**[Beginners’ Guide to Finetuning Large Language Models (LLMs)](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/)**

[SHIVANSH KAUSHAL](https://www.analyticsvidhya.com/blog/author/shivansh/) — Published On August 29, 2023

[Beginner](https://www.analyticsvidhya.com/blog/category/beginner/) [Generative AI](https://www.analyticsvidhya.com/blog/category/generative-ai/) [Large Language Models](https://www.analyticsvidhya.com/blog/category/large-language-models/) [LLMs](https://www.analyticsvidhya.com/blog/category/llms/) [NLP](https://www.analyticsvidhya.com/blog/category/nlp/)

Introduction

Embark on a journey through the evolution of artificial intelligence and the astounding strides made in [Natural Language Processing](https://www.analyticsvidhya.com/blog/2017/01/ultimate-guide-to-understand-implement-natural-language-processing-codes-in-python/) (NLP). In a mere blink, AI has surged, shaping our world. The seismic impact of finetuning large language models has utterly transformed NLP, revolutionizing our technological interactions. Rewind to 2017, a pivotal moment marked by ‘Attention is all you need,’ birthing the groundbreaking ‘Transformer’ architecture. This architecture now forms the cornerstone of NLP, an irreplaceable ingredient in every Large Language Model recipe – including the renowned ChatGPT.

Imagine generating coherent, context-rich text effortlessly – that’s the magic of models like GPT-3. Powerhouses for chatbots, translations, and content generation, their brilliance stems from architecture and the intricate dance of pretraining and fine-tuning. Our upcoming article delves into this symphony, uncovering the artistry behind leveraging Large Language Models for tasks, wielding the dynamic duet of pre-training and fine-tuning to masterful effect. Join us in demystifying these transformative techniques!

Learning Objectives

Understand the different ways to build LLM applications.

Learn techniques like feature extraction, layers finetuning, and adapter methods.

Finetune LLM on a downstream task using the Huggingface transformers library.

Table of contents

1. [Introduction](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/#Introduction)
2. [Getting Started with LLMs](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/#What_are_LLMs?)
3. [Training LLMs from Scratch](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/#h-training-llms-from-scratch)
4. [Finetuning Large Language Models](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/#h-finetuning-large-language-models)
5. [Prompting](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/#h-prompting)
6. [Understand Different Finetuning Techniques](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/#Finetuning_technique)
7. [Feature Extraction](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/#h-feature-extraction)
8. [Full Model Finetuning](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/#h-full-model-finetuning)
9. [Adapter-Based Finetuning](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/#h-adapter-based-finetuning)
10. [Conclusion](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/#h-conclusion)
11. [Frequently Asked Questions](https://www.analyticsvidhya.com/blog/2023/08/finetuning-large-language-models-llms/#h-frequently-asked-questions)

Getting Started with LLMs

LLMs stands for Large Language Models. LLMs are deep learning models designed to understand the meaning of human-like text and perform various tasks such as sentiment analysis, language modeling(next-word prediction), text generation, text summarization, and much more. They are trained on a huge amount of text data.

We use applications based on these LLMs daily without even realizing it. Google uses BERT(Bidirectional Encoder Representations for Transformers) for various applications such as query completion, understanding the context of queries, outputting more relevant and accurate search results, language translation, and more.These models are built upon deep learning techniques, profound neural networks, and advanced techniques such as self-attention. They are trained on vast amounts of text data to learn the language’s patterns, structures, and semantics.

Since these models are trained on extensive datasets, it takes a lot of time and resources to train them, and it does not make sense to train them from scratch.  
There are techniques by which we can directly use these models for a specific task. So let’s discuss them in detail.

Overview of Different Ways to Build LLM Applications

We often see exciting LLM applications in a day to day life. Are you curious to know how to build LLM applications? Here are the 3 ways to build LLM applications:

* Training LLMs from Scratch
* Finetuning Large Language Models
* Prompting

Training LLMs from Scratch

People often get confused between these 2 terminologies: training and finetuning LLMs. Both of these techniques work in a similar way i.e., change the model parameters, but the training objectives are different.

Training LLMs from Scratch is also known as pretraining. Pretraining is the technique in which a large language model is trained on a vast amount of unlabeled text. But the question is, ‘How can we train a model on unlabeled data and then expect the model to predict the data accurately?’. Here comes the concept of ‘Self-Supervised Learning’. In self-supervised learning, a model masks a word and tries to predict the next word with the help of the preceding words. For, e.g., Suppose we have a sentence: ‘I am a data scientist’.

