Member-only story

Fine-Tuning Large Language Models (LLMs)

A conceptual overview with example Python code

Shaw Talebi

Published in Towards Data Science · 14 min read · Sep 11, 2023

This is the 5th article in a series on using Large Language Models (LLMs) in practice. In this post, we will discuss how to fine-tune (FT) a pre-trained LLM. We start by introducing key FT concepts and techniques, then finish with a concrete example of how to fine-tune a model (locally) using Python and Hugging Face's software ecosystem.

Tuning a language model. Image by author.

In the previous article of this series, we saw how we could build practical LLM-powered applications by integrating prompt engineering into our Python code. For the vast majority of LLM use cases, this is the initial approach I recommend because it requires significantly less resources and technical expertise than other methods while still providing much of the upside.

However, there are situations where prompting an existing LLM out-of-the-box doesn't cut it, and a more sophisticated solution is required. This is where model fine-tuning can help.

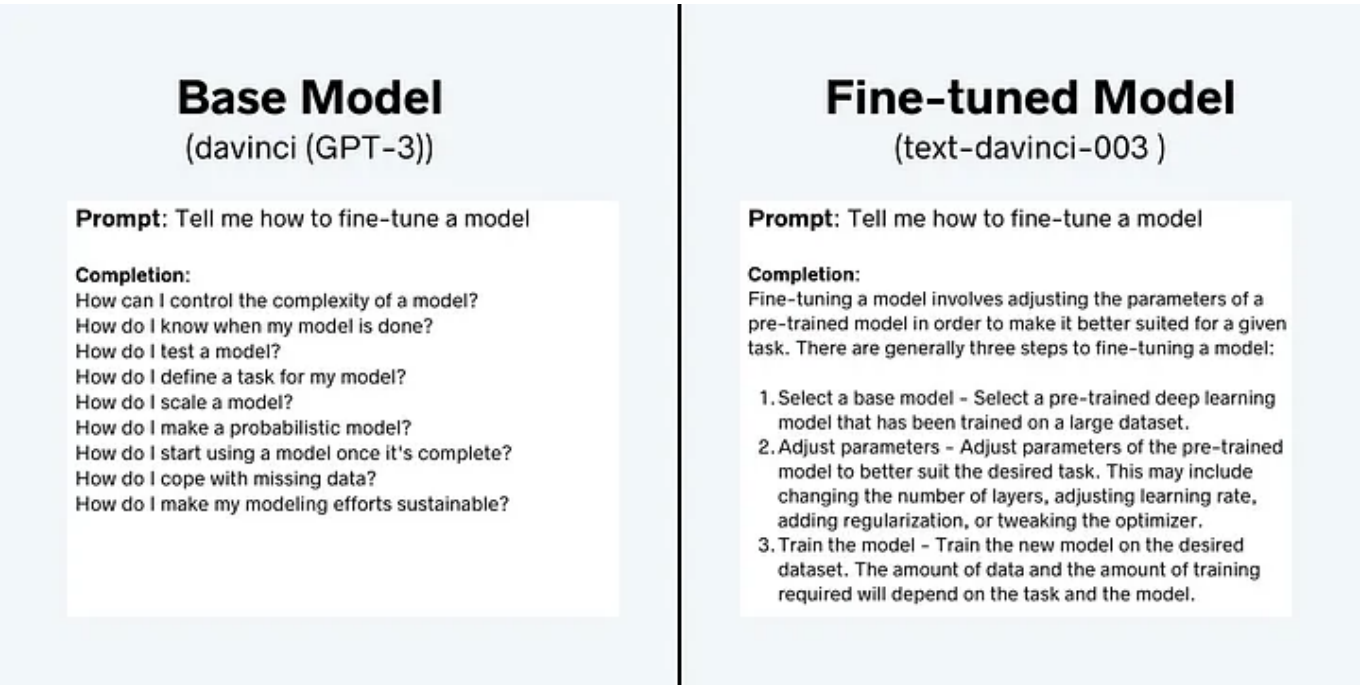
Supplemental Video.

