5

Diving into Fine-Tuning through BERT

In *Chapter 3*, *Emergent vs Downstream Tasks: The Unseen Depths of Transformers*, we explored tasks through pretrained models that could perform efficiently. However, in some cases, a pretrained model will not produce the desired outputs. We can pretrain a model from scratch, as we will see in *Chapter 6*, *Pretraining a Transformer from Scratch through RoBERTa*. However, pretraining a model can require a large amount of machine, data, and human resources. The alternative can be fine-tuning a transformer model.

This chapter will dive into fine-tuning transformer models through a Hugging Face pretrained BERT model. By the end of the chapter, you should be able to fine-tune other Hugging Face models such as GPT, T5, RoBERTa, and more.

The chapter is divided into two parts. We will first go through the architecture of BERT models. Then, we will explore fine-tuning a transformer model through BERT.

We will first explore the architecture of **Bidirectional Encoder Representations from Transformers** (**BERT**). BERT only uses the blocks of the encoders of the Transformer in a novel way and does not use the decoder stack.

BERT added a new piece to the Transformer building kit: a bidirectional multi-head attention sub-layer. When we humans have problems understanding a sentence, we do not just look at the past words. BERT, like us, looks at all the words in a sentence at the same time.

Then, we will fine-tune a BERT model trained by a third party and uploaded to Hugging Face. This shows that transformers can be pretrained. A pretrained BERT, for example, can be fine-tuned on several NLP tasks. We will go through this fascinating experience of fine-tuning a transformer using Hugging Face modules, including building an interface with `ipywidgets` to interact with our trained model.

This chapter covers the following topics:

- BERT architecture
- The encoder stack of transformers and bidirectional attention
- Fine-tuning process for the acceptability judgment task
- Creating training data, labels, and BERT tokens
- Processing the training data with a BERT tokenizer and attention masks
- Splitting data into training and validation sets
- Setting up a Hugging Face BERT uncased base model
- Grouping parameters
- Setting hyperparameters
- Running a training loop
- Post-training evaluation with holdout
- Building a Python interface to interact with the model

With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples: https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter05.

You can also post a message in our Discord community (https://www.packt.link/Transformers) if you have any trouble running the code in this or any chapter.

Our first step will be to explore the background of the BERT model.

The architecture of BERT

In *Chapter 2, Getting Started with the Architecture of the Transformer Model*, we defined the building blocks of the architecture of the Original Transformer. Think of the Original Transformer as a model built with LEGO® bricks. The construction set contains bricks such as encoders, decoders, embedding layers, positional encoding methods, multi-head attention layers, masked multi-head attention layers, post-layer normalization, feed-forward sub-layers, and linear output layers.

The bricks come in various sizes and forms. As a result, you can spend hours building all sorts of models using the same building

kit! Some constructions will only require some of the bricks. Other structures will add a new piece, like when we obtain additional bricks for a model built using LEGO® components.

BERT introduces bidirectional attention to transformer models. Bidirectional attention requires many other changes to the Original Transformer model.

We will not go through the building blocks of transformers described in *Chapter 2, Getting Started with the Architecture of the Transformer Model*. You can consult *Chapter 2* at any time to review an aspect of the building blocks of transformers. Instead, this section will focus on the specific aspects of BERT models.

We will focus on the evolutions designed by *Devlin et al*. (2018), which describe the encoder stack. We will first go through the encoder stack and then the preparation of the pretraining input environment. Then, we will explain the two-step framework of BERT: pretraining and fine-tuning.

Let's first explore the encoder stack.

The encoder stack

The first building block we will take from the Original Transformer model is an encoder layer. The encoder layer, as described in *Chapter 2, Getting Started with the Architecture of the Transformer Model*, is shown in *Figure 5.1*:

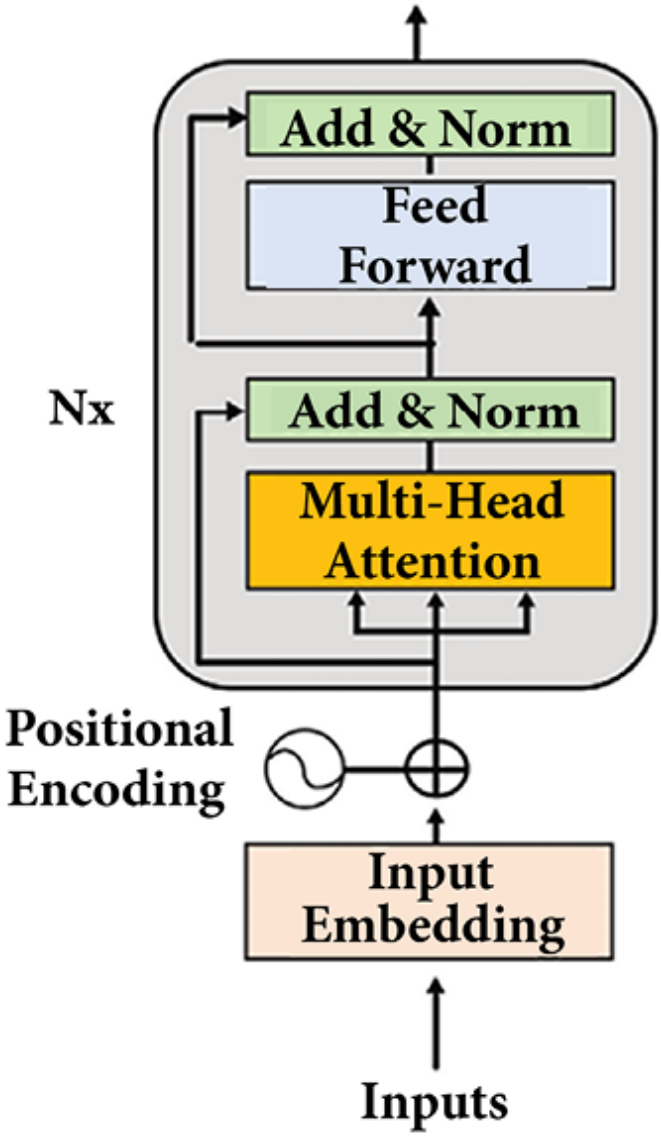

Figure 5.1: The encoder layer

The BERT model does not use decoder layers. A BERT model has an encoder stack but no decoder stacks. The BERT model uses **Masked Language Modeling** (**MLM**) in which some input tokens are hidden ("masked"), and the attention layers must learn to understand the context. The model will predict the hidden tokens, as we will see when we zoom into a BERT encoder layer in the following sections.