The model can create its own labeled data from this sentence like:

|  |  |
| --- | --- |
| Text | Label |
| I | am |
| I am | a |
| I am a | data |
| I am a Data | Scientist |

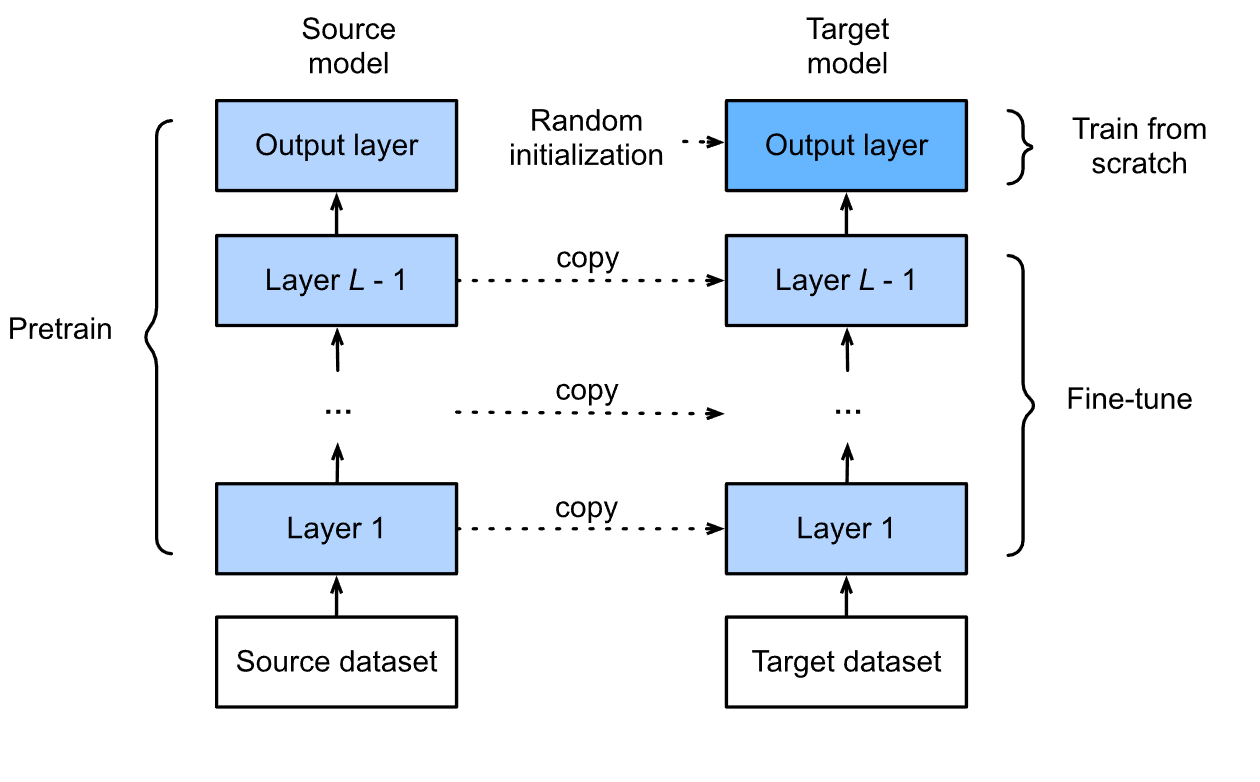
This is known as the next work prediction, done by an MLM (Masked Language Model). BERT, a masked language model, uses this technique to predict the masked word. We can think of MLM as a `fill in the blank` concept, in which the model predicts what word can fit in the blank.

There are different ways to predict the next word, but for this article, we only talk about BERT, the MLM. BERT can look at both the preceding and the succeeding words to understand the context of the sentence and predict the masked word.

So, as a high-level overview of pre-training, it is just a technique in which the model learns to predict the next word in the text.

Finetuning Large Language Models

Finetuning is tweaking the model’s parameters to make it suitable for performing a specific task. After the model is pre-trained, it is then fine-tuned or in simple words, trained to perform a specific task such as sentiment analysis, text generation, finding document similarity, etc. We do not have to train the model again on a large text; rather, we use the trained model to perform a task we want to perform. We will discuss how to finetune a Large Language Model in detail later in this article.