What is Fine-tuning?

Fine-tuning is taking a pre-trained model and **training at least one internal model parameter** (i.e. weights). In the context of LLMs, what this typically accomplishes is transforming a general-purpose base model (e.g. GPT-3) into a specialized model for a particular use case (e.g. ChatGPT) [1].

The **key upside** of this approach is that models can achieve better performance while requiring (far) fewer manually labeled examples compared to models that solely rely on supervised training.

While strictly self-supervised base models can exhibit impressive performance on a wide variety of tasks with the help of prompt engineering [2], they are still word predictors and may generate completions that are not entirely helpful or accurate. For example, let's compare the completions of davinci (base GPT-3 model) and text-davinci-003 (a fine-tuned model).



Completion comparison of davinci (base GPT-3 model) and text-davinci-003 (a fine-tuned model). Image by author.

Notice the base model is simply trying to complete the text by listing a set of questions like a Google search or homework assignment, while the **fine-tuned model gives a more helpful response.** The flavor of fine-tuning used for text-davinci-003 is **alignment tuning,** which aims to make the LLM's responses more helpful, honest, and harmless, but more on that later [3,4].

Why Fine-tune

Fine-tuning not only improves the performance of a base model, but **a smaller (fine-tuned) model can often outperform larger (more expensive) models** on the set of tasks on which it was trained [4]. This was demonstrated by OpenAI with their first generation "InstructGPT" models, where the 1.3B parameter InstructGPT model completions were preferred over the 175B parameter GPT-3 base model despite being 100x smaller [4].

Although most of the LLMs we may interact with these days are not strictly self-supervised models like GPT-3, there are still drawbacks to prompting an

existing fine-tuned model for a specific use case.

A big one is LLMs have a finite context window. Thus, the model may perform sub-optimally on tasks that require a large knowledge base or domain-specific information [1]. Fine-tuned models can avoid this issue by "learning" this information during the fine-tuning process. This also precludes the need to jam-pack prompts with additional context and thus can result in lower inference costs.

3 Ways to Fine-tune

There are 3 **generic ways one can fine-tune** a model: self-supervised, supervised, and reinforcement learning. These are not mutually exclusive in that any combination of these three approaches can be used in succession to fine-tune a single model.

Self-supervised Learning

Self-supervised learning consists of **training a model based on the inherent structure of the training data.** In the context of LLMs, what this typically looks like is given a sequence of words (or tokens, to be more precise), predict the next word (token).

While this is how many pre-trained language models are developed these days, it can also be used for model fine-tuning. A potential use case of this is developing a model that can mimic a person's writing style given a set of example texts.

Supervised Learning

The next, and perhaps most popular, way to fine-tune a model is via **supervised learning.** This involves **training a model on input-output pairs** for a particular task. An example is **instruction tuning,** which aims to

improve model performance in answering questions or responding to user prompts [1,3].

The **key step** in supervised learning is **curating a training dataset.** A simple way to do this is to create question-answer pairs and integrate them into a prompt template [1,3]. For example, the question-answer pair: *Who was the 35th President of the United States? — John F. Kennedy* could be pasted into the below prompt template. More example prompt templates are available in section A.2.1 of ref [4].

```
"""Please answer the following question.

Q: {Question}

A: {Answer}"""
```

Using a prompt template is important because base models like GPT-3 are essentially "document completers". Meaning, given some text, the model generates more text that (statistically) makes sense in that context. This goes back to the previous blog of this series and the idea of "tricking" a language model into solving your problem via prompt engineering.

Prompt Engineering — How to trick AI into solving your problems

7 prompting tricks, Langchain, and Python example code

towardsdatascience.com

Reinforcement Learning

Finally, one can use **reinforcement learning (RL)** to fine-tune models. RL **uses a reward model to guide the training of the base model**. This can take many different forms, but the basic idea is to train the reward model to score language model completions such that they reflect the preferences of human labelers [3,4]. The reward model can then be combined with a reinforcement learning algorithm (e.g. Proximal Policy Optimization (PPO)) to fine-tune the pre-trained model.