The Original Transformer contains a stack of $N = 6$ layers. The number of dimensions of the Original Transformer is $d_{\text{model}} = 512$. The number of attention heads of the Original Transformer is $A = 8$. The dimensions of the head of the Original Transformer are:

$$d_k = \frac{d_{model}}{A} = \frac{512}{8} = 64$$

BERT encoder layers are larger than the Original Transformer model.

Different sizes of BERT models can be built with the encoder layers:

- $\text{BERT}_{\text{BASE}}$, which contains a stack of $N = 12$ encoder layers. $d_{\text{model}} = 768$, which can also be expressed as $H = 768$, as in the BERT paper. A multi-head attention sub-layer contains $A = 12$ heads. The dimension of each head z_{A} remains 64 as in the Original Transformer model:

  $$d_k = \frac{d_{model}}{A} = \frac{512}{8} = 64$$

  The output of each multi-head attention sub-layer before concatenation will be the output of the 12 heads:

  $$output\_multi\text{-}head\_attention = \{z_0, z_1, z_2, \ldots, z_{11}\}$$

- $\text{BERT}_{\text{LARGE}}$, which contains a stack of $N = 24$ encoder layers. $d_{\text{model}} = 1024$. A multi-head attention sub-layer contains $A = 16$ heads. The dimension of each head z_{A} also remains 64 as in the Original Transformer model:

  $$d_k = \frac{d_{model}}{A} = \frac{1024}{16} = 64$$

The output of each multi-head attention sub-layer before concatenation will be the output of the 16 heads:

$$output\_multi\text{-}head\_attention=\{z_0, z_1, z_2,\ldots, z_{15}\}$$

The sizes of the models can be summed up as follows:

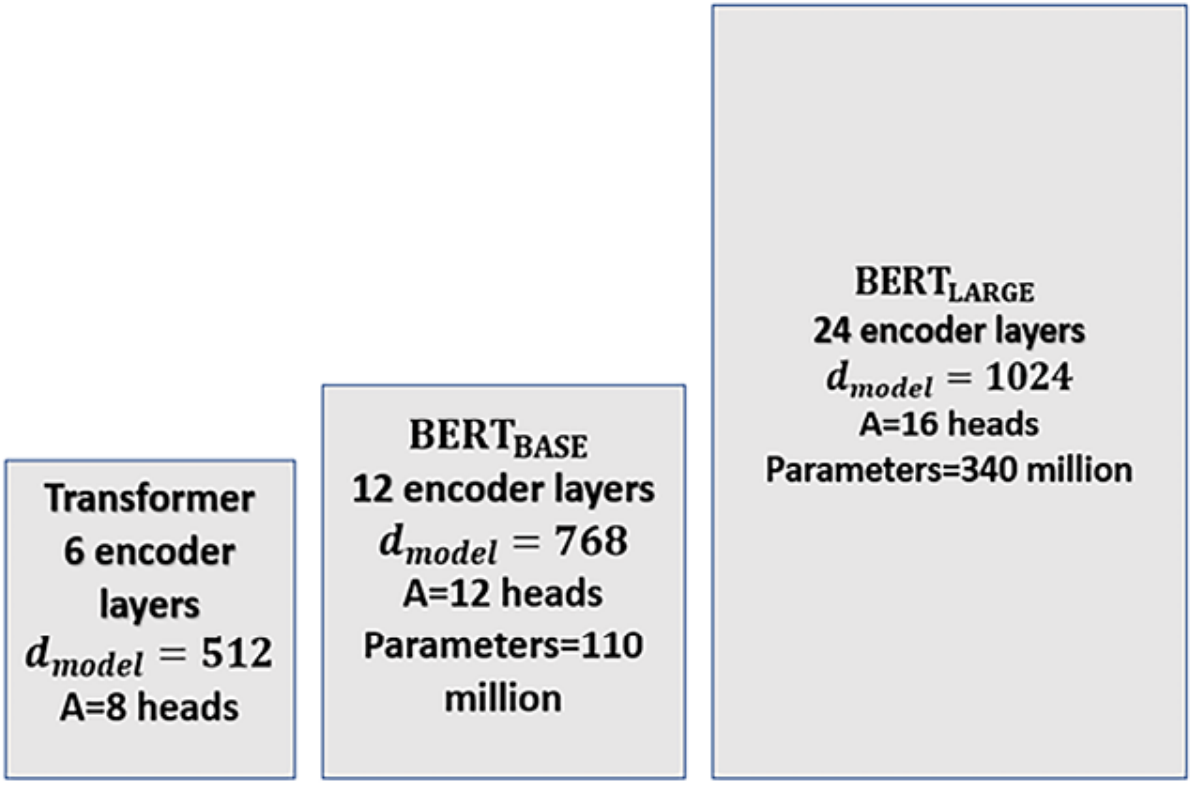

Figure 5.2: Transformer models

BERT models are not limited to these configurations, which illustrate the main aspects of BERT models. Numerous variations are possible.

Size and dimensions play an essential role in BERT-style pretraining. BERT models are like humans; they produce better results with more working memory (dimensions) and knowledge (data). Large

transformer models that learn large amounts of data will pretrain better for downstream NLP tasks.

Let's go to the first sub-layer and see the fundamental aspects of input embedding and positional encoding in a BERT model.

Preparing the pretraining input environment

The BERT model only has encoder layers and no decoder stack. Its architecture features a multi-head self-attention mechanism that allows each token to learn to understand all the surrounding tokens (masked or not). This bidirectional method enables BERT to understand the context on both sides of each token.

A masked multi-head attention layer masks some tokens randomly to force this system to learn contexts. For example, take the following sentence:

```
The cat sat on it because it was a nice rug.
```

If we have just reached the word `it`, the input of the encoder could be:

```
The cat sat on it<masked sequence>
```

The random mask can be anywhere in the sequence, not necessarily at the end. To know what `it` refers to, we need to see the whole sentence to reach the word `rug` and figure out that `it` was the rug.

The authors enhanced their bidirectional attention model, letting an attention head attend to *all* the words from left to right and right to left. In other words, the self-attention mask of an encoder could do the job without being hindered by the masked multi-head attention sub-layer of the decoder.

