#### Fine Tuning

1. when we fine tune a quantized model then is my fine tuned model requires quantization

Q. How do i know how many parameters do i have in my model

# PEFT LoRA for tuning LLM:

INT8 tuning of the OPT-6.7B model in Google Colab using PEFT LoRA +bitsandbytes (a lightweight wrapper around CUDA custom functions , in particular 8-bit optimizer , matrix multiplication (LLM.int8 ()), and quantization functions).

**PEFT: Parameter-Efficient Fine-Tuning of Billion-Scale Models on Low-Resource Hardware**

HuggingFace Library freezing most parameters of a pre-trained LLM

**LoRA: Low-Rank Adaption (a random projection to a smaller subspace)**

freezing pre-trained model weights and injects trainable rank decomposition matrices into each layer of Transformers

**Adapter Transformers: adapter-transformers extend transformers**

AdapterHub: A Framework for Adapting Transformers

Earlier Fine-tuning involves copying the weights from a pre-trained network and tuning them on the downstream task(a new set of weights for each task)

Multi-task learning requires simultaneous access to all task

Adapter yield “parameter-efficient tuning” for NLP . It permits training on task squentially

**Tuning** with adapter module involves adding a small number of new parameter to a model , which are trained on downstream task.

In **adapter-tuning** , the parameters of the original networks are frozen and there may be shared by many task.

## ****Why Finetuning LLMs?****

Large language models (LLMs) like BERT, GPT-3, GPT-4, LLaMA, and others are trained on a large corpus of data and have general knowledge. However, they may not perform as well on specific tasks without finetuning. For example, if you want to use a pretrained LLM for analyzing legal or medical documents, finetuning it on a corpus of legal documents can significantly improve the model's performance. (Interested readers can find an overview of different LLM finetuning methods in my previous article, [Finetuning Large Language Models: An Introduction To The Core Ideas And Approaches](https://magazine.sebastianraschka.com/p/finetuning-large-language-models).)

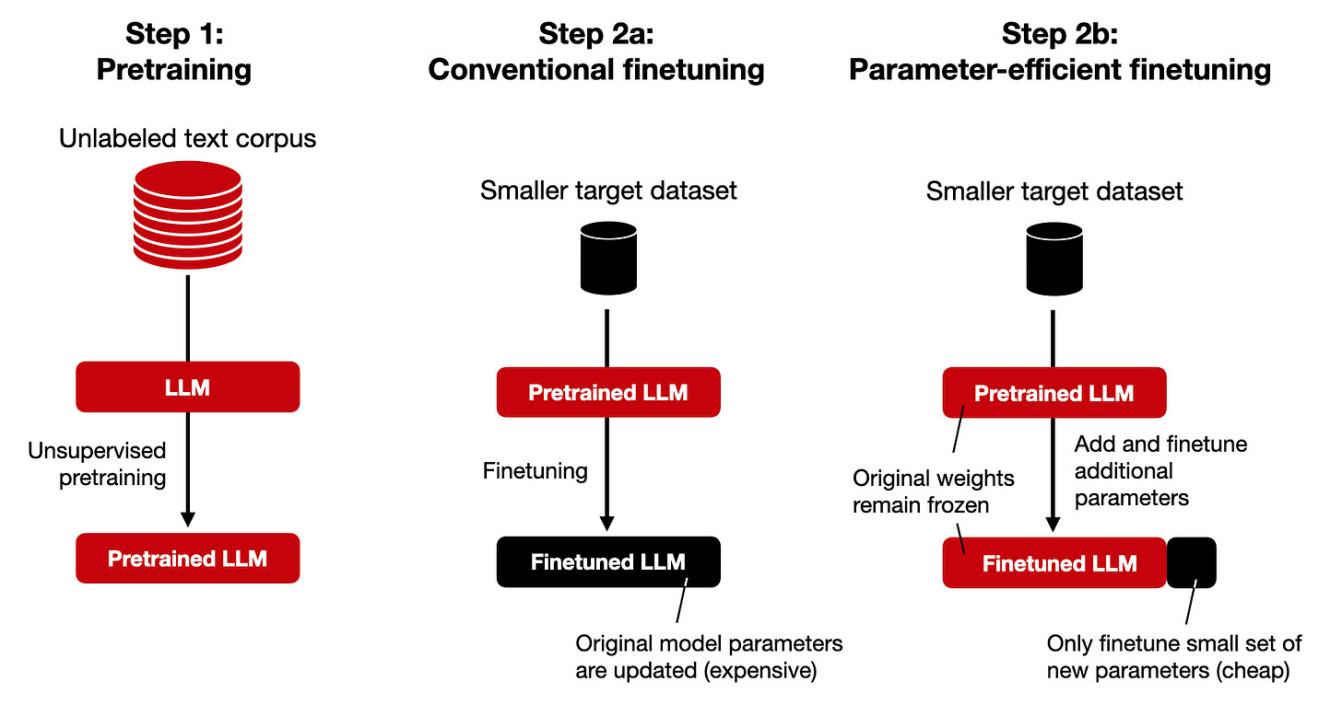
However, finetuning LLMs can be very expensive in terms of computational resources and time, which is why researchers started developing parameter-efficient finetuning methods.

## ****Parameter-Efficient Finetuning Methods****

As discussed in a previous article, many different types of parameter-efficient methods are out there. [In an earlier post, I wrote about prompt and prefix tuning](https://magazine.sebastianraschka.com/p/understanding-parameter-efficient). (Although the techniques are somewhat related, you don't need to know or read about prefix tuning before reading this article about adapters.)

