

[W&B Fully Connected](#) > [Articles](#)

How to Fine-Tune an LLM Part 2: Instruction Tuning Llama 2

In part 1, we prepped our dataset. In part 2, we train our model

[Thomas Capelle](#)

Created on October 13 | Last edited on February 21

In our [previous article on datasets for instruction tuning](#), we explored how to create an instruction dataset for a Llama 2 model. In this article, we'll fine-tune it using the Alpaca dataset we previously prepared.

This codebase and article aim to be pedagogical and straightforward. The main goal here is to understand what is happening under the hood when you fine-tune an LLM for instruction tuning.

There are more sophisticated training recipes out there like the [Hugging Face transformers' Trainer](#), [trl](#), [Axolotl](#), [Peft](#), [llama_recipes](#), [the alignment_handbook](#), etc. In this article, we will try our best to make it as simple as possible and make the training loop straightforward to follow.



Llama al alpaca meeting again (the left one looks more Guanaco to me but that might be a personal thing)

What We'll Be Covering:

[Downloading the Preprocessed Dataset from W&B](#)

[Loading Local JSON Data from Disk Using HuggingFace Datasets](#)

[DataLoader](#)

[Training Loop](#)

[Freezing the Model to Save Memory: 🤖 Jeremy Howard Style](#)

[Optimizer and Scheduler](#)

[Sampling from the Model](#)

[Validation Step](#)

[A Simple PyTorch Training Loop for Your LLM](#)

[Results](#)

[GPT-4 based evaluation](#)

[Evaluation Results](#)

[Continue to Part 3 📌](#)

[Conclusion and Final Remarks](#)



NOTE: The code associate with this post can be found [here](#).

Downloading the Preprocessed Dataset from W&B

Let's get started. In the previous article, we saved our preprocessed dataset as a [Weights & Biases Artifact](#), so we can easily retrieve the dataset from there. Here's the code:

```
import wandb
from pathlib import Path

run = wandb.init(project="alpaca_ft")
artifact = run.use_artifact('capecape/alpaca_ft/packed_alpaca:v0', type='dataset')
artifact_dir = Path(artifact.download())
```

As we kept the dataset as plain JSON files, we can open them directly using the Python built-in `json` module:

```
import json

def load_jsonl(filename):
    data = []
    with open(filename, 'r') as file:
        for line in file:
            data.append(json.loads(line))
    return data

train_ds_packed = load_jsonl(artifact_dir/"train_packed_alpaca.jsonl")
eval_ds_packed = load_jsonl(artifact_dir/"eval_packed_alpaca.jsonl")
```

From there, we can continue our training!

Loading Local JSON Data from Disk Using HuggingFace Datasets

`project("capecape", "alpaca_ft").artifact("packed_alpaca_hf")`



packed_alpaca_hf

latest ▾

Version

Metadata

Usage

Files

Lineage

🔍 Search keys

Key

Value

▾ Version Metadata

max_seq_len

1,024

model_id

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

A better container for your dataset than plain JSON might be something like the [Hugging Face datasets library](#). This has many advantages, such as fast loading, built-in map/filter methods, and bucket streaming, among others. You can quickly convert the `jsonl` files we created to `datasets` format by using the `load_from_disk` method:

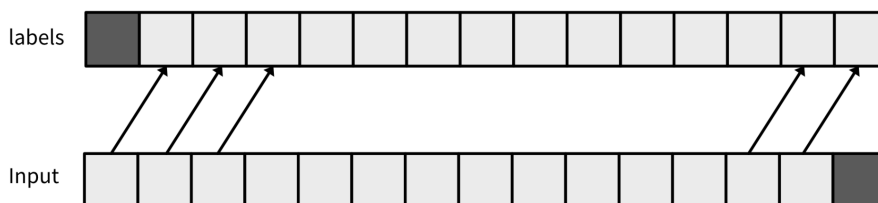
```
import wandb
from datasets import load_from_disk # for some reason load_dataset gives an error

run = wandb.init(project="alpaca_ft")
artifact = run.use_artifact('capecape/alpaca_ft/packed_alpaca_hf:v0', type='dataset')
artifact_dir = artifact.download()
ds_packed = load_from_disk(artifact_dir)

# we are back where we started!
train_ds_packed = ds_packed["train"]
eval_ds_packed = ds_packed["eval"]
max_seq_len = artifact.metadata["max_seq_len"]
```

DataLoader

As we are training for completion, the labels (or targets) will be the inputs shifted by one. We will train with regular cross-entropy and predict the next token on this packed dataset.



as input and target are the same but shifted, we lose one token at each end

In code, we accomplish this by setting the labels as the inputs shifted by one:

```
{"input_ids": input_ids[:-1], "labels": input_ids[1:]} # you actually drop one value
```

Beware that [the Hugging Face model does this for you automatically](#) when computing the loss on the model attribute (`ModelOutput.loss``), in that case, inputs and labels are identical.