Prompting

Prompting is the easiest of all the 3 techniques but a bit tricky. It involves giving the model a context(Prompt) based on which the model performs tasks. Think of it as teaching a child a chapter from their book in detail, being very discrete about the explanation, and then asking them to solve the problem related to that chapter.

In context to LLM, take, for example, ChatGPT; we set a context and ask the model to follow the instructions to solve the problem given.

Suppose I want ChatGPT to ask me some interview questions on Transformers only. For a better experience and accurate output, you need to set a proper context and give a detailed task description.

Example: I am a Data Scientist with two years of experience and am currently preparing for a job interview at so and so company. I love problem-solving, and currently working with state-of-the-art NLP models. I am up to date with the latest trends and technologies. Ask me very tough questions on the Transformer model that the interviewer of this company can ask based on the company’s previous experience. Ask me ten questions and also give the answers to the questions.

The more detailed and specific you prompt, the better the results. The most fun part Is that you can generate the prompt from the model itself and then add a personal touch or the information needed.

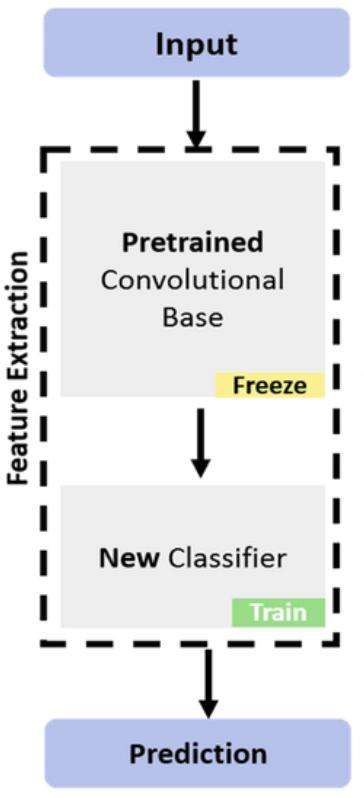
Understand Different Finetuning Techniques

There are different ways to finetune a model conventionally, and the different approaches depend on the specific problem you want to solve.  
Let’s discuss the techniques to fine-tune a model.

There are 3 ways of conventionally finetuning an LLM.

Feature Extraction

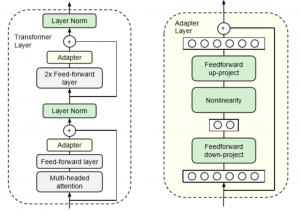
People use this technique to extract features from a given text, but why do we want to extract embeddings from a given text? The answer is straightforward. Because computers do not comprehend text, there needs to be a representation of the text that we can use to carry out various tasks. Once we extract the embeddings, they are capable of performing tasks like sentiment analysis, identifying document similarity, and more. In feature extraction, we lock the backbone layers of the model, meaning we do not update the parameters of those layers; only the parameters of the classifier layers get updated. The classifier layers involve the fully connected layers.



Full Model Finetuning

As the name suggests, we train each model layer on the custom dataset for a specific number of epochs in this technique. We adjust the parameters of all the layers in the model according to the new custom dataset. This can improve the model’s accuracy on the data and the specific task we want to perform. It is computationally expensive and takes a lot of time for the model to train, considering there are billions of parameters in the finetuning Large Language Models.

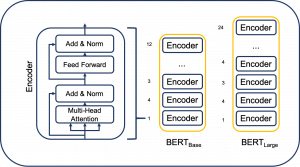
Adapter-Based Finetuning



Adapter-based finetuning is a comparatively new concept in which an additional randomly initialized layer or a module is added to the network and then trained for a specific task. In this technique, the model’s parameters are left undisturbed, or we can say that the model’s parameters are not changed or tuned. Rather, the adapter layer parameters are trained. This technique helps in tuning the model in a computationally efficient manner.

Implementation: Finetuning BERT on a Downstream Task

Now that we know the finetuning techniques let’s perform sentiment analysis on the IMDB movie reviews using BERT. BERT is a large language model that combines transformer layers and is encoder-only. Google developed it and has proven to perform very well on various tasks. BERT comes in different sizes and variants like BERT-base-uncased, BERT Large, RoBERTa, LegalBERT, and many more.



BERT Model to Perform Sentiment Analysis

Let’s use the BERT model to perform sentiment analysis on IMDB movie reviews. For free availability of GPU, it is recommended to use Google Colab. Let us start the training by loading some important libraries.