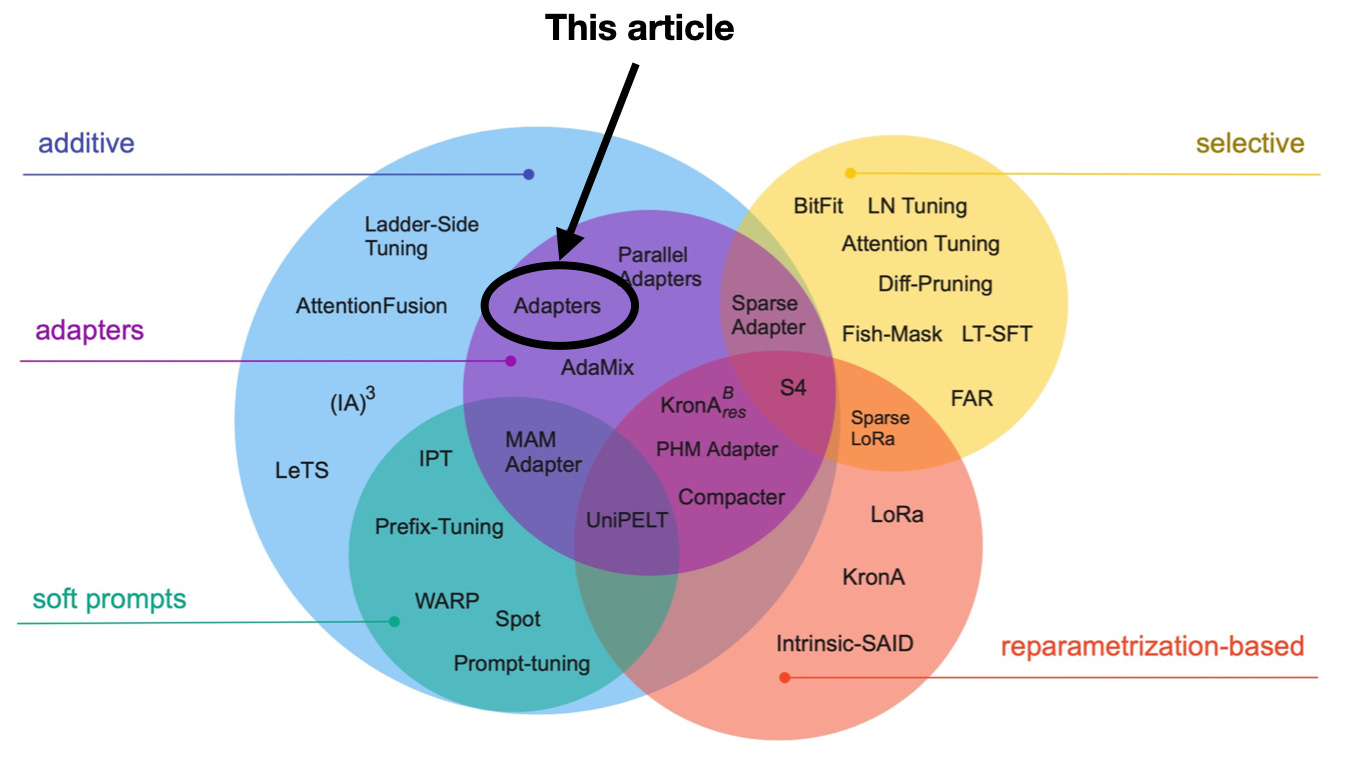
In a nutshell, prompt tuning (different from prompting) appends a tensor to the embedded inputs of a pretrained LLM. The tensor is then tuned to optimize a loss function for the finetuning task and data while all other parameters in the LLM remain frozen. For example, imagine an LLM pretrained on a general dataset to generate texts. Prompt (fine)tuning would entail taking this pretrained LLM, adding prompt tokens to the embedded inputs, and then finetuning the LLM to perform, for example, sentiment classification on a finetuning dataset.

The main idea behind prompt tuning, and parameter-efficient finetuning methods in general, is to add a small number of new parameters to a pretrained LLM and only finetune the newly added parameters to make the LLM perform better on (a) a target dataset (for example, a domain-specific dataset like medical or legal documents) and (b) a target task (for example, sentiment classification).

[](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https://substack-post-media.s3.amazonaws.com/public/images/d9c855ba-814f-4f95-9b1b-c97a46eb2f42_1646x880.png" \t "/home/shtlp_0015/Documents\\x/_blank)

The idea of parameter-efficient finetuning techniques (like prompt, prefix, and adapter tuning) is to add a small set of parameters to a pretrained LLM. Only the newly added parameters are finetuned while all the parameters of the pretrained LLM remain frozen.

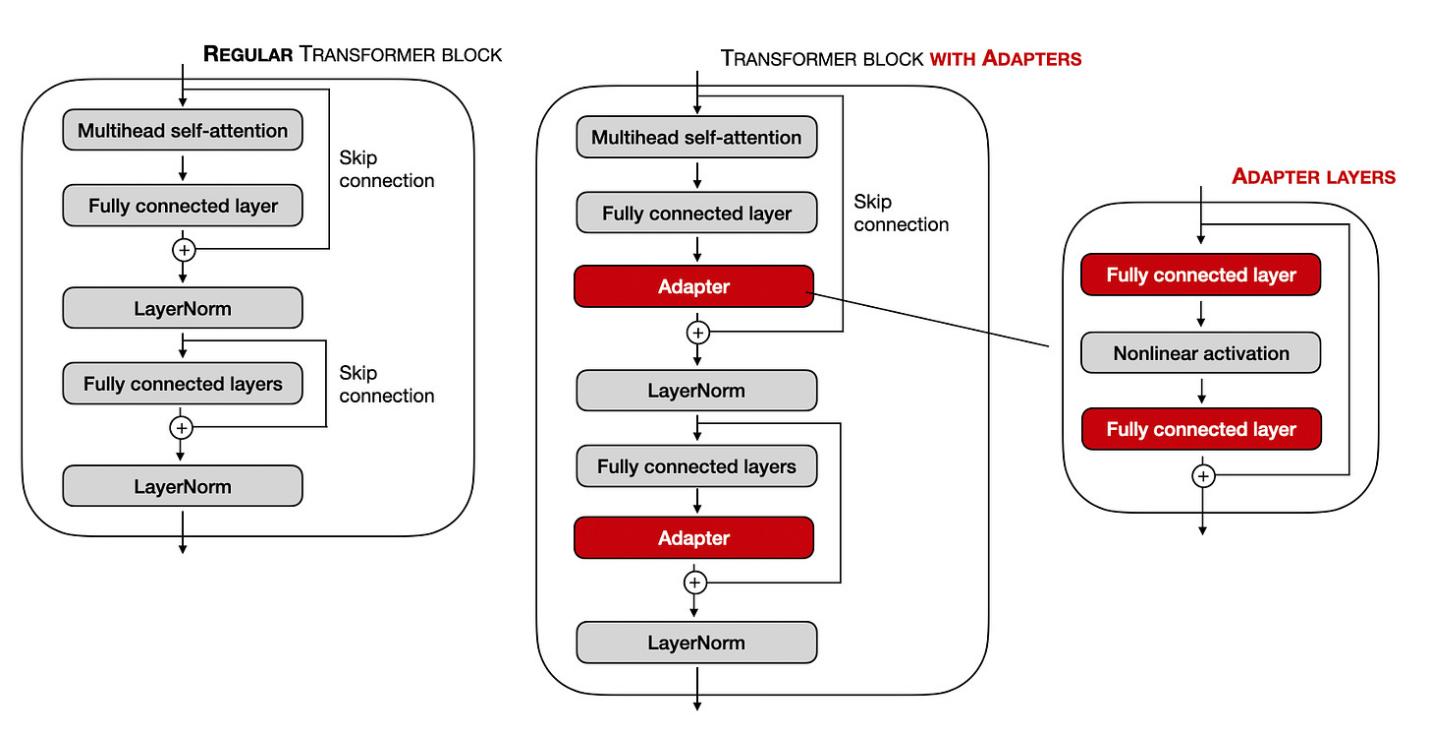
In this article, we are now discussing a related method called adapters, which is centered around the idea of adding tunable layers to the various transformer blocks of an LLM, as opposed to only modifying the input prompts.