An example of how RL can be used for model fine-tuning is demonstrated by OpenAI's InstructGPT models, which were developed through **3 key steps** [4].

1. Generate high-quality prompt-response pairs and fine-tune a pre-trained model using supervised learning. (~13k training prompts) *Note: One can (alternatively) skip to step 2 with the pre-trained model [3].*
2. Use the fine-tuned model to generate completions and have human-labelers rank responses based on their preferences. Use these preferences to train the reward model. (~33k training prompts)
3. Use the reward model and an RL algorithm (e.g. PPO) to fine-tune the model further. (~31k training prompts)

While the strategy above does generally result in LLM completions that are significantly more preferable to the base model, it can also come at a cost of lower performance in a subset of tasks. This drop in performance is also known as an **alignment tax** [3,4].

Supervised Fine-tuning Steps (High-level)

As we saw above, there are many ways in which one can fine-tune an existing language model. However, for the remainder of this article, we will

focus on fine-tuning via supervised learning. Below is a high-level procedure for supervised model fine-tuning [1].

1. **Choose fine-tuning task** (e.g. summarization, question answering, text classification)
2. **Prepare training dataset** i.e. create (100–10k) input-output pairs and preprocess data (i.e. tokenize, truncate, and pad text).
3. **Choose a base model** (experiment with different models and choose one that performs best on the desired task).
4. **Fine-tune model via supervised learning**
5. **Evaluate model performance**

While each of these steps could be an article of their own, I want to focus on **step 4** and discuss how we can go about training the fine-tuned model.

3 Options for Parameter Training

When it comes to fine-tuning a model with ~100M-100B parameters, one needs to be thoughtful of computational costs. Toward this end, an important question is — *which parameters do we (re)train?*

With the mountain of parameters at play, we have countless choices for which ones we train. Here, I will focus on **three generic options** of which to choose.

Option 1: Retrain all parameters

The first option is to **train all internal model parameters** (called **full parameter tuning**) [3]. While this option is simple (conceptually), it is the most computationally expensive. Additionally, a known issue with full

parameter tuning is the phenomenon of catastrophic forgetting. This is where the model "forgets" useful information it "learned" in its initial training [3].

One way we can mitigate the downsides of Option 1 is to freeze a large portion of the model parameters, which brings us to Option 2.

Option 2: Transfer Learning

The big idea with **transfer learning (TL)** is to preserve the useful representations/features the model has learned from past training when applying the model to a new task. This generally consists of **dropping "the head" of a neural network (NN) and replacing it with a new one** (e.g. adding new layers with randomized weights). *Note: The head of an NN includes its final layers, which translate the model's internal representations to output values.*

While leaving the majority of parameters untouched mitigates the huge computational cost of training an LLM, TL may not necessarily resolve the problem of catastrophic forgetting. To better handle both of these issues, we can turn to a different set of approaches.

Option 3: Parameter Efficient Fine-tuning (PEFT)

PEFT involves **augmenting a base model with a relatively small number of trainable parameters.** The key result of this is a fine-tuning methodology that demonstrates comparable performance to full parameter tuning at a tiny fraction of the computational and storage cost [5].

PEFT encapsulates a family of techniques, one of which is the popular **LoRA (Low-Rank Adaptation)** method [6]. The basic idea behind LoRA is to pick a subset of layers in an existing model and modify their weights according to the following equation.

$$h(x) = W_0 x + \Delta W x = W_0 x + BAx$$

$$\text{Where,} \quad W_0, \Delta W \in R^{d \times k}$$

$$B \in R^{d \times r}$$

$$A \in R^{r \times k}$$

Equation showing how weight matrices are modified for fine-tuning using LoRA [6]. Image by author.