Since BERT(Bidirectional Encoder Representations for Encoders) is based on Transformers, the first step would be to install transformers in our environment.

!pip install transformers

Let’s load some libraries that will help us to load the data as required by the BERT model, tokenize the loaded data, load the model we will use for classification, perform train-test-split, load our CSV file, and some more functions.

Import pandas as pd

import numpy as np

import os

from sklearn.model\_selection import train\_test\_split

import torch

import torch.nn as nn

from transformers import BertTokenizer, BertModel

For faster computation, we have to change the device from CPU to GPU

device = torch.device(“cuda”)

The next step would be to load our dataset and look at the first 5 records in the dataset.

Df = pd.read\_csv(‘/content/drive/MyDrive/movie.csv’)

df.head()

We will split our dataset into training and validation sets. You can also split the data into train, validation, and test sets, but for the sake of simplicity, I am just splitting the dataset into training and validation.

X\_train, x\_val, y\_train, y\_val = train\_test\_split(df.text, df.label, random\_state = 42, test\_size = 0.2, stratify = df.label)

Import and Load the BERT Model

Let us import and load the BERT model and tokenizer.

From transformers.models.bert.modeling\_bert import BertForSequenceClassification

# import BERT-base pretrained model

BERT = BertModel.from\_pretrained(‘bert-base-uncased’)

# Load the BERT tokenizer

tokenizer = BertTokenizer.from\_pretrained(‘bert-base-uncased’)

We will use the tokenizer to convert the text into tokens with a maximum length of 250 and padding and truncation when required.

Train\_tokens = tokenizer.batch\_encode\_plus(x\_train.tolist(), max\_length = 250, pad\_to\_max\_length=True, truncation=True)

val\_tokens = tokenizer.batch\_encode\_plus(x\_val.tolist(), max\_length = 250, pad\_to\_max\_length=True, truncation=True)

The tokenizer returns a dictionary with three key-value pairs containing the input\_ids, which are the tokens relating to a particular word; token\_type\_ids, which is a list of integers that distinguish between different segments or parts of the input. And attention\_mask which indicates which token to attend to.

Converting these values into tensors

train\_ids = torch.tensor(train\_tokens[‘input\_ids’])

train\_masks = torch.tensor(train\_tokens[‘attention\_mask’])

train\_label = torch.tensor(y\_train.tolist())

val\_ids = torch.tensor(val\_tokens[‘input\_ids’])

val\_masks = torch.tensor(val\_tokens[‘attention\_mask’])

val\_label = torch.tensor(y\_val.tolist())

Loading TensorDataset and DataLoaders to preprocess the data further and make it suitable for the model.

From torch.utils.data import TensorDataset, DataLoader

train\_data = TensorDataset(train\_ids, train\_masks, train\_label)

val\_data = TensorDataset(val\_ids, val\_masks, val\_label)

train\_loader = DataLoader(train\_data, batch\_size = 32, shuffle = True)

val\_loader = DataLoader(val\_data, batch\_size = 32, shuffle = True)

Our task is to freeze the parameters of BERT using our classifier and then fine-tune those layers on our custom dataset. So, let’s freeze the parameters of the model.   
for param in BERT.parameters():  
param.requires\_grad = False

Now, we will have to define the forward and the backward pass for the layers that we have added. The BERT model will act as a feature extractor while we will have to define the forward and backward passes for classification explicitly.

Class Model(nn.Module):

def \_\_init\_\_(self, bert):

super(Model, self).\_\_init\_\_()

self.bert = bert

self.dropout = nn.Dropout(0.1)

self.relu = nn.ReLU()

self.fc1 = nn.Linear(768, 512)

self.fc2 = nn.Linear(512, 2)

self.softmax = nn.LogSoftmax(dim=1)

def forward(self, sent\_id, mask):

# Pass the inputs to the model

outputs = self.bert(sent\_id, mask)

cls\_hs = outputs.last\_hidden\_state[:, 0, :]

x = self.fc1(cls\_hs)

x = self.relu(x)

x = self.dropout(x)

x = self.fc2(x)

x = self.softmax(x)

return x

Let’s move the model to GPU

model = Model(BERT)

# push the model to GPU

model = model.to(device)

Defining the Optimizer

model = Model(BERT)

# push the model to GPU

model = model.to(device)