[](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https://substack-post-media.s3.amazonaws.com/public/images/56034505-b52c-4305-b231-91a07a04221c_1360x762.png" \t "/home/shtlp_0015/Documents\\x/_blank)

Annotated figure from Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning, <https://arxiv.org/abs/2303.15647>.

## ****Adapters****

The original adapter method ([Houlsby et al.](https://arxiv.org/abs/1902.00751) 2019) is somewhat related to the aforementioned [prefix tuning method](https://magazine.sebastianraschka.com/p/understanding-parameter-efficient) as they also add additional parameters to each transformer block. However, while prefix tuning prepends tunable tensors to the embeddings, the adapter method adds adapter layers in two places, as illustrated in the figure below.

[](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https://substack-post-media.s3.amazonaws.com/public/images/1ef3bc00-ff4c-4c5a-bb6a-02cec0a76ab1_1698x874.png" \t "/home/shtlp_0015/Documents\\x/_blank)

Comparison of the regular transformer blocks used in various LLMs and a transformer block modified via the adapter layers.

And for readers who prefer (Python) pseudo-code, the adapter layer-modification can be written as follows:

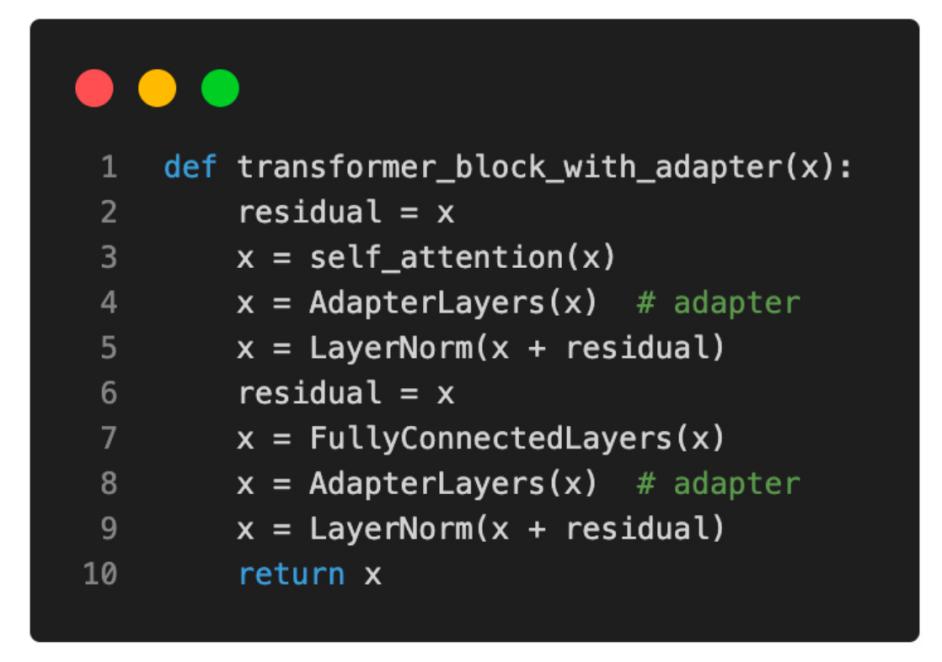
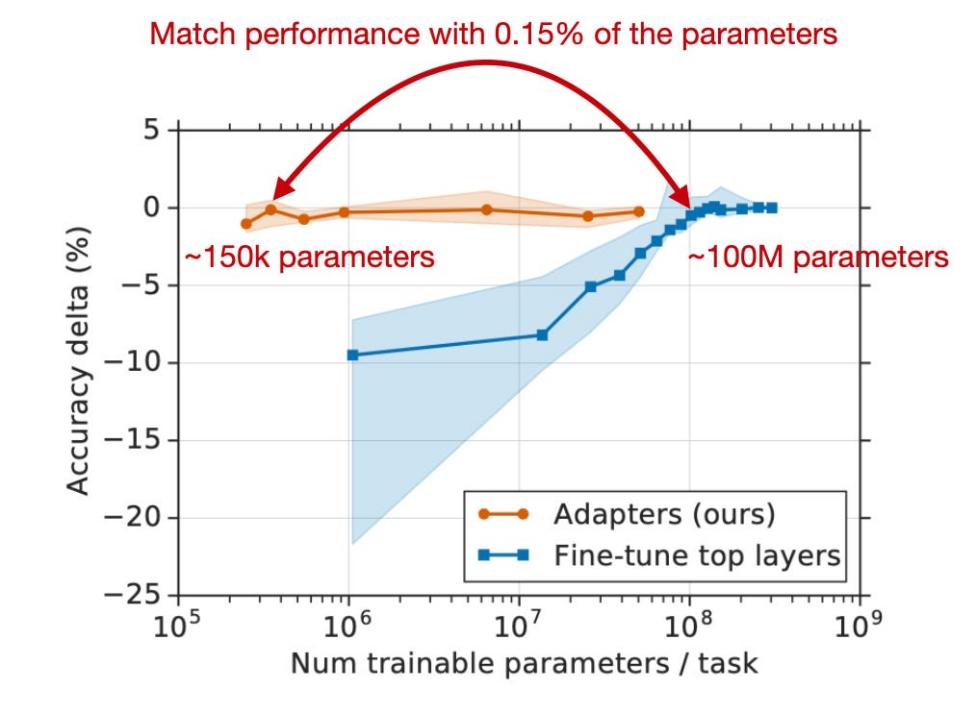
[](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https://substack-post-media.s3.amazonaws.com/public/images/712d0036-3ba7-4021-9c83-0dfb89a25cbf_1118x788.png" \t "/home/shtlp_0015/Documents\\x/_blank)

