into tokenized ones**.

The goal is to create input sequences of uniform length (which are suitable for fine-tuning the language model because it maximizes efficiency and minimize computational overhead), that must not exceed the model's maximum token limit.

```

# SOURCE https://github.com/databrickslabs/dolly/blob/master/training/trainer.py
def get_max_length(model):
    conf = model.config
    max_length = None
    for length_setting in ["n_positions", "max_position_embeddings", "seq_length"]:
        max_length = getattr(model.config, length_setting, None)
        if max_length:
            print(f"Found max length: {max_length}")
            break
    if not max_length:
        max_length = 1024
        print(f"Using default max length: {max_length}")
    return max_length

def preprocess_batch(batch, tokenizer, max_length):
    """
    Tokenizing a batch
    """
    return tokenizer(
        batch["text"],
        max_length=max_length,
        truncation=True,
    )

# SOURCE https://github.com/databrickslabs/dolly/blob/master/training/trainer.py
def preprocess_dataset(tokenizer: AutoTokenizer, max_length: int, seed, dataset: str):

```

```

"""Format & tokenize it so it is ready for training
:param tokenizer (AutoTokenizer): Model Tokenizer
:param max_length (int): Maximum number of tokens to emit from tokenizer
"""

# Add prompt to each sample
print("Preprocessing dataset...")
dataset = dataset.map(create_prompt_formats)#, batched=True)

# Apply preprocessing to each batch of the dataset & and remove 'instruction', 'context'
_preprocessing_function = partial(preprocess_batch, max_length=max_length, tokenizer=tokenizer)
dataset = dataset.map(
    _preprocessing_function,
    batched=True,
    remove_columns=["instruction", "context", "response", "text", "category"],
)

# Filter out samples that have input_ids exceeding max_length
dataset = dataset.filter(lambda sample: len(sample["input_ids"]) < max_length)

# Shuffle dataset
dataset = dataset.shuffle(seed=seed)

return dataset

```

With these functions, our dataset will be ready for fine-tuning !

Create a bitsandbytes configuration

This will allow us to load our LLM in 4 bits. This way, we can divide the used memory by 4 and import the model on smaller devices. We choose to apply bfloat16 compute data type and nested quantization for memory-saving purposes.

```

def create_bnb_config():
    bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=torch.bfloat16,
    )

    return bnb_config

```

To leverage the LoRa method, we need to wrap the model as a PeftModel.

To do this, we need to implement a [LoRa configuration](#):

```

def create_peft_config(modules):
    """
    Create Parameter-Efficient Fine-Tuning config for your model
    :param modules: Names of the modules to apply Lora to
    """

```

```

config = LoraConfig(
    r=16, # dimension of the updated matrices
    lora_alpha=64, # parameter for scaling
    target_modules=modules,
    lora_dropout=0.1, # dropout probability for layers
    bias="none",
    task_type="CAUSAL_LM",
)

return config

```

Previous function needs the **target modules** to update the necessary matrices. The following function will get them for our model:

```

# SOURCE https://github.com/artidoro/qlora/blob/main/qlora.py

def find_all_linear_names(model):
    cls = bnb.nn.Linear4bit #if args.bits == 4 else (bnb.nn.Linear8bitLt if args.bits == 8 else bnb.nn.Linear)
    lora_module_names = set()
    for name, module in model.named_modules():
        if isinstance(module, cls):
            names = name.split('.')
            lora_module_names.add(names[0] if len(names) == 1 else names[-1])

    if 'lm_head' in lora_module_names: # needed for 16-bit
        lora_module_names.remove('lm_head')
    return list(lora_module_names)

```

Once everything is set up and the base model is prepared, we can use the *print_trainable_parameters()* helper function to see how many trainable parameters are in the model.

```

def print_trainable_parameters(model, use_4bit=False):
    """
    Prints the number of trainable parameters in the model.
    """
    trainable_params = 0
    all_param = 0
    for _, param in model.named_parameters():
        num_params = param.numel()
        # if using DS Zero 3 and the weights are initialized empty
        if num_params == 0 and hasattr(param, "ds_numel"):
            num_params = param.ds_numel

        all_param += num_params
        if param.requires_grad:
            trainable_params += num_params
    if use_4bit:
        trainable_params /= 2
    print(
        f"all params: {all_param:,d} || trainable params: {trainable_params:,d} || trainable"
    )

```

We expect the LoRa model to have fewer trainable parameters compared to the original one, since we want to perform fine-tuning.

