Extended Guide: Instruction-tune Llama 2

July 26, 2023 10 minute read View Code

This blog post is an extended guide on instruction-tuning Llama 2 from Meta AI. The idea of the blog post is to focus on creating the instruction dataset, which we can then use to fine-tune the base model of Llama 2 to follow our instructions.

The goal is to create a model which can create instructions based on input. The idea behind this is that this can then be used for others to create instruction data from inputs. That's especially helpful if you want to personalize models for, e.g., tweeting, email writing, etc, which means that you would be able to generate an instruction dataset from your emails to then train a model to mimic your email writing.

Okay, so can we get started on this? In the blog, we are going to:

1. Define the use case and create a prompt template for instructions
2. Create an instruction dataset
3. Instruction-tune Llama 2 using `trl` and the `SFTTrainer`
4. Test the Model and run Inference

Note: This tutorial was created and run on a g5.2xlarge AWS EC2 Instance, including an NVIDIA A10G GPU.

1. Define the use case and create a prompt template for instructions

Before we describe our use case, we need to better understand what even is an instruction.

> *"An instruction is a piece of text or prompt that is provided to an LLM, like Llama, GPT-4, or Claude, to guide it to generate a response. Instructions allow humans to steer the conversation and constrain the language model's output to be more natural, useful, and aligned with the user's goals. Crafting clear, well-formulated instructions is key to productive conversations."*

Examples of instructions are listed below in the table.

Capability	Example Instruction
Brainstorming	Provide a diverse set of creative ideas for new flavors of ice cream.
Classification	Categorize these movies as either comedy, drama, or horror based on the plot summary.
Closed QA	Answer the question 'What is the capital of France?' with a single word.
Generation	Write a poem in the style of Robert Frost about nature and the changing seasons.
Information Extraction	Extract the names of the main characters from this short story.
Open QA	Why do leaves change color in autumn? Explain the scientific reasons.
Summarization	Summarize this article on recent advancements in renewable energy in 2-3 sentences.

As described in the beginning, we want to fine-tune a model to be able to generate instructions based on input. (output). We want to use this as a way to create synthetic datasets to personalize LLMs and Agents.

Converting the idea into a basic prompt template following the Alpaca format we get.

```
### Instruction:
Use the Input below to create an instruction, which could have been used to generate the input us

### Input:
Dear [boss name],

I'm writing to request next week, August 1st through August 4th,
off as paid time off.

I have some personal matters to attend to that week that require
me to be out of the office. I wanted to give you as much advance
notice as possible so you can plan accordingly while I am away.

Please let me know if you need any additional information from me
or have any concerns with me taking next week off. I appreciate you
considering this request.

Thank you, [Your name]

### Response:
Write an email to my boss that I need next week 08/01 - 08/04 off.
```

2. Create an instruction dataset

After we defined our use case and prompt template, we need to create our instruction dataset. Creating a high-quality instruction dataset is key for a good-performing model. Research shows that "Less Is More for Alignment" shows that creating a high-quality, low-quantity (~1000 samples) dataset can achieve the same performance as less-quality and high-quantity datasets.

There are several ways to create an instruction dataset, including:

1. Using an existing dataset and converting it into an instruction dataset, e.g., FLAN
2. Use existing LLMs to create synthetically instruction datasets, e.g., Alpaca
3. Use Humans to create instructions datasets, e.g., Dolly.

Each of the methods has its own advantages and disadvantages and depends on the budget, time, and quality requirements. For example, using an existing dataset is the easiest but might not be tailored to your specific use case, while using humans might be the most accurate but can be time-consuming and expensive. It is also possible to combine several methods to create an instruction dataset, as shown in Orca: Progressive Learning from Complex Explanation Traces of GPT-4.

To keep it simple, we are going to use Dolly an open-source dataset of instruction-following records generated by thousands of Databricks employees in several of the behavioral categories outlined in the InstructGPT paper, including brainstorming, classification, closed QA, generation, information extraction, open QA, and summarization.