Illustration of a transformer block modified with adapter layers.

Note that the fully connected layers of the adapters are usually relatively small and have a bottleneck structure similar to autoencoders. Each adapter block's first fully connected layer projects the input down onto a low-dimensional representation. The second fully connected layer projects the input back into the input dimension. How is this parameter efficient? For example, assume the first fully connected layer projects a 1024-dimensional input down to 24 dimensions, and the second fully connected layer projects it back into 1024 dimensions. This means we introduced 1,024 x 24 + 24 x 1,024 = 49,152 weight parameters. In contrast, a single fully connected layer that reprojects a 1024-dimensional input into a 1,024-dimensional space would have 1,024 x 1024 = 1,048,576 parameters.

According to the original [adapter paper](https://arxiv.org/abs/1902.00751), a BERT model trained with the adapter method reaches a modeling performance comparable to a fully finetuned BERT model while only requiring the training of 3.6% of the parameters. Moreover, the researchers included a figure where they compared the adapter method to only finetung the output (top) layers of a BERT model and found that using adapters, it's possible to match the finetuning top-layer-finetuning performance with a much smaller number of parameters:

[](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https://substack-post-media.s3.amazonaws.com/public/images/75e6face-6d72-4ce6-888e-b17dbf8bd39b_1014x742.png" \t "/home/shtlp_0015/Documents\\x/_blank)

Annotated figure from the adapter paper, https://arxiv.org/abs/1902.00751.

## ****Conclusion****

Finetuning pre-trained large language models (LLMs) is an effective method to tailor these models to suit specific business requirements and align them with target domain data. This process involves adjusting the model parameters using a smaller dataset relevant to the desired domain, which enables the model to learn domain-specific knowledge and vocabulary.

However, as LLMs are "large," updating multiple layers in a transformer model can be very expensive, so researchers started developing parameter-efficient alternatives.

In this article, we discussed several parameter-efficient alternatives to the conventional LLM finetuning mechanism. In particular, we discussed how to insert and finetune additional adapter layers to improve the predictive performance of an LLM compared to training the original model parameters.

# Additional Experiments

## ****Additional Code Examples and Adapter Experiment****

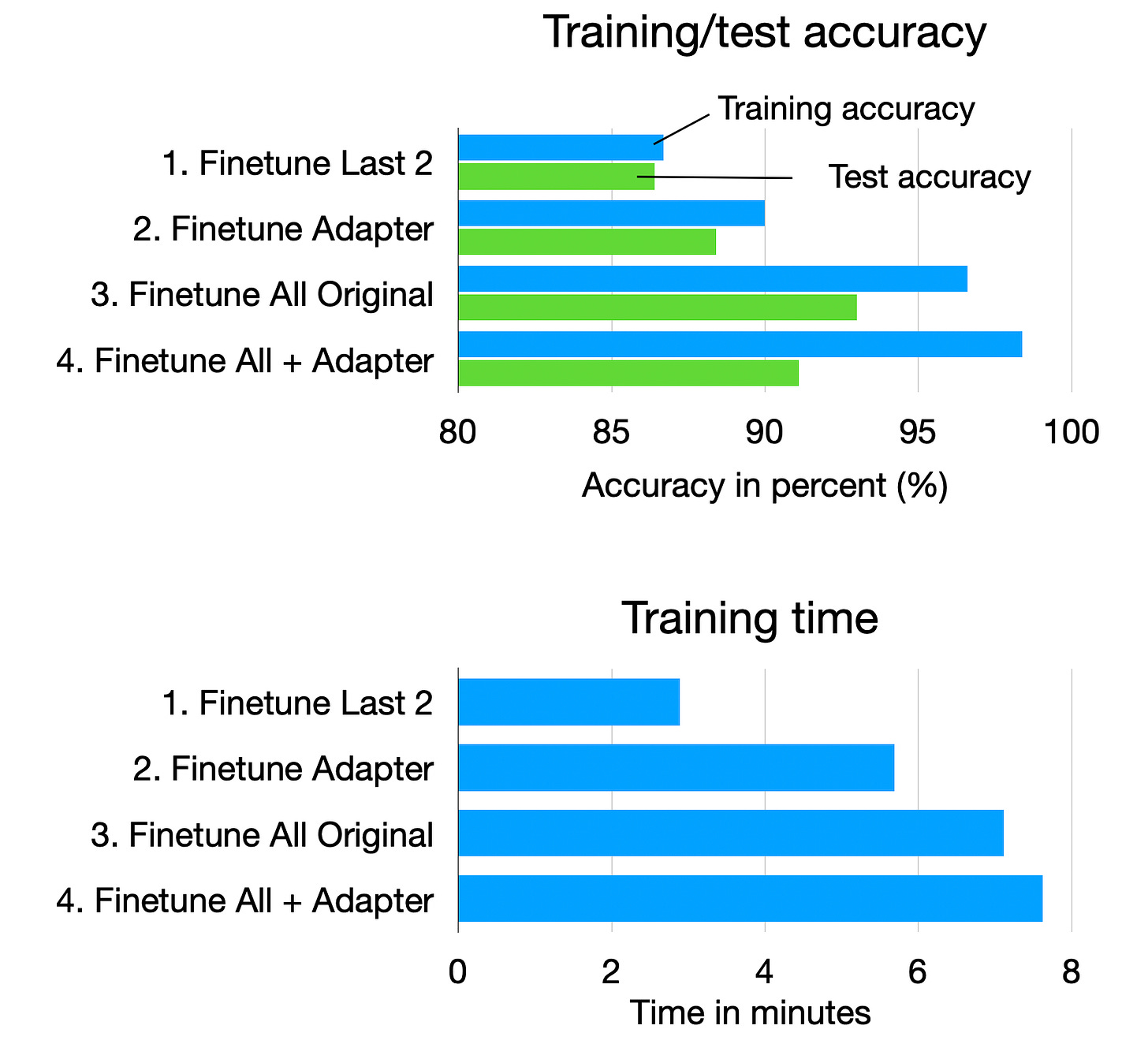
Below are additional experiments where I implemented the adapter method and ran a comparison to finetune a DistilBERT model for sentiment classification:

finetuning only the last two layers as a performance baseline;

inserting and finetuning adapter layers;

finetuning all layers of the original model;

inserting adapter layers and finetuning all layers as a control experiment.

[](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https://substack-post-media.s3.amazonaws.com/public/images/7c13d237-cc8e-422d-afd1-8dcbead182b9_1680x1528.png" \t "/home/shtlp_0015/Documents\\x/_blank)

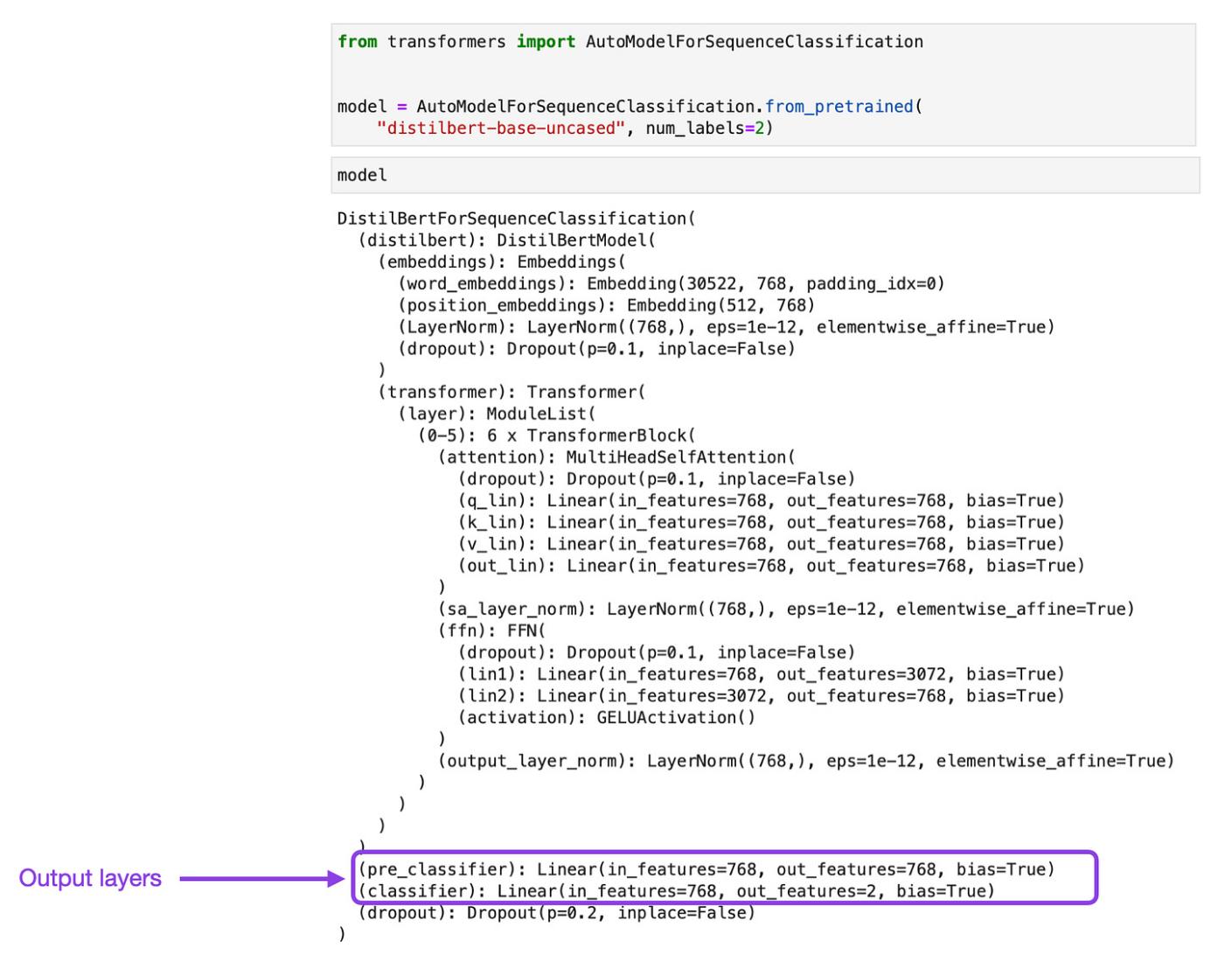
All code examples are available [here on GitHub](https://github.com/rasbt/LLM-finetuning-scripts/tree/main/adapter/distilbert-movie-review).

As a thanks to those who supported the newsletter in the previous months, I included a bonus section below discussing the code examples. Thanks again for your support!

### ****1. A Finetuning Baseline****

First, let's establish a performance baseline by only finetuning the last layers of a DistilBERT model on a movie review dataset. Here, we will only look at the relevant lines of code, omitting the non-finetuning specific code for brevity. However, as mentioned above, the full code examples are available [here](https://github.com/rasbt/LLM-finetuning-scripts/tree/main/adapter/distilbert-movie-review).