Where $h()$ = a hidden layer that will be tuned, x = the input to $h()$, W_0 = the original weight matrix for the h, and ΔW = a matrix of trainable parameters injected into h. ΔW is decomposed according to $\Delta W = BA$, where ΔW is a d by k matrix, B is d by r, and A is r by k. r is the assumed "intrinsic rank" of ΔW (which can be as small as 1 or 2) [6].

Sorry for all the math, but the **key point is the (d * k) weights in W_0 are frozen and, thus, not included in optimization**. Instead, the ((d * r) + (r * k)) weights making up matrices B and A are the only ones that are trained.

Plugging in some made-up numbers for d=100, k=100, and r=2 to get a sense of the efficiency gains, the **number of trainable parameters drops from 10,000 to 400** in that layer. In practice, the authors of the LoRA paper cited a **10,000x reduction in parameter checkpoint size** using LoRA fine-tune GPT-3 compared to full parameter tuning [6].

To make this more concrete, let's see how we can use LoRA to fine-tune a language model efficiently enough to run on a personal computer.

Example Code: Fine-tuning an LLM using LoRA

In this example, we will use the Hugging Face ecosystem to fine-tune a language model to classify text as 'positive' or 'negative'. Here, we fine-tune *distilbert-base-uncased*, a ~70M parameter model based on BERT. Since this base model was trained to do language modeling and not classification, we employ **transfer learning** to replace the base model head with a classification head. Additionally, we use **LoRA** to fine-tune the model efficiently enough that it can run on my Mac Mini (M1 chip with 16GB memory) in a reasonable amount of time (~20 min).

The code, along with the conda environment files, are available on the GitHub repository. The final model and dataset [7] are available on Hugging Face.

YouTube-Blog/LLMs/fine-tuning at main · ShawhinT/YouTube-Blog

Codes to complement YouTube videos and blog posts on Medium. - YouTube-Blog/LLMs/fine-tuning at main ·...

github.com

Imports

We start by importing helpful libraries and modules. Datasets, transformers, peft, and evaluate are all libraries from Hugging Face (HF).

```
from datasets import load_dataset, DatasetDict, Dataset

from transformers import (
    AutoTokenizer,
    AutoConfig,
    AutoModelForSequenceClassification,
    DataCollatorWithPadding,
    TrainingArguments,
    Trainer)
```

```
from peft import PeftModel, PeftConfig, get_peft_model, LoraConfig
import evaluate
import torch
import numpy as np
```

Base model

Next, we load in our base model. The base model here is a relatively small one, but there are several other (larger) ones that we could have used (e.g. roberta-base, llama2, gpt2). A full list is available here.

```
model_checkpoint = 'distilbert-base-uncased'

# define label maps
id2label = {0: "Negative", 1: "Positive"}
label2id = {"Negative":0, "Positive":1}

# generate classification model from model_checkpoint
model = AutoModelForSequenceClassification.from_pretrained(
    model_checkpoint, num_labels=2, id2label=id2label, label2id=label2id)
```

Load data

We can then load our training and validation data from HF's datasets library. This is a dataset of 2000 movie reviews (1000 for training and 1000 for validation) with binary labels indicating whether the review is positive (or not).

```
# load dataset
dataset = load_dataset("shawhin/imdb-truncated")
dataset

# dataset =
# DatasetDict({
```

```
#     train: Dataset({
#         features: ['label', 'text'],
#         num_rows: 1000
#     })
#     validation: Dataset({
#         features: ['label', 'text'],
#         num_rows: 1000
#     })
# })
```

Preprocess data

Next, we need to preprocess our data so that it can be used for training. This consists of using a tokenizer to convert the text into an integer representation understood by the base model.

```
# create tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, add_prefix_space=Tru
```