Let's start coding, but first, let's install our dependencies.

```
!pip install "transformers==4.34.0" "datasets==2.13.0" "peft==0.4.0" "accelerate==0.23.0" "bitsar
```

To load the `databricks/databricks-dolly-15k` dataset, we use the `load_dataset()` method from the 🤗 Datasets library.

```
from datasets import load_dataset
from random import randrange

# Load dataset from the hub
dataset = load_dataset("databricks/databricks-dolly-15k", split="train")

print(f"dataset size: {len(dataset)}")
print(dataset[randrange(len(dataset))])
# dataset size: 15011
```

To instruct tune our model, we need to convert our structured examples into a collection of tasks described via instructions. We define a `formatting_function` that

takes a sample and returns a string with our format instruction.

```
def format_instruction(sample):
    return f"""### Instruction:
Use the Input below to create an instruction, which could have been used to generate the input us

### Input:
{sample['response']}

### Response:
{sample['instruction']}
"""
```

Let's test our formatting function on a random example.

```
from random import randrange

print(format_instruction(dataset[randrange(len(dataset))]))
```

```
### Instruction:
Use the Input below to create an instruction, which could have been used to generate the input us

### Input:
22nd July 1947

### Response:
When was the Indian National Flag adopted
```

3. Instruction-tune Llama 2 using `trl` and the `SFTTrainer`

We will use the recently introduced method in the paper "QLoRA: Quantization-aware Low-Rank Adapter Tuning for Language Generation" by Tim Dettmers et al. QLoRA is a new technique to reduce the memory footprint of large language models during finetuning, without sacrificing performance. The TL;DR; of how QLoRA works is:

- Quantize the pre-trained model to 4 bits and freeze it.
- Attach small, trainable adapter layers. (LoRA)
- Finetune only the adapter layers while using the frozen quantized model for context.

If you want to learn more about QLoRA and how it works, I recommend you to read the Making LLMs even more accessible with bitsandbytes, 4-bit quantization and QLoRA blog post.

Flash Attention

Flash Attention is a an method that reorders the attention computation and leverages classical techniques (tiling, recomputation) to significantly speed it up and reduce memory usage from quadratic to linear in sequence length. It is based on the paper "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". The TL;DR; accelerates training up to 3x. Learn more at FlashAttention. Flash Attention is currently only available for Ampere (A10, A40, A100, ...) & Hopper (H100, ...) GPUs. You can check if your GPU is supported and install it using the following command:

Note: If your machine has less than 96GB of RAM and lots of CPU cores, reduce the number of `MAX_JOBS`. On the `g5.2xlarge` we used `4`.

```
python -c "import torch; assert torch.cuda.get_device_capability()[0] >= 8, 'Hardware not support
pip install ninja packaging
MAX_JOBS=4 pip install flash-attn --no-build-isolation
```

Installing flash attention can take quite a bit of time (10-45 minutes).

The example supports the use of Flash Attention for all Llama checkpoints, but is not enabled by default. To use Flash Attention change the value of `use_flash_attentin` to `True`

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig

use_flash_attention = False

# Hugging Face model id
model_id = "NousResearch/Llama-2-7b-hf"  # non-gated
# model_id = "meta-llama/Llama-2-7b-hf" # gated


# BitsAndBytesConfig int-4 config
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True, bnb_4bit_use_double_quant=True, bnb_4bit_quant_type="nf4", bnb_4bit_compu
)

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    quantization_config=bnb_config,
    use_cache=False,
    use_flash_attention_2=use_flash_attention,
    device_map="auto",
)
model.config.pretraining_tp = 1


tokenizer = AutoTokenizer.from_pretrained(model_id)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"
```

The `SFTTrainer` supports a native integration with `peft`, which makes it super easy to efficiently instruction tune LLMs. We only need to create our `LoRAConfig` and provide it to the trainer.

```
from peft import LoraConfig, prepare_model_for_kbit_training, get_peft_model

# LoRA config based on QLoRA paper
peft_config = LoraConfig(
        lora_alpha=16,
        lora_dropout=0.1,
        r=64,
        bias="none",
        task_type="CAUSAL_LM",
)


# prepare model for training
model = prepare_model_for_kbit_training(model)
model = get_peft_model(model, peft_config)
```

Before we can start our training we need to define the hyperparameters (`TrainingArguments`) we want to use.