The model was trained with two tasks. The first method is **MLM**. The second method is **Next-Sentence Prediction** (**NSP**).

Let's start with MLM.

Masked language modeling

MLM does not require training a model with a sequence of visible words followed by a masked sequence to predict.

BERT introduces the *bidirectional* analysis of a sentence with a random mask on a word of the sentence.

It is important to note that BERT applies `WordPiece`, a subword segmentation tokenization method, to the inputs. It also uses learned positional encoding, not the sine-cosine approach.

A potential input sequence could be:

```
The cat sat on it because it was a nice rug.
```

The decoder could potentially mask the attention sequence after the model reaches the word `it`:

```
The cat sat on it <masked sequence>.
```

But the BERT encoder masks a random token to make a prediction, which makes it more powerful:

```
The cat sat on it [MASK] it was a nice rug.
```

The multi-attention sub-layer can now see the whole sequence, run the self-attention process, and predict the masked token.

The input tokens were masked in a tricky way *to force the model to train longer but produce better results* with three methods:

- Surprise the model by not masking a single token on 10% of the dataset; for example:

  `The cat sat on it [because] it was a nice rug.`

- Surprise the model by replacing the token with a random token on 10% of the dataset; for example:

  `The cat sat on it [often] it was a nice rug.`

- Replace a token with a `[MASK]` token on 80% of the dataset; for example:

  `The cat sat on it [MASK] it was a nice rug.`

The authors' bold approach avoids overfitting and forces the model to train efficiently.

BERT was also trained to perform next-sentence prediction.

Next-sentence prediction

The second method invented to train BERT is **NSP**. The input contains two sentences. In 50% of the cases, the second sentence was the actual second sentence of a document. In 50% of the cases, the second sentence was selected randomly and had no relation to the first one.

Two new tokens were added:

`[CLS]` is a binary classification token added to the beginning of the first sequence to predict if the second sequence follows the first sequence. A positive sample is usually a pair of consecutive sentences taken from a dataset. A negative sample is created using sequences from different documents.

`[SEP]` is a separation token that signals the end of a sequence, such as a sentence, sentence part, or question, depending on the task at hand.

For example, the input sentences taken out of a book could be:

```
The cat slept on the rug. It likes sleeping all day.
```

These two sentences will become one complete input sequence:

```
[CLS] the cat slept on the rug [SEP] it likes sleep ##ing all day[SEP]
```

This approach requires additional encoding information to distinguish sequence A from sequence B.

Note that the double hash (##) is because of the WordPiece tokenization used. The word “sleep” was tokenized separately from “ing.” This enables the tokenizer to work on a smaller dictionary of subwords, WordPiece, and then assemble them through the training process.

If we put the whole embedding process together, we obtain the following:

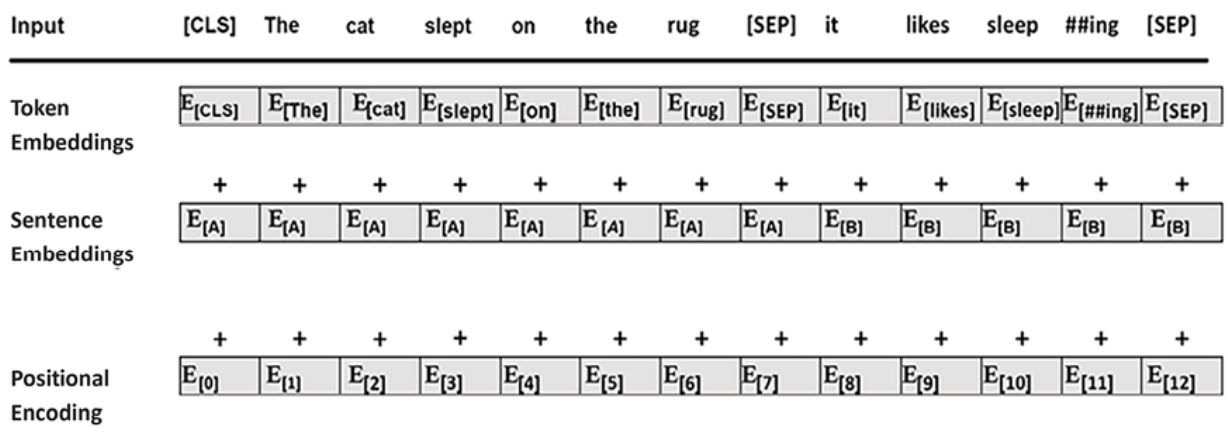

Figure 5.3: Input embeddings

The input embeddings are obtained by summing the token embeddings, the segment (sentence, phrase, word) embeddings, and the positional encoding embeddings.

The input embedding and positional encoding sub-layer of a BERT model can be summed up as follows:

- A sequence of words is broken down into `WordPiece` tokens.
- A `[MASK]` token will randomly replace the initial word tokens for MLM training.
- A `[CLS]` classification token is inserted at the beginning of a sequence for classification purposes.
- A `[SEP]` token separates two sentences (segments, phrases) for NSP training:
    - Sentence embedding is added to token embedding, so that sentence A has a different sentence embedding value than sentence B.
    - Positional encoding is learned. The sine-cosine positional encoding method of the Original Transformer is not applied.

Some additional key features of BERT are:

- It uses bidirectional attention in its multi-head attention sub-layers, opening vast horizons of learning and understanding relationships between tokens.
- It introduces scenarios of unsupervised embedding and pretraining models with unlabeled text. Unsupervised methods force the model to think harder during the multi-head attention learning process. This makes BERT learn how languages are built and apply this knowledge to downstream tasks without having to pretrain each time.
- It also uses supervised learning, covering all bases in the pretraining process.

BERT has improved the training environment of transformers. Let's now see the motivation for pretraining and how it helps the fine-tuning process.