To apply the tokenizer to the dataset, we use the *.map()* method. This takes in a custom function that specifies how the text should be preprocessed. In this case, that function is called *tokenize_function()*. In addition to translating text to integers, this function truncates integer sequences such that they are no longer than 512 numbers to conform to the base model's max input length.

```
# create tokenize function
def tokenize_function(examples):
    # extract text
    text = examples["text"]

    #tokenize and truncate text
    tokenizer.truncation_side = "left"
    tokenized_inputs = tokenizer(
        text,
```

```
            return_tensors="np",
            truncation=True,
            max_length=512
        )

        return tokenized_inputs

    # add pad token if none exists
    if tokenizer.pad_token is None:
        tokenizer.add_special_tokens({'pad_token': '[PAD]'})
        model.resize_token_embeddings(len(tokenizer))

    # tokenize training and validation datasets
    tokenized_dataset = dataset.map(tokenize_function, batched=True)
    tokenized_dataset

    # tokenized_dataset =
    # DatasetDict({
    #     train: Dataset({
    #        features: ['label', 'text', 'input_ids', 'attention_mask'],
    #         num_rows: 1000
    #     })
    #     validation: Dataset({
    #         features: ['label', 'text', 'input_ids', 'attention_mask'],
    #         num_rows: 1000
    #     })
    # })
```

At this point, we can also create a data collator, which will dynamically pad examples in each batch during training such that they all have the same length. This is computationally more efficient than padding all examples to be equal in length across the entire dataset.

```
    # create data collator
    data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Evaluation metrics

We can define how we want to evaluate our fine-tuned model via a custom function. Here, we define the *compute_metrics()* function to compute the model's accuracy.

```
# import accuracy evaluation metric
accuracy = evaluate.load("accuracy")

# define an evaluation function to pass into trainer later
def compute_metrics(p):
    predictions, labels = p
    predictions = np.argmax(predictions, axis=1)

    return {"accuracy": accuracy.compute(predictions=predictions,
                                         references=labels)}
```

Untrained model performance

Before training our model, we can evaluate how the base model with a randomly initialized classification head performs on some example inputs.

```
# define list of examples
text_list = ["It was good.", "Not a fan, don't recommed.",
"Better than the first one.", "This is not worth watching even once.",
"This one is a pass."]

print("Untrained model predictions:")
print("----------------------------")
for text in text_list:
    # tokenize text
    inputs = tokenizer.encode(text, return_tensors="pt")
    # compute logits
    logits = model(inputs).logits
    # convert logits to label
    predictions = torch.argmax(logits)

    print(text + " - " + id2label[predictions.tolist()])

# Output:
```

```
# Untrained model predictions:
# ----------------------------
# It was good. - Negative
# Not a fan, don't recommed. - Negative
# Better than the first one. - Negative
# This is not worth watching even once. - Negative
# This one is a pass. - Negative
```

As expected, the model performance is equivalent to random guessing. Let's see how we can improve this with fine-tuning.

Fine-tuning with LoRA

To use LoRA for fine-tuning, we first need a config file. This sets all the parameters for the LoRA algorithm. See comments in the code block for more details.

```
peft_config = LoraConfig(task_type="SEQ_CLS", # sequence classification
                         r=4, # intrinsic rank of trainable weight matrix
                         lora_alpha=32, # this is like a learning rate
                         lora_dropout=0.01, # probablity of dropout
                         target_modules = ['q_lin']) # we apply lora to query lay
```

We can then create a new version of our model that can be trained via PEFT. Notice that the scale of trainable parameters was reduced by about 100x.

```
model = get_peft_model(model, peft_config)
model.print_trainable_parameters()

# trainable params: 1,221,124 || all params: 67,584,004 || trainable%: 1.8068239
```

Next, we define hyperparameters for model training.

```
# hyperparameters
lr = 1e-3 # size of optimization step
batch_size = 4 # number of examples processed per optimziation step
num_epochs = 10 # number of times model runs through training data

# define training arguments
training_args = TrainingArguments(
    output_dir= model_checkpoint + "-lora-text-classification",
    learning_rate=lr,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=num_epochs,
    weight_decay=0.01,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
)
```

Finally, we create a trainer() object and fine-tune the model!