First, after loading the pretrained DistilBERT model, let's look at the architecture:

[](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https://substack-post-media.s3.amazonaws.com/public/images/29674c84-c421-4476-94b5-f515836f6663_1692x1320.png" \t "/home/shtlp_0015/Documents\\x/_blank)

For this performance baseline, we only finetune the last two layers, which comprise 592,130 parameters. The simplest way to do that is to freeze all parameters and then unfreeze the last two layers via the code below:

# Freeze all layers

for param in model.parameters():

param.requires\_grad = False

# Unfreeze the two output layers

for param in model.pre\_classifier.parameters():

param.requires\_grad = True

for param in model.classifier.parameters():

param.requires\_grad = True

Then, after training this model for 3 epochs, we get the following results:

Training time: 2.89 min

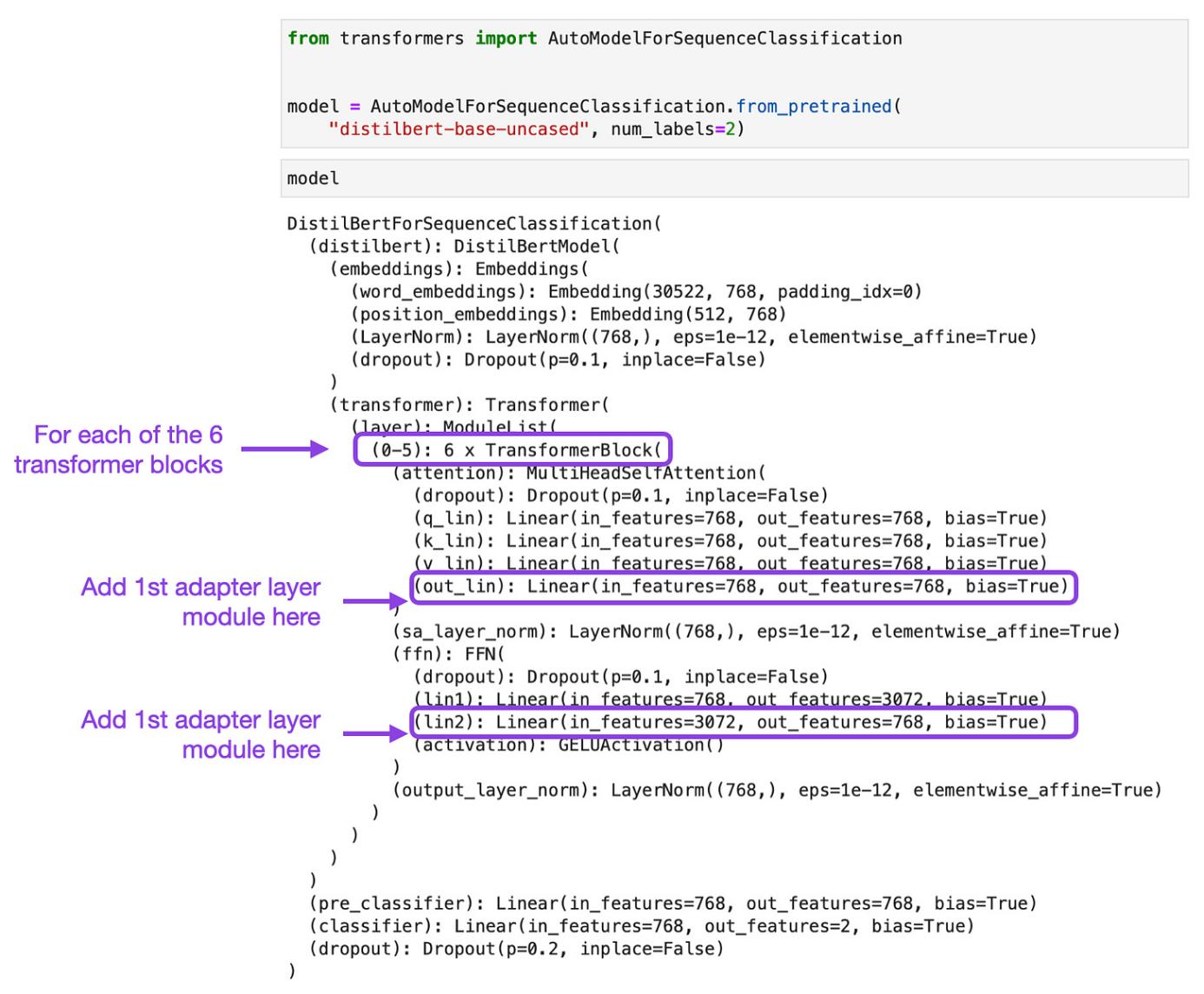
Training accuracy: 86.7%

Validation accuracy: 87.2%

Test accuracy: 86.4%

## ****2. Adding Adapter Layers****

Next, let's add the adapter layers to the model. Notice that DistilBERT has 6 transformer blocks. As discussed earlier, the adapter method inserts 2 adapter modules into each of the 6 transformer blocks, as shown in the figure below:

[](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https://substack-post-media.s3.amazonaws.com/public/images/b369c974-1881-4a74-81e6-d35260e3bc9e_1582x1306.png" \t "/home/shtlp_0015/Documents\\x/_blank)

Each adapter module consists of 2 fully connected layers with a nonlinear activation in-between. In code, we can define a make\_adapter function that creates such an adapter module as follows:

def make\_adapter(in\_dim, bottleneck\_dim, out\_dim):

adapter\_layers = torch.nn.Sequential(

torch.nn.Linear(in\_dim, bottleneck\_dim),

torch.nn.GELU(),

torch.nn.Linear(bottleneck\_dim, out\_dim),

)

return adapter\_layers

Then, we can use the make\_adapter function to insert the adapter layers into the 6 transformer blocks, as shown below:

total\_size = 0

bottleneck\_size = 32 # hyperparameter

for block\_idx in range(6):