Till now, we have preprocessed the dataset and defined our model. Now is the time to train the model. We have to write a code to train and evaluate the model.  
The train function:

def train():

model.train()

total\_loss, total\_accuracy = 0, 0

total\_preds = []

for step, batch in enumerate(train\_loader):

# Move batch to GPU if available

batch = [item.to(device) for item in batch]

sent\_id, mask, labels = batch

# Clear previously calculated gradients

optimizer.zero\_grad()

# Get model predictions for the current batch

preds = model(sent\_id, mask)

# Calculate the loss between predictions and labels

loss\_function = nn.CrossEntropyLoss()

loss = loss\_function(preds, labels)

# Add to the total loss

total\_loss += loss.item()

# Backward pass and gradient update

loss.backward()

optimizer.step()

# Move predictions to CPU and convert to numpy array

preds = preds.detach().cpu().numpy()

# Append the model predictions

total\_preds.append(preds)

# Compute the average loss

avg\_loss = total\_loss / len(train\_loader)

# Concatenate the predictions

total\_preds = np.concatenate(total\_preds, axis=0)

# Return the average loss and predictions

return avg\_loss, total\_preds

The Evaluation Function

def evaluate():

model.eval()

total\_loss, total\_accuracy = 0, 0

total\_preds = []

for step, batch in enumerate(val\_loader):

# Move batch to GPU if available

batch = [item.to(device) for item in batch]

sent\_id, mask, labels = batch

# Clear previously calculated gradients

optimizer.zero\_grad()

# Get model predictions for the current batch

preds = model(sent\_id, mask)

# Calculate the loss between predictions and labels

loss\_function = nn.CrossEntropyLoss()

loss = loss\_function(preds, labels)

# Add to the total loss

total\_loss += loss.item()

# Backward pass and gradient update

loss.backward()

optimizer.step()

# Move predictions to CPU and convert to numpy array

preds = preds.detach().cpu().numpy()

# Append the model predictions

total\_preds.append(preds)

# Compute the average loss

avg\_loss = total\_loss / len(val\_loader)

# Concatenate the predictions

total\_preds = np.concatenate(total\_preds, axis=0)

# Return the average loss and predictions

return avg\_loss, total\_preds

We will now use these functions to train the model:

# set initial loss to infinite

best\_valid\_loss = float(‘inf’)

#defining epochs

epochs = 5

# empty lists to store training and validation loss of each epoch

train\_losses=[]

valid\_losses=[]

#for each epoch

for epoch in range(epochs):

print(‘\n Epoch {:} / {:}’.format(epoch + 1, epochs))

#train model

train\_loss, \_ = train()

#evaluate model

valid\_loss, \_ = evaluate()

#save the best model

if valid\_loss < best\_valid\_loss:

best\_valid\_loss = valid\_loss

torch.save(model.state\_dict(), ‘saved\_weights.pt’)

# append training and validation loss

train\_losses.append(train\_loss)

valid\_losses.append(valid\_loss)

print(f’\nTraining Loss: {train\_loss:.3f}’)

print(f’Validation Loss: {valid\_loss:.3f}’)

And there you have it. You can use your trained model to infer any data or text you choose.

Conclusion

This article explored the world of finetuning Large Language Models (LLMs) and their significant impact on natural language processing (NLP). Discuss the pretraining process, where LLMs are trained on large amounts of unlabeled text using self-supervised learning. We also delved into finetuning, which involves adapting a pre-trained model for specific tasks and prompting, where models are provided with context to generate relevant outputs. Additionally, we examined different finetuning techniques, such as feature extraction, full model finetuning, and adapter-based finetuning Large Language Models have revolutionized NLP and continue to drive advancements in various applications.

Frequently Asked Questions

Q1. How do Large Language Models (LLMs) like BERT understand the meaning of text without explicit labels?

A. LLMs employ self-supervised learning techniques like masked language modeling, where they predict the next word based on the context of surrounding words, effectively creating labeled data from unlabeled text.

Q2. What is the purpose of finetuning Large Language Models?

A. Finetuning allows LLMs to adapt to specific tasks by adjusting their parameters, making them suitable for sentiment analysis, text generation, or document similarity tasks. It builds upon the pre-trained knowledge of the model.

Q3. What is the significance of prompting in LLMs?

A. Prompting involves providing context or instructions to LLMs to generate relevant outputs. Users can guide the model to answer questions, generate text, or perform specific tasks based on the given context by setting a specific prompt.