Pretraining and fine-tuning a BERT model

BERT is a two-step framework. The first step is pretraining, and the second is fine-tuning, as shown in *Figure 5.4*:

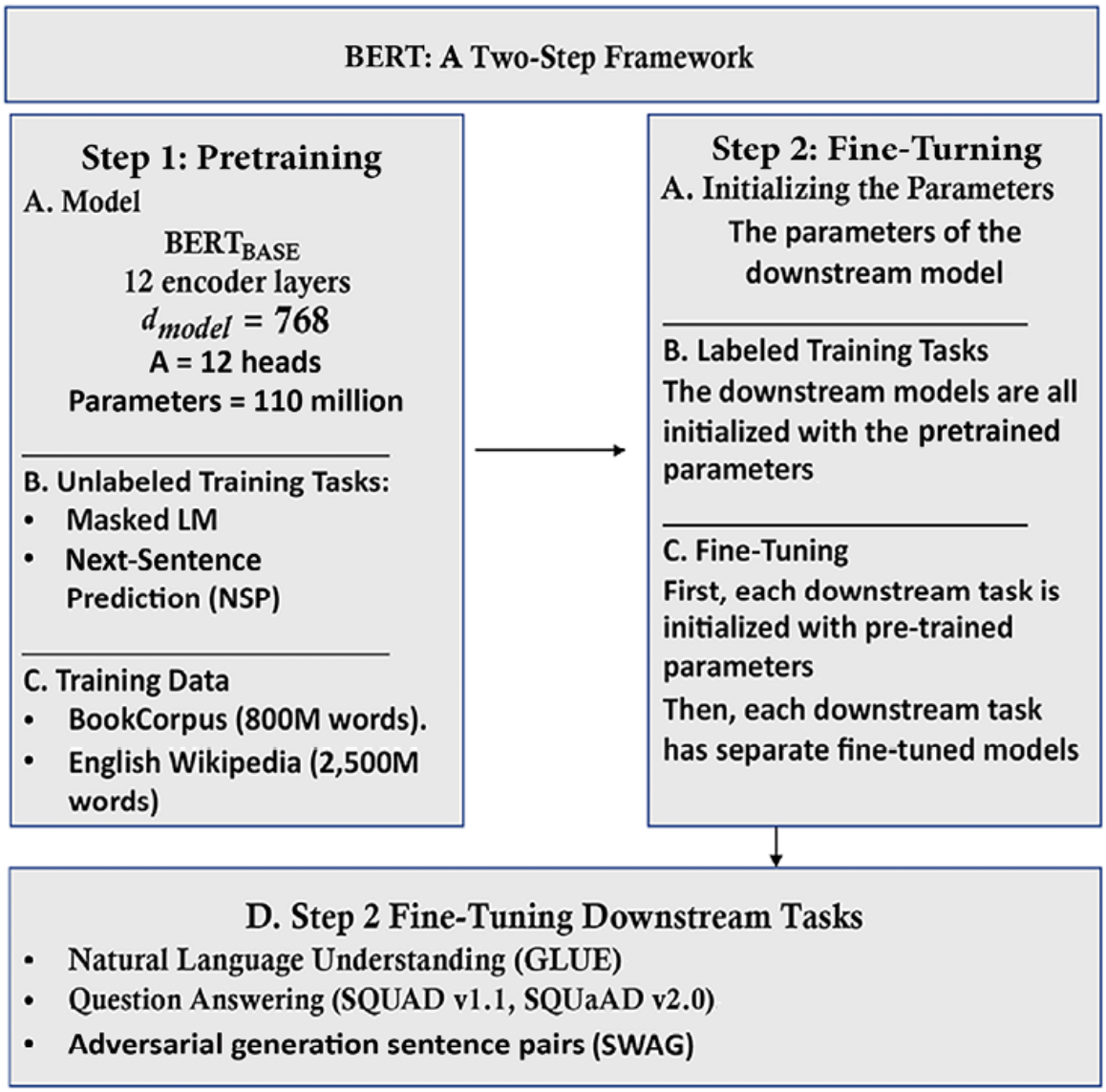

Figure 5.4: The BERT framework

Training a transformer model can take hours, if not days. Therefore, it takes quite some time to engineer the architecture and parameters and select the proper datasets to train a transformer model.

Pretraining is the first step of the BERT framework, which can be broken down into two sub-steps:

- Defining the model's architecture: number of layers, number of heads, dimensions, and the other building blocks of the model.
- Training the model on **MLM** and **NSP** tasks.

The second step of the BERT framework is fine-tuning, which can also be broken down into two sub-steps:

- Initializing the downstream model chosen with the trained parameters of the pretrained BERT model
- Fine-tuning the parameters for specific downstream tasks such as **Recognizing Textual Entailment** (**RTE**), question answering (`SQuAD v1.1`, `SQuAD v2.0`), and **Situations With Adversarial Generations** (**SWAG**).

In this chapter, the BERT model we will fine-tune will be trained on **the Corpus of Linguistic Acceptability** (**CoLA**). The downstream task is based on the *Neural Network Acceptability Judgments* paper by *Alex Warstadt*, *Amanpreet Singh*, and *Samuel R. Bowman*.

We will fine-tune a BERT model that will determine the grammatical acceptability of a sentence. The fine-tuned model will have acquired a certain level of linguistic competence.

We have gone through BERT's architecture and its pretraining and fine-tuning framework. Let's now fine-tune a BERT model.

Fine-tuning BERT

In this section, we will fine-tune a BERT model using Hugging Face. Hugging Face provides a large amount of resources for transformer models.

Hugging Face offers a significant number of pretrained models, such as BERT, GPT-2, RoBERTa, T5, and DistilBERT. These models can perform many tasks, such as those we went through in *Chapter 3, Emergent vs Downstream Tasks: The Unseen Depths of Transformers*. In addition, these models can be fine-tuned to fit your needs. This chapter focuses on the process of fine-tuning a Hugging Face BERT model. You can then apply the same method to other Hugging Face transformer models.

In *Chapter 15, Guarding the Giants: Mitigating Risks in Large Language Models*, we will use another platform, OpenAI, to fine-tune one of OpenAI's GPT models.

For this chapter, we will begin with fine-tuning BERT, which involves the following main steps:

- Retrieving the datasets, in this case, the CoLA datasets.
- Loading and configuring the pretrained model.
- Loading and preparing the data.
- Setting up the training parameters.
- Training the pretrained model for a new task. In this case, the trained model will learn how to assess linguistic acceptability.
- Evaluating the trained model.

It is essential to remember that this same process can be applied to other Hugging Face transformer models, such as RoBERTa, GPT-2, and many other models. Hugging Face provides documentation and a dynamic community. For more information on Hugging Face, explore their site: https://huggingface.co/.

Let's apply the fine-tuning process to a BERT model by first defining the goal we wish to achieve.

