Fine-tune an LLM in minutes (ft. Llama 2, CodeLlama, Mistral, etc.)

View on GitHub

Tired of prompt engineering? Fine-tuning helps you get more out of a pretrained LLM by adjusting the model weights to better fit a specific task. This operational guide will help you take a base model and fine-tune it on your own dataset (API docs, conversation transcripts, etc.) in the matter of minutes.

The repository comes ready to use as-is with all the recommended, start-of-the-art optimizations for fast training results:

- Parameter-Efficient Fine-Tuning (PEFT) via LoRA adapters for faster convergence
- Flash Attention for fast and memory-efficient attention during training (note: only works with certain hardware, like A100s)
- Gradient checkpointing to reduce VRAM footprint, fit larger batches and get higher training throughput
- Distributed training via DeepSpeed so training scales optimally with multiple GPUs

The heavy lifting is done by the axolotl library. For the purposes of this guide, we'll fine-tune CodeLlama 7B to generate SQL queries, but the code is easy to tweak for many base models, datasets, and training configurations.

Best of all, using Modal for training means you never have to worry about infrastructure headaches like building images, provisioning GPUs, and managing cloud storage. If a training script runs on Modal, it's repeatable and scalable enough to ship to production right away.

Prerequisites

To follow along, make sure that you have completed the following:

1. Set up Modal account:

```
pip install modal
python3 -m modal setup
```

2. Create a HuggingFace secret in your workspace (only `HF_TOKEN` is needed, which you can get if you go into your Hugging Face settings > API tokens)

3. Clone the repository and navigate to the directory:

```
git clone https://github.com/modal-labs/llm-finetuning.git
cd llm-finetuning
```

Some models like Llama 2 also require that you apply for access, which you can do on the Hugging Face page (granted instantly).

Code overview

The source directory contains a training script to launch a training job in the cloud with your config/dataset (`config.yml` and `my_data.jsonl` , unless otherwise specified), as well as an inference engine for testing your training results.

We use Modal's built-in cloud storage system to share data across all functions in the app. In particular, we mount a persisting volume at `/pretrained` inside the container to store our pretrained models (so we only need to load them once) and another persisting volume at `/runs` to store our training config, dataset, and results for each run (for easier reproducibility and management).

Training

The training script contains three Modal functions that run in the cloud:

- `launch` prepares a new folder in the `/runs` volume with the training config and data for a new training job. It also ensures the base model is downloaded from HuggingFace.
- `train` takes a prepared run folder in the volume and performs the training job using the config and data.

- `merge` merges the trained adapter with the base model (as a CPU job).

By default, when you make local changes to either `config.yml` or `my_data.jsonl`, they will be used for your next training run. You can also specify which local config and data files to use with the `--config` and `--dataset` flags. See Making it your own for more details on customizing your dataset and config.

To kickstart a training job with the CLI, you need to specify the config and data files:

```
modal run --detach src.train --config=config/mistral.yml --data=data/sqlqa.jsonl
```

`--detach` *lets the app continue running even if your client disconnects.*

The training run folder name will be in the command output (e.g. `axo-2023-11-24-17-26-66e8`). You can check if your fine-tuned model is stored properly in this folder using `modal volume ls`.

Serving your fine-tuned model

Once a training run has completed, run inference to compare the model before/after training.

- `Inference.completion` can spawn a vLLM inference container for any pre-trained or fine-tuned model from a previous training job.

You can serve a model for inference using the following command, specifying which training run folder to load the model from with the `--run-folder` flag (run folder name is in the training log output):

```
modal run -q src.inference --run-folder /runs/axo-2023-11-24-17-26-66e8
```

We use vLLM to speed up our inference up to 24x.

Making it your own

Training on your own dataset, using a different base model, and activating another SOTA technique is as easy as modifying a couple files.

Dataset

Bringing your own dataset is as simple as creating a JSONL file — Axolotl supports many dataset formats (see more).

We recommend adding your custom dataset as a JSONL file in the `src` directory and making the appropriate modifications to your config, as explained below.

Config

All of your training parameters and options are customizable in a single config file. We recommend duplicating one of the `example_configs` to `src/config.yml` and modifying as you need. See an overview of Axolotl's config options here.

The most important options to consider are:

- Model

```yaml
base_model: codellama/CodeLlama-7b-Instruct-hf
```

- Dataset (by default we upload a local .jsonl file from the `src` folder in completion format, but you can see all dataset options here)

```yaml
datasets:
- path: my_data.jsonl
   ds_type: json
   type: completion
```

- LoRA

```yaml
adapter: lora # for qlora, or leave blank for full finetune
lora_r: 8
lora_alpha: 16
lora_dropout: 0.05
lora_target_modules:
  - q_proj
  - v_proj
```

- Multi-GPU training

  We recommend DeepSpeed for multi-GPU training, which is easy to set up. Axolotl provides several default deepspeed JSON configurations and Modal makes it easy to attach multiple GPUs of any type in code, so all you need to do is specify which of these configs you'd like to use.

  In your `config.yml`:

```
deepspeed: /root/axolotl/deepspeed/zero3.json
```

In `train.py`:

```
import os

N_GPUS = int(os.environ.get("N_GPUS", 2))
GPU_MEM = os.environ.get("GPU_MEM", "80GB")
GPU_CONFIG = modal.gpu.A100(count=N_GPUS, size=GPU_MEM)  # you can also change this in
```

- Logging with Weights and Biases

  To track your training runs with Weights and Biases:

  1. Create a Weights and Biases secret in your Modal dashboard, if not set up already (only the `WANDB_API_KEY` is needed, which you can get if you log into your Weights and Biases account and go to the Authorize page)
  2. Add the Weights and Biases secret to your app, so initializing your app in `common.py` should look like:

```
from modal import App, Secret

app = App("my_app", secrets=[Secret.from_name("huggingface"), Secret.from_name("my-wan
```

  3. Add your wandb config to your `config.yml`:

```
wandb_project: mistral-7b-samsum
wandb_watch: gradients
```

Once you have your trained model, you can easily deploy it to production for serverless inference via Modal's web endpoint feature (see example here). Modal will handle all the auto-scaling for you, so that you only pay for the compute you use!