###################################################

# insert 1st adapter layer into transformer block

###################################################

orig\_layer\_1 = model.distilbert.transformer.layer[block\_idx].attention.out\_lin

adapter\_layers\_1 = make\_adapter(

in\_dim=orig\_layer\_1.out\_features,

bottleneck\_dim=bottleneck\_size,

out\_dim=orig\_layer\_1.out\_features)

new\_1 = torch.nn.Sequential(orig\_layer\_1, \*adapter\_layers\_1)

model.distilbert.transformer.layer[block\_idx].attention.out\_lin = new\_1

total\_size += count\_parameters(adapter\_layers\_1)

###################################################

# insert 2nd adapter layer into transformer block

###################################################

orig\_layer\_2 = model.distilbert.transformer.layer[block\_idx].ffn.lin2

adapter\_layers\_2 = make\_adapter(

in\_dim=orig\_layer\_2.out\_features,

bottleneck\_dim=bottleneck\_size,

out\_dim=orig\_layer\_2.out\_features)

new\_2 = torch.nn.Sequential(orig\_layer\_2, \*adapter\_layers\_2)

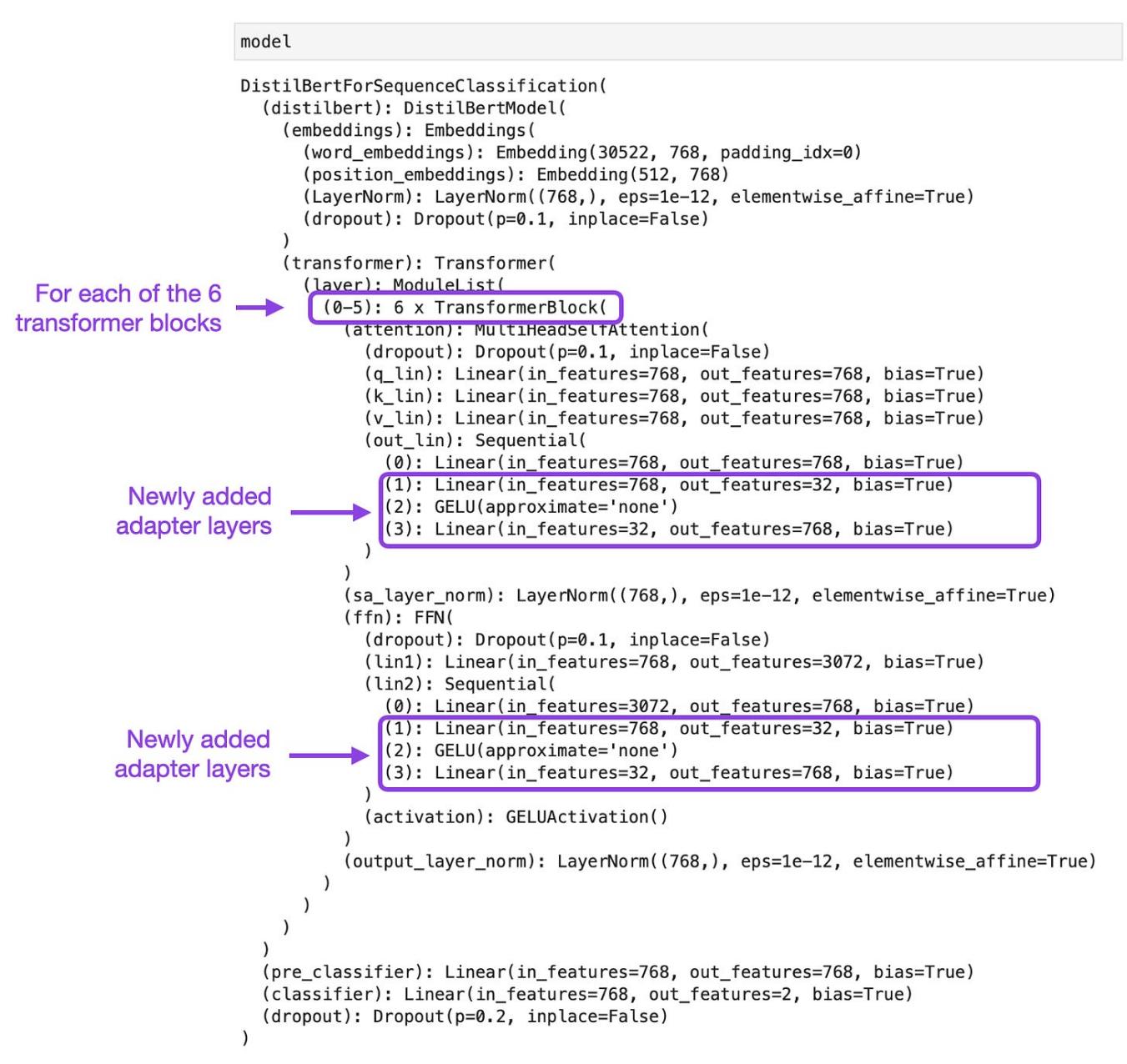
model.distilbert.transformer.layer[block\_idx].ffn.lin2 = new\_2

total\_size += count\_parameters(adapter\_layers\_2)

print("Number of adapter parameters added:", total\_size)

Number of adapter parameters added: 599,424

The modified DistilBERT architecture is shown in the figure below:

[](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https://substack-post-media.s3.amazonaws.com/public/images/2b4a7a9c-39c5-45fd-ab9e-ac9e4f18d670_1500x1418.png" \t "/home/shtlp_0015/Documents\\x/_blank)

Notice that using a bottleneck size of 32, we added 599,424 new parameters to the model. In comparison, the 2 fully connected layers we finetuned earlier have 592,130 parameters in total, which is approximately the same number of parameters to finetune. If we finetune this modified model, where all layers except the adapter layers are frozen, we get the following results:

Training time: 5.69 min

Training accuracy: 90.0%

Validation accuracy: 89.1%

Test accuracy: 88.4%

## ****3. Finetuning All Layers****

Now, for comparison, let's look at the results from finetuning all layers. For this, we are loading the DistilBERT model and training it as is (without freezing any layers).

from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from\_pretrained(

"distilbert-base-uncased", num\_labels=2)

def count\_parameters(model):

return sum(p.numel() for p in model.parameters() if p.requires\_grad)

num\_param = count\_parameters(model.pre\_classifier) + count\_parameters(model.classifier)

print("Parameters in last 2 layers:", num\_param)

66955010

The result from finetuning all 66.9 million parameters are as follows:

Training time: 7.12 min

Training accuracy: 96.6%

Validation accuracy: 92.9%

Test accuracy: 93.0%

## ****4. Inserting Adapter Layers and Finetuning All Layers****

Lastly, let's add a control experiment, where we train the model modified with the adapter layers in Section 2, but making all parameters trainable. That's 599,424 + 66,955,010 = 67,554,434 in total.