Defining a goal

The goal of fine-tuning BERT in this chapter is to focus on learning the ability to judge the grammatical acceptability of a text with the CoLA dataset, as seen in *Chapter 3, Emergent vs Downstream Tasks: The Unseen Depths of Transformers.* So, take some time to go back to the CoLA section of *Chapter 3* if necessary.

You can choose other tasks, including trying some of those we went through in *Chapter 3*, to fine-tune depending on your project.

We will measure the predictions with the **Matthews Correlation Coefficient** (**MCC**), which will be explained in the *Evaluating using the Matthews correlation coefficient* section.

Open `BERT_Fine_Tuning_Sentence_Classification_GPU.ipynb` in Google Colab (make sure you have a Gmail account). The notebook is in `Chapter05` in the GitHub repository of this book.

The title of each cell in the notebook is also the same as or very close to the title of each chapter subsection.

We will first examine why transformer models must take hardware constraints into account.

Hardware constraints

The scale of computation required to train and fine-tune transformer models makes them hardware-driven. This rapidly becomes a resource constraint. Small transformer models can be trained and fine-tuned on local machines or limited server configurations. However, large models such as state-of-the-art large OpenAI models and Google AI models require unique machine power.

The machine power required by **Large Language Models** (**LLMs**) has created a financial accessibility threshold. Therefore, Hugging Face offers a reasonable entry point to explore and implement transformer models.

Transformer models thus benefit from the parallel processing capability of GPUs. GPUs perform computational tasks far more efficiently than CPUs and will accelerate the training or fine-tuning process. Go to the **Runtime** menu in Google Colab, select **Change runtime type**, and select **GPU** in the **Hardware Accelerator** drop-down list.

The program will use Hugging Face modules, which we'll install next.

Installing Hugging Face Transformers

Hugging Face provides a pretrained BERT model. Hugging Face developed a base class named `PreTrainedModel`. We can load a model from a pretrained configuration by installing this class.

Hugging Face provides modules in TensorFlow and PyTorch. I recommend that a developer is comfortable with both

environments. AI research teams may use either or both environments.

In this chapter, we will install the modules required as follows:

```
try:
  import transformers
except:
  print("Installing transformers")
  !pip -q install transformers
```

The installation will run with the `-q` option, limiting the verbosity and only displaying warning or error messages.

We can now import the modules needed for the program.

Importing the modules

We will import the pretrained modules required, such as the pretrained `BERT tokenizer` and the configuration of the BERT model. The `BERT Adam` optimizer is also imported along with the sequence classification module:

```
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader, RandomSampler,
from keras.utils import pad_sequences
from sklearn.model_selection import train_test_split
from transformers import BertTokenizer, BertConfig
from transformers import AdamW, BertForSequenceClassification, get_lin
```

We will now import the progress bar module from tqdm:

```
from tqdm import tqdm, trange  #for progress bars
```

We can now import the widely used standard Python modules:

```
import pandas as pd
import io
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import Image #for image rendering
```

If all goes well, no message will be displayed, bearing in mind that Google Colab has pre-installed several of the modules we need on their VMs.

Specifying CUDA as the device for torch

We will now specify that torch uses the **Compute Unified Device Architecture** (**CUDA**) to put the parallel computing power of the NVIDIA card to work for our multi-head attention model:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
!nvidia-smi
```

The output may vary with Google Colab configurations. For example:

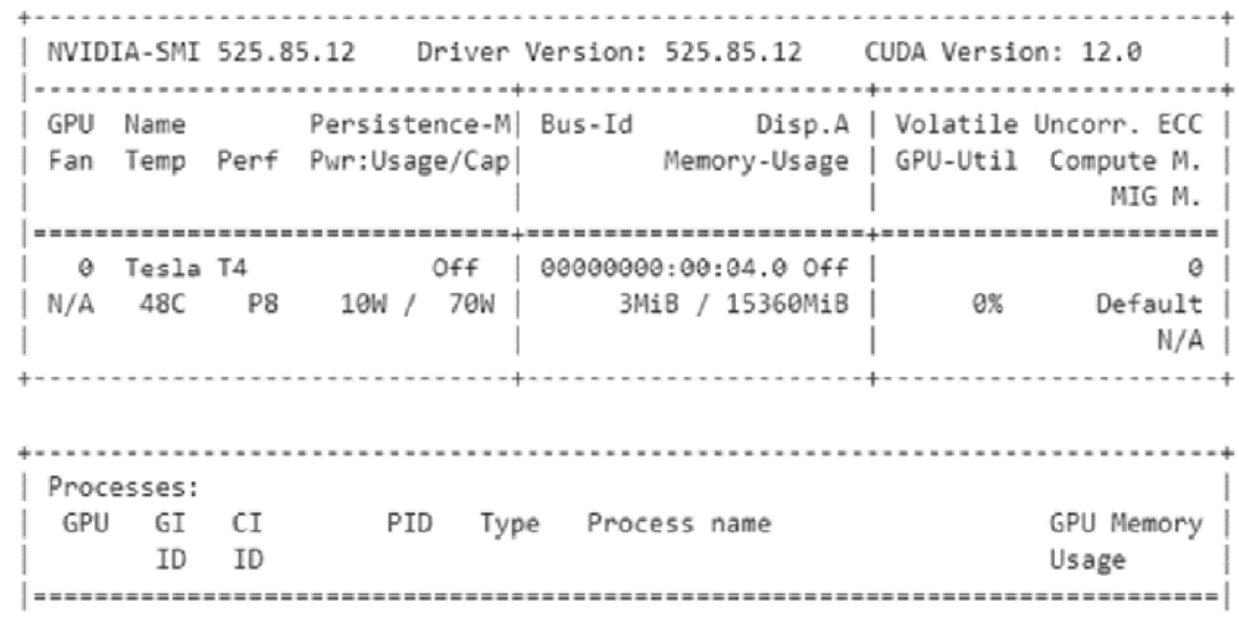

Figure 5.5: Output of NVIDIA-SMI

We will now load the dataset.

Loading the CoLA dataset

We will load *CoLA*, based on the *Warstadt et al.* (2018) paper.

Assessing linguistic acceptability is critical in **Natural Language Processing** (**NLP**). If necessary, take a moment to review the CoLA section of *Chapter 3, Emergent vs Downstream Tasks: The Unseen Depths of Transformers*.

The dataset implemented in this section was obtained from the CoLA homepage: https://nyu-mll.github.io/CoLA/ and uploaded to the GitHub directory of this chapter.

The following cells in the notebook automatically download the necessary files for fine-tuning, `in_domain_train.tsv`, and evaluating the predictions of the fine-tuned model, `out_of_domain_dev.tsv`:

```
import os
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers-for-
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers-for-
```

You should see them appear in the file manager:

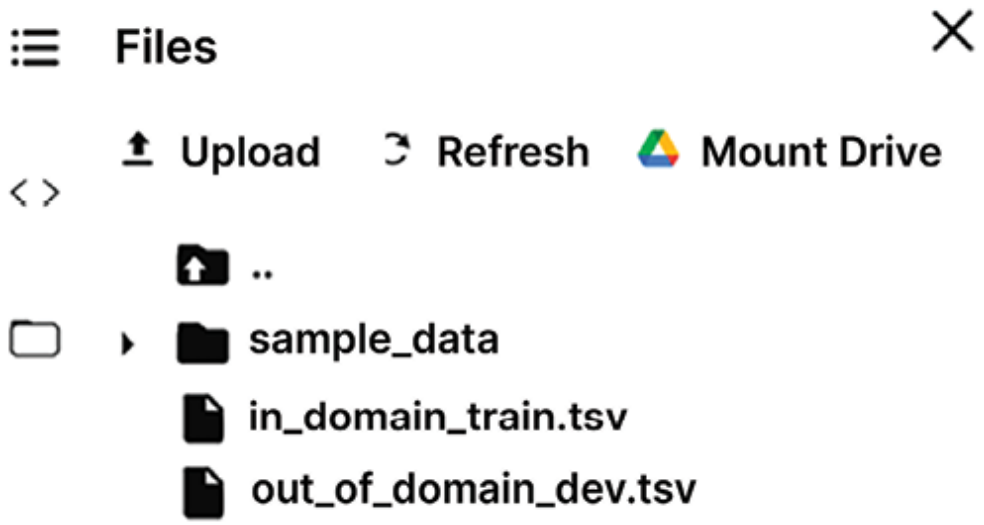

Figure 5.6: Uploading the datasets

Now, the program will load the datasets:

```
#source of dataset : https://nyu-mll.github.io/CoLA/
df = pd.read_csv("in_domain_train.tsv", delimiter='\t', header=None, n
df.shape
```

The output displays the shape of the dataset we have imported:

```
(8551, 4)
```

A 10-line sample is displayed to visualize the *acceptability judgment* task:

```
df.sample(10)
```

The output shows `10` lines of the labeled dataset, which may change after each run:

	sentence_source	label	label_notes	sentence
4517	ks08	1	NaN	do n't you even touch that !
7622	sks13	1	NaN	sharks seem to swim slowly in the tropics .
6549	g_81	0	*	the talkative and a bully man entered .
5336	b_73	1	NaN	jack eats caviar more than he eats mush .
460	bc01	0	*	the boat sank to collect the insurance .
1110	r-67	1	NaN	i ran an old man down .
8427	ad03	0	*	the dragons were slain all .
5140	ks08	0	*	it is kim on whom sandy relies on .
5639	c_13	1	NaN	the man loved peanut butter cookies .
5916	c_13	1	NaN	molly gave calvin a kiss .

Figure 5.7: 10 lines of the labeled dataset

Each sample in the `.tsv` files contains four tab-separated columns (column 0 is an index):

- Column 1: The source of the sentence (code).
- Column 2: The label (`0` = unacceptable, `1` = acceptable).
- Column 3: The label annotated by the author.
- Column 4: The sentence to be classified.

You can open the `.tsv` files locally to read a few samples of the dataset. The program will now process the data for the BERT model.

Creating sentences, label lists, and adding BERT tokens

The program will now create the sentences as described in the *Preparing the pretraining input environment* section of this chapter:

```
sentences = df.sentence.values
# Adding CLS and SEP tokens at the beginning and end of each sentence
sentences = ["[CLS] " + sentence + " [SEP]" for sentence in sentences]
labels = df.label.values
```

The `[CLS]` and `[SEP]` have now been added for the BERT model to find the start and end of a sentence.

The program now activates the tokenizer.

Activating the BERT tokenizer

In this section, we will initialize a pretrained BERT tokenizer. This will save the time it would take to train it from scratch.

The program selects an uncased tokenizer, activates it, and displays the first tokenized sentence:

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lowe
tokenized_texts = [tokenizer.tokenize(sent) for sent in sentences]
print ("Tokenize the first sentence:")
print (tokenized_texts[0])
```

The output contains the classification token and the sequence segmentation token:

```
Tokenize the first sentence:
['[CLS]', 'our', 'friends', 'wo', 'n', "'", 't', 'buy', 'this', 'analys
```

The program will now process the data.

Processing the data

We need to determine a fixed maximum length and process the data for the model. This is because the sentences in the datasets are short. But to make sure of this, the program sets the maximum length of a sequence to `128`, and the sequences are padded:

```
# Set the maximum sequence length. The longest sequence in our trainin
# In the original paper, the authors used a length of 512.
MAX_LEN = 128
# Use the BERT tokenizer to convert the tokens to their index numbers
input_ids = [tokenizer.convert_tokens_to_ids(x) for x in tokenized_tex
# Pad our input tokens
input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="long", tru
```

The sequences have been processed, and now the program creates the attention masks.

Creating attention masks

Now comes a tricky part of the process. We padded the sequences in the previous cell. But we want to prevent the model from paying attention to those padded tokens!

The idea is to apply a mask with a value of `1` for each token, then 0s will follow for padding:

```
attention_masks = []
# Create a mask of 1s for each token followed by 0s for padding
for seq in input_ids:
  seq_mask = [float(i>0) for i in seq]
  attention_masks.append(seq_mask)
```

The program will now split the data.

Splitting the data into training and validation sets

The program now performs the standard process of splitting the in-domain dataset data into training and validation sets:

```
# Use train_test_split to split our data into train and validation set
train_inputs, validation_inputs, train_labels, validation_labels = tra
train_masks, validation_masks, _, _ = train_test_split(attention_masks
```

The data is ready to be trained but needs to be adapted to torch.

Converting all the data into torch tensors

The fine-tuning model uses torch tensors. PyTorch tensors:

- Optimize the usage of GPUs.
- Facilitate GPU/CPU control. In this notebook, notice that the Hugging Face model and our datasets will be transferred to the GPU through the `to-device` command in the code of the training loop if the GPU is available, as defined in the *Specifying CUDA as the device for torch* section of this chapter and the notebook.