```
from transformers import TrainingArguments

args = TrainingArguments(
    output_dir="llama-7-int4-dolly",
    num_train_epochs=3,
    per_device_train_batch_size=6 if use_flash_attention else 4,
    gradient_accumulation_steps=2,
    gradient_checkpointing=True,
    optim="paged_adamw_32bit",
    logging_steps=10,
    save_strategy="epoch",
    learning_rate=2e-4,
    bf16=True,
    tf32=True,
    max_grad_norm=0.3,
    warmup_ratio=0.03,
    lr_scheduler_type="constant",
    disable_tqdm=True # disable tqdm since with packing values are in correct
)
```

We now have every building block we need to create our `SFTTrainer` to start then training our model.

```
from trl import SFTTrainer

max_seq_length = 2048 # max sequence length for model and packing of the dataset

trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    peft_config=peft_config,
    max_seq_length=max_seq_length,
    tokenizer=tokenizer,
    packing=True,
    formatting_func=format_instruction,
    args=args,
```

```
)
```

Start training our model by calling the `train()` method on our `Trainer` instance.

```
# train
trainer.train() # there will not be a progress bar since tqdm is disabled

# save model
trainer.save_model()
```

The training without Flash Attention enabled took 03:08:00 on a `g5.2xlarge`. The instance costs `1,212$/h` which brings us to a total cost of `3.7$`. The training with Flash Attention enabled took 02:08:00 on a `g5.2xlarge`. The instance costs `1,212$/h` which brings us to a total cost of `2.6$`.

The results using Flash Attention are mind blowing and impressive, 1.5x faster and 30% cheaper.

4. Test Model and run Inference

After the training is done we want to run and test our model. We will use `peft` and `transformers` to load our LoRA adapter into our model.

```
if use_flash_attention:
    # unpatch flash attention
    from utils.llama_patch import unplace_flash_attn_with_attn
    unplace_flash_attn_with_attn()

import torch
from peft import AutoPeftModelForCausalLM
from transformers import AutoTokenizer

args.output_dir = "llama-7-int4-dolly"

# load base LLM model and tokenizer
model = AutoPeftModelForCausalLM.from_pretrained(
    args.output_dir,
    low_cpu_mem_usage=True,
    torch_dtype=torch.float16,
    load_in_4bit=True,
)
tokenizer = AutoTokenizer.from_pretrained(args.output_dir)
```

Let's load the dataset again with a random sample to try to generate an instruction.

```
from datasets import load_dataset
from random import randrange


# Load dataset from the hub and get a sample
dataset = load_dataset("databricks/databricks-dolly-15k", split="train")
```

```
sample = dataset[randrange(len(dataset))]

prompt = f"""### Instruction:
Use the Input below to create an instruction, which could have been used to generate the input us

### Input:
{sample['response']}

### Response:
"""

input_ids = tokenizer(prompt, return_tensors="pt", truncation=True).input_ids.cuda()
# with torch.inference_mode():
outputs = model.generate(input_ids=input_ids, max_new_tokens=100, do_sample=True, top_p=0.9,tempe

print(f"Prompt:\n{sample['response']}\n")
print(f"Generated instruction:\n{tokenizer.batch_decode(outputs.detach().cpu().numpy(), skip_spec
print(f"Ground truth:\n{sample['instruction']}")
```

result

```
Prompt:
Jack Dorsey, Noah Glass, Biz Stone, Evan Williams

Generated instruction:
Extract the founders of Twitter from the passage. Display the results in a comma separated format

Ground truth:
List the founders of Twitter from the above passage in a comma separated format.
```

Nice! our model works! If want to accelerate our model we can deploy it with Text Generation Inference. Therefore we would need to merge our adapter weights into the base model.

```
from peft import AutoPeftModelForCausalLM

model = AutoPeftModelForCausalLM.from_pretrained(
    args.output_dir,
    low_cpu_mem_usage=True,
)

# Merge LoRA and base model
merged_model = model.merge_and_unload()

# Save the merged model
merged_model.save_pretrained("merged_model",safe_serialization=True)
tokenizer.save_pretrained("merged_model")

# push merged model to the hub
# merged_model.push_to_hub("user/repo")
# tokenizer.push_to_hub("user/repo")
```

Thanks for reading! If you have any questions, feel free to contact me on Twitter or LinkedIn.