We can now put together a standard [PyTorch](#) DataLoader:

```
from torch.utils.data import DataLoader
from transformers import default_data_collator

batch_size = 8 # I have an A100 GPU with 40GB of RAM 😎

train_dataloader = DataLoader(
    train_ds_packed,
    batch_size=batch_size,
```

```

        collate_fn=default_data_collator, # we don't need any special collator 😎
    )

eval_dataloader = DataLoader(
    eval_ds_packed,
    batch_size=batch_size,
    collate_fn=default_data_collator,
    shuffle=False,
)

```

It's always a good idea to check what a batch looks like. You can quickly do this by sampling from the DataLoader:

```

b = next(iter(train_dataloader))
b.keys(), b["input_ids"][0][:25], b["labels"][0][:25]

>> (dict_keys(['input_ids', 'labels']),
     tensor([ 1, 13866, 338, 385, 15278, 393, 16612, 263, 3414, 29889,
              14350, 263, 2933, 393, 7128, 2486, 1614, 2167, 278, 2009,
              29889, 13, 13, 2277, 29937]),
     tensor([13866, 338, 385, 15278, 393, 16612, 263, 3414, 29889, 14350,
              263, 2933, 393, 7128, 2486, 1614, 2167, 278, 2009, 29889,
              13, 13, 2277, 29937, 2799])) ### <<< ---- shifted by 1

# input_ids.shape: (8, 1024), labels.shape: (8, 1024)

```

Everything looks fine; let's train this thing!

Training Loop

[open](#) [code](#)

We'll start by training a model, naively making the model complete the sentence. As an exercise, I will implement this in pure PyTorch, so no abstractions are present besides grabbing the pre-trained model from the [HuggingFace Hub](#).

I like storing the configuration hyperparameters in a `SimpleNamespace`. It's like a dictionary with `.dot` attribute access. Then, I can access my batch size by doing `config.batch_size` instead of `config["batch_size"]`.

We will use some necessary tricks to make this possible:

- We're going to train a subset of the model parameters instead of the full model
- We're going to use gradient checkpointing to save on GPU memory. Checkpointing is a method that mitigates memory usage by eliminating and recalculating certain layers' activations during a backward pass, trading additional computation time for decreased memory usage.
- Automatic Mixed Precision: This technique makes training considerably faster as the computations are done in half-precision (`float16` or `bfloat16`). You can read more about this technique [here](#).
- We will implement an evaluation step that will sample from the model regularly.

Let's get started!

```
from types import SimpleNamespace

gradient_accumulation_steps = 32 // batch_size

config = SimpleNamespace(
    model_id='meta-llama/Llama-2-7b-hf',
    dataset_name="alpaca-gpt4",
    precision="bf16", # faster and better than fp16, requires new GPUs
    n_freeze=24, # How many layers we don't train, LLama 7B has 32.
    lr=2e-4,
    n_eval_samples=10, # How many samples to generate on validation
    max_seq_len=max_seq_len, # Length of the sequences to pack
    epochs=3, # we do 3 passes over the dataset.
    gradient_accumulation_steps=gradient_accumulation_steps, # every how many iterations
    batch_size=batch_size, # what my GPU can handle, depends on how many layers are we
    log_model=False, # upload the model to W&B?
    mom=0.9, # optim param
    gradient_checkpointing = True, # saves even more memory
    freeze_embed = True, # why train this? let's keep them frozen *
)

config.total_train_steps = config.epochs * len(train_dataloader) // config.gradient_acc
```

We first get a pre-trained model with some configuration parameters:

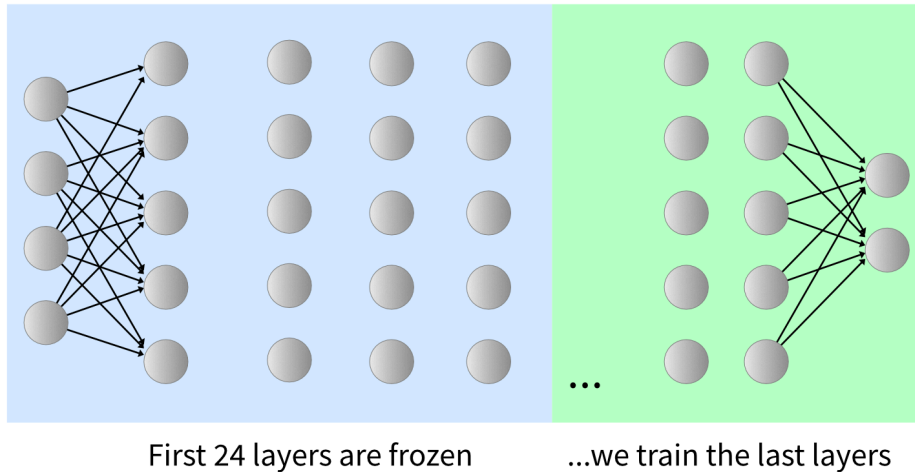
```
model = AutoModelForCausalLM.from_pretrained(
    config.model_id,
    device_map=0,
    trust_remote_code=True,
    low_cpu_mem_usage=True,
    torch_dtype=torch.bfloat16,
    use_cache=False,
)
```

Freezing the Model to Save Memory: 🤖 Jeremy Howard Style

Training the full models is expensive, but if you have a GPU that can fit the entire model, you can skip this part. Instead, we will train a subset of the model parameters. This technique has worked in other domains and was pioneered by [Jeremy and Sebastian Ruder](#).

Transformer-based models like Llama are a stack of identical layers on top of each other with a classification layer at the end. [Llama 2-7b](#) has 32 transformer layers, so we will only train the last 8 of them. You can experiment with how many layers to freeze. You always want to train the classification head (the last layer that makes the predictions).

In the rest of this piece, we'll explore how one can train the full model leveraging efficient parameter fine-tuning techniques like [LoRA](#).



This technique has proven to work well in most cases; try it!

Before trying fancy parameter-efficient methods, let's go [Jeremy style](#) and freeze most model layers. After loading the model, we freeze most of it. This way, we save a ton of memory by not computing gradients on the frozen layers.

```
n_freeze = 24. # you can play with this parameter

# freeze layers (disable gradients)
for param in model.parameters(): param.requires_grad = False
for param in model.lm_head.parameters(): param.requires_grad = True
for param in model.model.layers[n_freeze:].parameters(): param.requires_grad = True

>> Total params: 6738.42M, Trainable: 1750.14M
```

You can even gain a little bit more memory by freezing the embeddings!

```
# Just freeze embeddings for small memory decrease
if config.freeze_embed:
    model.model.embed_tokens.weight.requires_grad_(False);
```

You can also use gradient checkpointing to save even more (this makes training slower; how much it will depend on your particular configuration). There is an excellent article on the Huggingface website about how to [fit large models on memory](#); I encourage you to check it!

```
# save more memory
if config.gradient_checkpointing:
    model.gradient_checkpointing_enable(gradient_checkpointing_kwargs={"use_reentrant":
```

💡 The code is borrowed from excellent [Jeremy's notebook](#) and [video](#)

💡 Had to fix this cell as the "use_reentrant" argument is needed to make gradients flow from the

Optimizer and Scheduler

[open](#) [code](#)

We'll now set the optimizer and scheduler for our training. We need this to tell PyTorch how to compute the optimization step and adjust the learning rate accordingly. There are probably fancier techniques to try, but `Adam` and `cosine_schedule` are safe starting points. We will also set up our training loop using `bfloat`, to make good use of those TensorCores available on modern Nvidia GPUs. We will also set up the `loss_fn` as [Cross Entropy](#).

```
from transformers import get_cosine_schedule_with_warmup

optim = torch.optim.Adam(model.parameters(), lr=config.lr, betas=(0.9,0.99), eps=1e-5)
scheduler = get_cosine_schedule_with_warmup(
    optim,
    num_training_steps=config.total_train_steps,
    num_warmup_steps=config.total_train_steps // 10,
)

def loss_fn(x, y):
    "A Flat CrossEntropy"
    return torch.nn.functional.cross_entropy(x.view(-1, x.shape[-1]), y.view(-1))
```

We grab the scheduler from the transformer library; why not? It's already there waiting for us. You can implement the scheduler [Karpathy's style](#) if you like.

Sampling from the Model

We are almost there! Let's create a simple function to sample from the model now and then to visually see what the model is outputting.