The program will thus convert the data into torch tensors:

```
# Torch tensors are the required datatype for our model
train_inputs = torch.tensor(train_inputs)
validation_inputs = torch.tensor(validation_inputs)
train_labels = torch.tensor(train_labels)
validation_labels = torch.tensor(validation_labels)
train_masks = torch.tensor(train_masks)
validation_masks = torch.tensor(validation_masks)
```

The conversion is over. Now, we need to create an iterator.

Selecting a batch size and creating an iterator

The program selects a batch size in this cell and creates an iterator. The iterator is an efficient way of looping through the data and loading batches instead of loading all the data in memory. The iterator, coupled with the torch `DataLoader`, can batch-train massive datasets without crashing the machine's memory.

In this model, the batch size is `32`:

```
# Select a batch size for training. For fine-tuning BERT on a specific
batch_size = 32
# Create an iterator of our data with torch DataLoader. This helps sav
# with an iterator the entire dataset does not need to be loaded into
train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler, batch
validation_data = TensorDataset(validation_inputs, validation_masks, v
validation_sampler = SequentialSampler(validation_data)
validation_dataloader = DataLoader(validation_data, sampler=validation
```

The data has been processed and is all set. The program can now load and configure the BERT model.

BERT model configuration

The program now initializes a BERT uncased configuration:

```
# Initializing a BERT bert-base-uncased style configuration
from transformers import BertModel, BertConfig
configuration = BertConfig()
# Initializing a model from the bert-base-uncased style configuration
model = BertModel(configuration)
# Accessing the model configuration
configuration = model.config
print(configuration)
```

The output displays the main Hugging Face parameters similar to the following (the library is often updated):

```
BertConfig {
  "attention_probs_dropout_prob": 0.1,
  "hidden_act": "gelu",
```

```
    "hidden_dropout_prob": 0.1,
    "hidden_size": 768,
    "initializer_range": 0.02,
    "intermediate_size": 3072,
    "layer_norm_eps": 1e-12,
    "max_position_embeddings": 512,
    "model_type": "bert",
    "num_attention_heads": 12,
    "num_hidden_layers": 12,
    "pad_token_id": 0,
    "type_vocab_size": 2,
    "vocab_size": 30522
  }
```

Let's go through some of the main parameters displayed:

- `attention_probs_dropout_prob`: `0.1` applies a `0.1` dropout ratio to the attention probabilities.
- `hidden_act`: `"gelu"` is a non-linear activation function in the encoder. It is a **Gaussian Error Linear Unit** activation function. The input is weighted by its magnitude, which makes it non-linear.
- `hidden_dropout_prob`: `0.1` is the dropout probability applied to the fully connected layers. Full connections can be found in the embeddings, encoder, and pooler layers. The output is not always a good reflection of the content of a sequence. Pooling the sequence of hidden states improves the output sequence.
- `hidden_size`: `768` is the dimension of the encoded layers and also the pooler layer.
- `initializer_range`: `0.02` is the standard deviation value when initializing the weight matrices.
- `intermediate_size`: `3072` is the dimension of the feed-forward layer of the encoder.

- `layer_norm_eps`: `1e-12` is the epsilon value for layer normalization layers.
- `max_position_embeddings`: `512` is the maximum length the model uses.
- `model_type`: `"bert"` is the name of the model.
- `num_attention_heads`: `12` is the number of heads.
- `num_hidden_layers`: `12` is the number of layers.
- `pad_token_id`: `0` is the ID of the padding token to avoid training padding tokens.
- `type_vocab_size`: `2` is the size of the `token_type_ids`, which identifies the sequences. For example, "the `dog[SEP]` The `cat[SEP]`" can be represented with token IDs `[0,0,0, 1,1,1]`.
- `vocab_size`: `30522` is the number of tokens the model uses to represent the `input_ids`.

With these parameters in mind, we can load the pretrained model.

Loading the Hugging Face BERT uncased base model

The program now loads the pretrained BERT model:

```
model = BertForSequenceClassification.from_pretrained("bert-base-uncas
model = nn.DataParallel(model)
model.to(device)
```

We have defined the model, defined parallel processing, and sent the model to the device.

This pretrained model can be trained further if necessary. It is interesting to explore the architecture in detail to visualize the parameters of each sub-layer, as shown in the following excerpt:

```
DataParallel(
  (module): BertForSequenceClassification(
    (bert): BertModel(
      (embeddings): BertEmbeddings(
        (word_embeddings): Embedding(30522, 768, padding_idx=0)
        (position_embeddings): Embedding(512, 768)
        (token_type_embeddings): Embedding(2, 768)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=Tr
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (encoder): BertEncoder(
        (layer): ModuleList(
          (0-11): 12 x BertLayer(
            (attention): BertAttention(
              (self): BertSelfAttention(
                (query): Linear(in_features=768, out_features=768, bias
                (key): Linear(in_features=768, out_features=768, bias=T
                (value): Linear(in_features=768, out_features=768, bias
                (dropout): Dropout(p=0.1, inplace=False)
              )
              (output): BertSelfOutput(
                (dense): Linear(in_features=768, out_features=768, bias
                (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_a
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
            (intermediate): BertIntermediate(
              (dense): Linear(in_features=768, out_features=3072, bias=
              (intermediate_act_fn): GELUActivation()
            )
            (output): BertOutput(
              (dense): Linear(in_features=3072, out_features=768, bias=
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_aff
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
```

```
        )
        (pooler): BertPooler(
          (dense): Linear(in_features=768, out_features=768, bias=True)
          (activation): Tanh()
        )
      )
      (dropout): Dropout(p=0.1, inplace=False)
      (classifier): Linear(in_features=768, out_features=2, bias=True)
    )
  )
```

Let's now go through the main parameters of the optimizer.

Optimizer grouped parameters

The program will now initialize the optimizer for the model's parameters. Fine-tuning a model begins with initializing the pretrained model parameter values (not their names).

The parameters of the optimizer include a weight decay rate to avoid overfitting, and some parameters are filtered.