Training time: 7.62 min

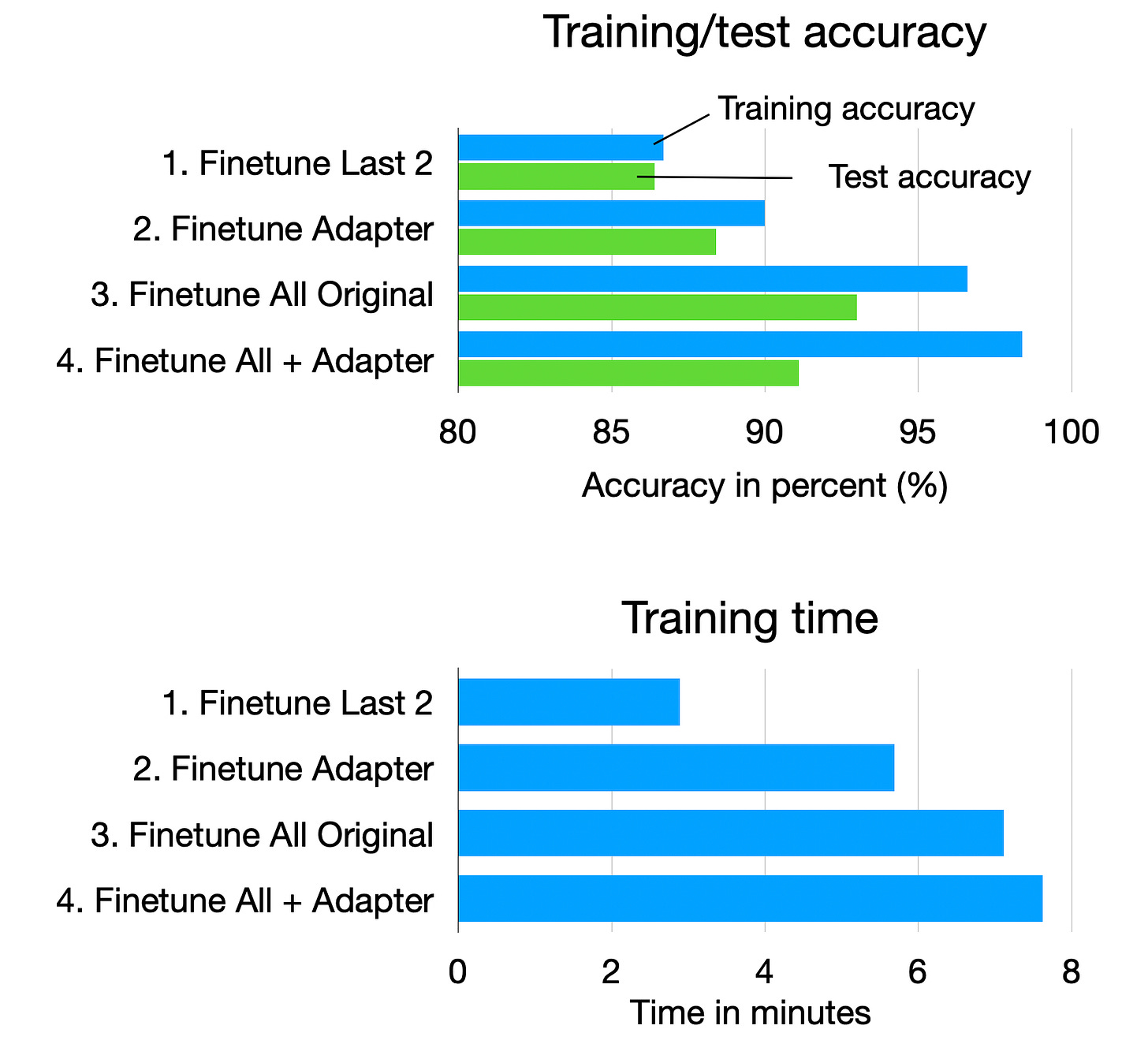
Training accuracy: 98.4%

Validation accuracy: 91.5%

Test accuracy: 91.1%

## ****Result Analysis and Summary****

Now that we gathered all the results via the experiments above, let's look at the summary plots below.

[](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https://substack-post-media.s3.amazonaws.com/public/images/7c13d237-cc8e-422d-afd1-8dcbead182b9_1680x1528.png" \t "/home/shtlp_0015/Documents\\x/_blank)

As we can see, finetuning the adapter layers outperforms finetuning only the last layers. This is a nice, positive, but not unexpected outcome, consistent with the adapter paper results.

However, finetuning the adapter layers (2) takes almost twice as finetuning the last two layers only (1). The number of trainable parameters in (1) and (2) is practically identical. However, the adapter layer model has several additional layers in the forward pass, which can explain the extra training time.

Now, looking at method (3), finetuning the whole network does still outperform the adapter method (3). Still, it is, of course, also computationally more expensive, which is expected since we have a substantially larger number of parameters (66 million versus 600 thousand).

Lastly, finetuning all layers plus adapter layers (4) performs better than only finetuning the adapter layers (2). However, (4) performs worse than finetuning all layers (3), which is surprising at first glance since (4) has 600,000 additional parameters compared to (3). If we look at the training set, we can see that method (4) overfits more, which is a possible explanation for why method (4) performs worse than (3) despite the additional parameters.

## **WHAT IS THE TRANSFORMER NEURAL NETWORK?**

The transformer neural network is a novel architecture that aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease. It was first proposed in the paper “Attention Is All You Need.”  and is now a state-of-the-art technique in the field of NLP.