Let's wrap the `model.generate` method for simplicity. You can grab the default sampling parameters from the `GenerationConfig` and pass the corresponding `model_id`. This will hold the defaults for parameters like temperature, top p, etc.

```
from transformers import GenerationConfig
gen_config = GenerationConfig.from_pretrained(config.model_id)

def generate(prompt, max_new_tokens=100, gen_config=gen_config):
    with torch.inference_mode():
        tokenized_prompt = tokenizer(prompt, return_tensors='pt')['input_ids'].cuda()
        output = model.generate(tokenized_prompt,
                                max_new_tokens=max_new_tokens,
                                generation_config=gen_config)
    return tokenizer.decode(output[0][len(tokenized_prompt[0]):], skip_special_tokens=T
```


We'll run our model over the `eval_dataset` every 1/10th of the total train steps and log a table to Weights & Biases containing the model predictions. We will also add the relevant sampling parameters in case we change them later on.

```
def prompt_table(prompts, log=True):
    table = wandb.Table(columns=["prompt", "generation", "concat", "max_new_tokens", "t
    for prompt in progress_bar(prompts):
        out = generate(prompt, test_config.max_new_tokens, test_config.gen_config)
        table.add_data(prompt, out, prompt+out, test_config.max_new_tokens, test_config
    if log:
        wandb.log({"predictions":table})
    return table
```

Validation Step

[open](#) [code](#)

You should always have some validation done during training runs. You may skip this if the training is concise, but computing metrics on a validation dataset can give you precious insight into how the training is going. For LLMs, you also want to sample from the model to visualize how the alignment with your data is going. We implement a `validate` function that does a couple of things:

- Iterates through the `eval_dataloader` and accumulates loss and accuracy
- Logs those metrics to W&B over the entire dataset
- Sample from the model and log the generation to a [W&B Table](#).

```
@torch.no_grad()
def validate():
    model.eval()
    eval_acc = Accuracy()
    for step, batch in enumerate(tqdm(eval_dataloader)):
        batch = to_gpu(batch)
        with torch.amp.autocast("cuda", dtype=torch.bfloat16):
            out = model(**batch)
            loss = loss_fn(out.logits, batch["labels"]) # you could use out.loss and n
            eval_acc.update(out.logits, batch["labels"])
    # we log results at the end
    wandb.log({"eval_loss": loss.item(),
              "eval_accuracy": eval_acc.compute()})
    prompt_table(eval_dataset[:config.n_eval_samples], log=True)
    model.train();
```

It's a good idea to run validation after some steps to assess that everything is going okay, as this is a short fine-tuning. You want to call `validate` at least a couple of times during training; this will depend on the task and the dataset size. For this experiment, we will perform validation 3 times (at the end of every epoch).

A Simple PyTorch Training Loop for Your LLM

[open](#) [code](#)

This [PyTorch](#) training loop is a standard training loop that iterates through the train data loader and performs evaluation every a fixed amount of steps. It saves the model at the end of the training.

- **Gradient accumulation:** This technique enables us to simulate larger batch sizes, which is very useful when using GPUs with less memory capabilities.
- **Sampling and model checkpoint saving** (this trains very fast, so there is no need to save multiple checkpoints)
- **Compute token accuracy:** It is a better metric than loss because it is easy to understand, as the accuracy number represents a quantity we can interpret. Also, let's not forget that this is a classification task for the next token prediction! If you don't believe me, Jeremy Howard still suggests accuracy for Causal Language Modeling as the metric to go with.

```
wandb.init(project="alpaca_ft", # the project I am working on
           tags=["baseline", "7b"],
           job_type="train",
           config=config) # the Hyperparameters I want to keep track of

# Training
acc = Accuracy()
model.train()
train_step = 0
pbar = tqdm(total=config.total_train_steps)
for epoch in range(config.epochs):
    for step, batch in enumerate(train_dataloader):
        batch = to_gpu(batch)
        with torch.amp.autocast("cuda", dtype=torch.bfloat16):
            out = model(**batch)
            loss = loss_fn(out.logits, batch["labels"]) / config.gradient_accumulation
            loss.backward()
        if step % config.gradient_accumulation_steps == 0:
            # we can log the metrics to W&B
            wandb.log({"train/loss": loss.item() * config.gradient_accumulation_steps,
                      "train/accuracy": acc.update(out.logits, batch["labels"]),
                      "train/learning_rate": scheduler.get_last_lr()[0],
                      "train/global_step": train_step})
            optim.step()
            scheduler.step()
            optim.zero_grad(set_to_none=True)
            train_step += 1
            pbar.update(1)
    validate()
pbar.close()
# we save the model checkpoint at the end
save_model(
```

This trains in around 120 minutes on an A100.