```
# creater trainer object
trainer = Trainer(
    model=model, # our peft model
    args=training_args, # hyperparameters
    train_dataset=tokenized_dataset["train"], # training data
    eval_dataset=tokenized_dataset["validation"], # validation data
    tokenizer=tokenizer, # define tokenizer
    data_collator=data_collator, # this will dynamically pad examples in each ba
    compute_metrics=compute_metrics, # evaluates model using compute_metrics() f
)

# train model
trainer.train()
```

The above code will generate the following table of metrics during training.

Epoch	Training Loss	Validation Loss	Accuracy
1	No log	0.423429	{'accuracy': 0.876}
2	0.412400	0.551401	{'accuracy': 0.878}
3	0.412400	0.593060	{'accuracy': 0.899}
4	0.211600	0.640572	{'accuracy': 0.894}
5	0.211600	0.839775	{'accuracy': 0.891}
6	0.064300	0.930830	{'accuracy': 0.887}
7	0.064300	0.988515	{'accuracy': 0.889}
8	0.020000	1.007572	{'accuracy': 0.884}
9	0.020000	1.040836	{'accuracy': 0.888}
10	0.009500	1.034907	{'accuracy': 0.896}

Model training metrics. Image by author.

Trained model performance

To see how the model performance has improved, let's apply it to the same 5 examples from before.

```
model.to('mps') # moving to mps for Mac (can alternatively do 'cpu')

print("Trained model predictions:")
print("--------------------------")
for text in text_list:
    inputs = tokenizer.encode(text, return_tensors="pt").to("mps") # moving to m

    logits = model(inputs).logits
    predictions = torch.max(logits,1).indices
```

```
    print(text + " - " + id2label[predictions.tolist()[0]])

# Output:
# Trained model predictions:
# ----------------------------
# It was good. - Positive
# Not a fan, don't recommed. - Negative
# Better than the first one. - Positive
# This is not worth watching even once. - Negative
# This one is a pass. - Positive # this one is tricky
```

The fine-tuned model improved significantly from its prior random guessing, correctly classifying all but one of the examples in the above code. This aligns with the ~90% accuracy metric we saw during training.

Links: Code Repo | Model | Dataset

Conclusions

While fine-tuning an existing model requires more computational resources and technical expertise than using one out-of-the-box, (smaller) fine-tuned models can outperform (larger) pre-trained base models for a particular use case, even when employing clever prompt engineering strategies. Furthermore, with all the open-source LLM resources available, it's never been easier to fine-tune a model for a custom application.

The next (and final) article of this series will go one step beyond model fine-tuning and discuss how to train a language model from scratch.

👉 **More on LLMs:** Introduction | OpenAI API | Hugging Face Transformers | Prompt Engineering | Build an LLM

How to Build an LLM from Scratch

Data Curation, Transformers, Training at Scale, and Model Evaluation

towardsdatascience.com

Resources

Connect: My website | Book a call | Ask me anything

Socials: YouTube 📹 | LinkedIn | Twitter

Support: Buy me a coffee ☕

The Data Entrepreneurs

A community for entrepreneurs in the data space. 👉 Join the Discord!

medium.com

[1] Deeplearning.ai Finetuning Large Langauge Models Short Course: https://www.deeplearning.ai/short-courses/finetuning-large-language-models/

[2] arXiv:2005.14165 **[cs.CL]** (GPT-3 Paper)

[3] arXiv:2303.18223 **[cs.CL]** (Survey of LLMs)

[4] arXiv:2203.02155 **[cs.CL]** (InstructGPT paper)

[5] 🤗 PEFT: Parameter-Efficient Fine-Tuning of Billion-Scale Models on Low-Resource Hardware: https://huggingface.co/blog/peft

[6] arXiv:2106.09685 **[cs.CL]** (LoRA paper)

[7] Original dataset source — Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA. Association for Computational Linguistics.

Large Language Models | AI | Fine Tuning | Machine Learning | Getting Started

Written by Shaw Talebi

Edit profile