Train

Now that everything is ready, we can pre-process our dataset and load our model using the set configurations:

```
# Load model from HF with user's token and with bitsandbytes config

model_name = "meta-llama/Llama-2-7b-hf"

bnb_config = create_bnb_config()

model, tokenizer = load_model(model_name, bnb_config)
```

```
## Preprocess dataset

max_length = get_max_length(model)

dataset = preprocess_dataset(tokenizer, max_length, seed, dataset)
```

Then, we can run our fine-tuning process:

```
def train(model, tokenizer, dataset, output_dir):
    # Apply preprocessing to the model to prepare it by
    # 1 - Enabling gradient checkpointing to reduce memory usage during fine-tuning
    model.gradient_checkpointing_enable()

    # 2 - Using the prepare_model_for_kbit_training method from PEFT
    model = prepare_model_for_kbit_training(model)

    # Get lora module names
    modules = find_all_linear_names(model)

    # Create PEFT config for these modules and wrap the model to PEFT
    peft_config = create_peft_config(modules)
    model = get_peft_model(model, peft_config)

    # Print information about the percentage of trainable parameters
    print_trainable_parameters(model)

    # Training parameters
    trainer = Trainer(
        model=model,
        train_dataset=dataset,
        args=TrainingArguments(
            per_device_train_batch_size=1,
            gradient_accumulation_steps=4,
            warmup_steps=2,
            max_steps=20,
            learning_rate=2e-4,
            fp16=True,
            logging_steps=1,
```



```

        output_dir="outputs",
        optim="paged_adamw_8bit",
    ),
    data_collator=DataCollatorForLanguageModeling(tokenizer, mlm=False)
)

model.config.use_cache = False # re-enable for inference to speed up predictions for si

### SOURCE https://github.com/artidoro/qlora/blob/main/qlora.py
# Verifying the datatypes before training

dtypes = {}
for _, p in model.named_parameters():
    dtype = p.dtype
    if dtype not in dtypes: dtypes[dtype] = 0
    dtypes[dtype] += p.numel()
total = 0
for k, v in dtypes.items(): total += v
for k, v in dtypes.items():
    print(k, v, v/total)

do_train = True

# Launch training
print("Training...")

if do_train:
    train_result = trainer.train()
    metrics = train_result.metrics
    trainer.log_metrics("train", metrics)
    trainer.save_metrics("train", metrics)
    trainer.save_state()
    print(metrics)

###

# Saving model
print("Saving last checkpoint of the model...")
os.makedirs(output_dir, exist_ok=True)
trainer.model.save_pretrained(output_dir)

# Free memory for merging weights
del model
del trainer
torch.cuda.empty_cache()

output_dir = "results/llama2/final_checkpoint"
train(model, tokenizer, dataset, output_dir)

```

If you prefer to have a number of epochs (entire training dataset will be passed through the model) instead of a number of training steps (forward and backward passes through the model with one batch of data), you can replace the `max_steps` argument by `num_train_epochs`.

To later load and use the model for inference, we have used the

`trainer.model.save_pretrained(output_dir)` function, which saves the fine-tuned model's weights, configuration, and tokenizer files.

Step	Training Loss
1	1.873400
2	1.794300
3	2.062500
4	1.814200
5	1.383500
6	1.751800
7	1.607400
8	1.153100
9	1.215900
10	1.396800
11	1.380100
12	1.143000
13	1.103000
14	1.469400
15	1.009700

```
***** train metrics *****
epoch                = 0.0
total_flos           = 258560GF
train_loss           = 1.4772
train_runtime        = 0:00:47.77
train_samples_per_second = 1.256
train_steps_per_second = 0.314
{'train_runtime': 47.7745, 'train_samples_per_second': 1.256, 'train_steps_per_second': 0.314, 'total_flos': 277627574255616.0, 'train_loss': 1.4772128502527873, 'epoch': 0.0}
Saving last checkpoint of the model...
```

Fine-tuning llama2 results on [databricks-dolly-15k](#) dataset

Unfortunately, it is possible that the latest weights are not the best. To solve this problem, you can implement a `EarlyStoppingCallback`, from transformers, during your fine-tuning. This will enable you to regularly test your model on the validation set, if you have one, and keep only the best weights.

Merge weights

Once we have our fine-tuned weights, we can build our fine-tuned model and save it to a new directory, with its associated tokenizer. By performing these steps, we can have a memory-efficient fine-tuned model and tokenizer ready for inference!

```
model = AutoPeftModelForCausalLM.from_pretrained(output_dir, device_map="auto", torch_dtype=
model = model.merge_and_unload()

output_merged_dir = "results/llama2/final_merged_checkpoint"
os.makedirs(output_merged_dir, exist_ok=True)
model.save_pretrained(output_merged_dir, safe_serialization=True)

# save tokenizer for easy inference
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.save_pretrained(output_merged_dir)
```

Conclusion

We hope you have enjoyed this article!

You are now able to fine-tune LLaMA 2 models on your own datasets!

In our next tutorial, you will discover how to **Deploy your Fine-tuned LLM on OVHcloud AI Deploy for inference!**

Mathieu Busquet

+ posts



I am an *engineering student* who has been working at OVHcloud for a few months. I am familiar with several computer languages, but within my studies, I specialized in artificial intelligence and Python is therefore my main working tool.

It is a growing field that allows me to discover and understand things, to create but also as you see to explain them :)



[← Previous Post](#)

[Next Post →](#)

Copyright © 2023 OVHcloud Blog | Powered by Astra WordPress Theme

OVHcloud       