The Hugging Face course has a similar to this one that uses [pure PyTorch to train a model from the HF hub](#).

We present the loss curves and the accuracy metrics. Our total training steps are around 1150 steps (3 epochs) with gradient accumulation steps = 4. We pass two samples before updating the gradients.

The figure displays three line charts for the 'ethereal-smoke-371' model. The top-left chart, 'Train Token Accuracy', shows accuracy fluctuating between 0.66 and 0.8, with a general upward trend from 0.66 to 0.74. The top-right chart, 'Train Loss', shows loss fluctuating between 0.8 and 1.3, with a general downward trend from 1.3 to 0.9. The bottom chart, 'Eval Loss', shows a steady decrease in loss from 0.937 to 0.933 over 1,000 steps.

| train/global_step | Train Token Accuracy | Train Loss | Eval Loss |
|-------------------|----------------------|------------|-----------|
| 0 | 0.66 | 1.3 | 0.937 |
| 200 | 0.74 | 0.9 | 0.936 |
| 400 | 0.74 | 0.9 | 0.935 |
| 600 | 0.74 | 0.9 | 0.934 |
| 800 | 0.74 | 0.9 | 0.933 |
| 1000 | 0.74 | 0.9 | 0.933 |

```
runs[0].loggedArtifactVersions.map((row, index) =>
row.file("predictions.table.json"))
```



These were samples generated and computed during training. We'll evaluate them on the entire test dataset in the next section.

GPT-4 based evaluation

gpt4 eval [code](#)

Let's use [GPT-4](#) to compare the results generated by the fine-tuned model against GPT-3.5. Let's also get GPT-4's reason for picking one over the other. This evaluation technique has been used in multiple places, for instance, [MT Bench](#) (MT-bench is a set of challenging multi-turn open-ended questions for evaluating chat assistants)

💡 You can read more about LLM supervised evaluation [in Ayush's article](#).

GPT4 is better at reasoning than GPT3.5. Also, it wouldn't be fair to use the same model generating one of the responses to judge itself. Of course, this technique is not perfect, and other studies have shown that sometimes this evaluation strategy may not be consistent with permutation (switching the answers) or even calling the model multiple times, which could lead to different responses due to the stochastic nature of the generations. One way to mitigate this is setting up the temperature sampling parameters closer to zero to make the model more deterministic.

The clear win this approach has is that one can quickly implement LLM-based evaluation, and using a powerful model like GPT-4, we can create a baseline score quickly. Ideally, you would want to set up a human-based assessment at some point, but this is more costly and slower to implement.

We can leverage [OpenAI function calling](#) to format the output of GPT-4 with the corresponding choice made and the reason.

```
def gpt4_judge(instruction, gen1, gen2, model="gpt-4"):
    system_prompt = ("You will be presented with a choice of two possible responses for\n"
                     "You have to pick the best one and give a reason why.\n"
                     "The reponse should follow the instructions and use the provided c\n"
                     "If both answers are equivalent, pick the value 0")
    message = "{instruction}\n Answer 1: \n{gen1}\n Answer 2:\n{gen2}".format(instruction, gen1, gen2)
    completion = openai.chat.completions.create(
        model=model,
        messages=[{"role": "system",
                   "content": system_prompt,
                   },
                 {"role": "user",
                   "content": message,
                   },
                ],
        function_call = {"name": "make_choice"},
        functions = [{
            "name": "make_choice",
            "description": "Select the best generation and explain why",
            "parameters": {
                "type": "object",
                "properties": {
                    "choice": {
                        "type": "integer",
                        "description": "the choosen alternative, zero if equivalent",
                    },
                    "argument": {
                        "type": "string",
                        "description": "Reason why the choice was made",
                    },
                },
                "required": ["choice", "argument"],
            },
        }],
    )
    return completion
```

You can inspect the results in the evaluation tables below. We generated 250 completions using GPT-3.5 and asked GPT-4 to pick the best one; we also left the possibility of marking both as equally good:

- Both models are good
- Fine-tuned Llama was better
- GPT-3.5 produced better output



To make our testing more robust, we inverted the order and asked GPT-4 again, and we only kept the choices were GPT-4 consistently picked the same answer no matter the order. For our surprise 34 times GPT-4 switched sides! So take this evaluation with a grain of salt!

Order matter: Inverting the query order makes GPT-4 switch sides (sometimes)

You can check GPT-4 inconsistency here:

- Occasionally it prefers short answers and then switches sides and values the explanation and the longer answer 🤔
- In some cases, it judges both equally good and, when inverted, prefers one.
- Check the answers below by clicking the < > 📌

`runs.summary["gpt4_eval_disagree"]`

| | prompt | llama7b_ft | gpt3\5 | choice | choice_inverted | reason | reason_i |
|----|--------|------------|--------|--------|-----------------|--------|----------|
| 12 | | | | | | | |

≡ ≡ = —

← < 12 - 12 of 34 > →

Export as CSV Columns... Reset table

Evaluation Results

You can also check why GPT-4 picked what on the "argument" column. The fine-tuned Llama is good but not near as good as GPT-3.5. This makes sense as to how the 7b model trained on a handful of GPT4 generations would be better than a probably much bigger model like GPT3.5. Anyway, other questions arise:

- Is the 7b model too small? If we switched to Llama2-13b, would the outcome be the same?
- Should we train more layers of the model? All layers?
- We will explore some of these questions in the following articles.


`runs.summary["gpt4_eval"]`

| | choice_name | choice_name.count |
|---|-------------|-------------------|
| 1 | both | |
| 2 | gpt3.5 | |
| 3 | llama7b_ft | |


≡ ≡ = —

← < 1 - 3 of 3 > →

Export as CSV Columns... Reset table

runs.summary["gpt4_eval"] 

| | prompt | llama7b_ft | gpt3\5 | choice | choice_inverte | reason | reason_inve |
|---|--------|------------|--------|--------|----------------|--------|-------------|
| 1 | | | | | | | |
| 2 | | | | | | | |


1 - 2 of 216
Export as CSV Columns... Reset table

Continue to Part 3

How to Fine-tune an LLM Part 3: ...

Exploring how to get the best out of the ...

How to Fine-Tune an LLM Part 1: ...

Learn how to fine-tune an LLM on an ...

Conclusion and Final Remarks

Fine-tuning a model on an instruction dataset is just a particular case of completion training, where one constructs the dataset in an organized way so it can learn to follow instructions. This is a small example to demystify the complexity of what's happening under the hood when using specialized libraries to fine-tune.

Of course, Llama 7B is the smallest of the models out there, and one may obtain better results using the bigger brothers, but we managed to give instruction capabilities to a pre-trained model that did not have them. Now the model replies most of the time in the format specified and generates reasonable answers.

GPT-4 tends to prefer GPT-3.5... "GPTs prefer GPTs. — Ayush T." 🤔

This is the first of two articles about Instruction tuning. In the following piece, we will train the model by using the Hugging Face ecosystem and W&B integration. This will significantly simplify the preprocessing and code one must write.

How to Fine-Tune an LLM Part 1: ...

Learn how to fine-tune an LLM on an ...

How to Run LLMs Locally With ...

This article explores how to run LLMs loca...

Training Tiny Llamas for Fun...


Exploring how SoftMax implementation can ...

How to Evaluate, Compare, and ...

This article provides an interactive look into ...

Tags: Articles, LLM, GenAI, NLP, Experiment, Framework / Integration, Intermediate, Fine-tuning

New course
RAG++ : From POC to Production


ENROLL NOW →

Created with ❤️ on Weights & Biases.

https://wandb.ai/capecape/alpaca_ft/reports/How-to-Fine-Tune-an-LLM-Part-2-Instruction-Tuning-Llama-2--Vmlldzo1NjY0MjE1

Made with Weights & Biases. [Sign up](#) or [log in](#) to create reports like this one.

Never lose track of another ML
project. Try W&B today.

[SIGN UP](#)

[TRY W&B NOW](#)



Weights & Biases

Get weekly updates with the latest ML news.

[Subscribe](#)

PRODUCTS

[Dashboard](#) [Sweeps](#) [Artifacts](#) [Reports](#) [Tables](#)

QUICKSTART

[Documentation](#)

RESOURCES

[Courses](#) [Forum](#) [Tutorials](#) [Benchmarks](#)

W&B

[About Us](#) [Authors](#) [Contact](#) [Terms of Service](#) [Privacy Policy](#)

Copyright ©2024 Weights & Biases. All rights reserved.