The goal is to prepare the model's parameters for the training loop:

```
#This code is taken from:
# https://github.com/huggingface/transformers/blob/5bfcd0485ece086ebcb
# Don't apply weight decay to any parameters whose names include these
# (Here, the BERT doesn't have 'gamma' or 'beta' parameters, only 'bia
param_optimizer = list(model.named_parameters())
no_decay = ['bias', 'LayerNorm.weight']
# Separate the 'weight' parameters from the 'bias' parameters.
# - For the 'weight' parameters, this specifies a 'weight_decay_rate'
# - For the 'bias' parameters, the 'weight_decay_rate' is 0.0.
optimizer_grouped_parameters = [
    # Filter for all parameters which *don't* include 'bias', 'gamma',
    {'params': [p for n, p in param_optimizer if not any(nd in n for n
     'weight_decay_rate': 0.1},
```

```
    # Filter for parameters which *do* include those.
    {'params': [p for n, p in param_optimizer if any(nd in n for nd in
     'weight_decay_rate': 0.0}
]
# Note - 'optimizer_grouped_parameters' only includes the parameter va
```

Let's go through some of the main parameters in the following cells:

- Looking into layer 3 of the `param_optimizer`, all the parameters of the `model` and their names are collected in `param_optimizer`.
- This function produces an iterator. The iterator contains tuples. Each tuple contains the name of a parameter and the parameters, which are often tensors in PyTorch. The following code displays the name `n` of the parameters and the value `p` of the parameters of layer 3:

```
 # Displaying a sample of the parameter_optimizer:  layer 3
layer_parameters = [p for n, p in model.named_parameters() if 'la
```

The following excerpt of the output displays a string and the value of its parameters:

```
'module.bert.embeddings.word_embeddings.weight', Parameter contair
```

Note that additional information is displayed. `cuda:0` means that the first CUDA-enabled GPU has been activated. `requires_grad=True` means that the gradients will be computed and

the tensors updated during the training process. The model is "learning."

- Looking into `no_decay`:

  ```
  no_decay = ['bias', 'LayerNorm.weight']
  ```

  Weight decay regularization will not be applied to the `bias` and `LayerNorm.weight` to avoid overfitting.

- The `optimizer_grouped_parameters` variable contains two dictionaries. These two groups of parameters contain the weight decay rate:

  `Group 1:` contains the parameters for which weight decay will be applied:

  ```
  Group 1:
  Weight decay rate: 0.1
  Parameter 1: Parameter containing:
  tensor([[-0.0102, -0.0615, -0.0265,  ..., -0.0199, -0.0372,…
  ```

  `Group 2:` contains the parameters for which weight decay will not be applied, such as `bias` and `LayerNorm.weight`:

  ```
  Group 2:
  Weight decay rate: 0.0
  Parameter 1: Parameter containing:
  tensor([0.9257, 0.8852, 0.8587, 0.8617, 0.8934, 0.8964, 0.9290,…
  ```

The parameter names are not included because the optimizer does not require them.

The parameters have been retrieved, and the model is ready for the training loop.

The hyperparameters for the training loop

The hyperparameters for the training loop are critical, though they seem innocuous.

For example, the **Adam** optimizer will activate weight decay and undergo a warmup phase.

The number of `epochs` and `steps` will determine how much the model will "learn." Several runs may prove helpful in determining the optimal level of training.

The learning rate (`lr`) and warmup rate (`warmup`) should be set to a very small value early in the optimization phase and gradually increase after a certain number of iterations. This avoids large gradients and overshooting the optimization goals.

Some researchers argue that the gradients at the output level of the sub-layers before layer normalization do not require a warmup rate. However, solving this problem requires many experimental runs.

The optimizer is a BERT version of Adam called `BertAdam`:

```
optimizer = BertAdam(optimizer_grouped_parameters,
                     lr=2e-5,
                     warmup=.1)
```

The program adds an accuracy measurement function to compare the predictions to the labels:

```
#Creating the Accuracy Measurement Function
# Function to calculate the accuracy of our predictions vs labels
def flat_accuracy(preds, labels):
    pred_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
    return np.sum(pred_flat == labels_flat) / len(labels_flat)
```

The data is ready. The parameters are prepared. It's time to activate the training loop!

The training loop

The training loop follows standard learning processes. The number of epochs is set to `4`, and measurements for loss and accuracy will be plotted. Next, the training loop uses the `dataloader` to load and train batches. Finally, the training process is measured and evaluated.

The code starts by initializing the `train_loss_set`, which will store the loss and accuracy, which will be plotted. It starts training its epochs and runs a standard training loop, as shown in the following excerpt:

```
t = []
# Store our loss and accuracy for plotting
train_loss_set = []
# Number of training epochs (authors recommend between 2 and 4)
epochs = 4
# trange is a tqdm wrapper around the normal python range
for _ in trange(epochs, desc="Epoch"):
…./…
    tmp_eval_accuracy = flat_accuracy(logits, label_ids)
```

```
        eval_accuracy += tmp_eval_accuracy
        nb_eval_steps += 1
    print("Validation Accuracy: {}".format(eval_accuracy/nb_eval_steps))
```

The output displays the information for each `epoch` with the `trange` wrapper, `for _ in trange(epochs, desc="Epoch")`:

```
Epoch:   0%|          | 0/4 [00:00<?, ?it/s]
Train loss: 0.5381132976395461
Epoch:  25%|██        | 1/4 [07:54<23:43, 474.47s/it]
Validation Accuracy: 0.788966049382716
Train loss: 0.315329696132929
Epoch:  50%|█████     | 2/4 [15:49<15:49, 474.55s/it]
Validation Accuracy: 0.836033950617284
Train loss: 0.1474070605354314
Epoch:  75%|███████   | 3/4 [23:43<07:54, 474.53s/it]
Validation Accuracy: 0.814429012345679
Train loss: 0.07655430570461196
Epoch: 100%|██████████| 4/4 [31:38<00:00, 474.58s/it]
Validation Accuracy: 0.810570987654321
```

Transformer models are evolving quickly, and deprecation messages and errors might occur. Hugging Face is no exception, and we must update our code accordingly when this happens.

The model is trained. We can now display the training evaluation.

Training evaluation

The loss and accuracy values were stored in `train_loss_set` as defined at the beginning of the training loop.

The program now plots the measurements:

```
plt.figure(figsize=(15,8))
plt.title("Training loss")
plt.xlabel("Batch")
plt.ylabel("Loss")
plt.plot(train_loss_set)
plt.show()
```

The output is a graph that shows that the training process went well and was relatively efficient:

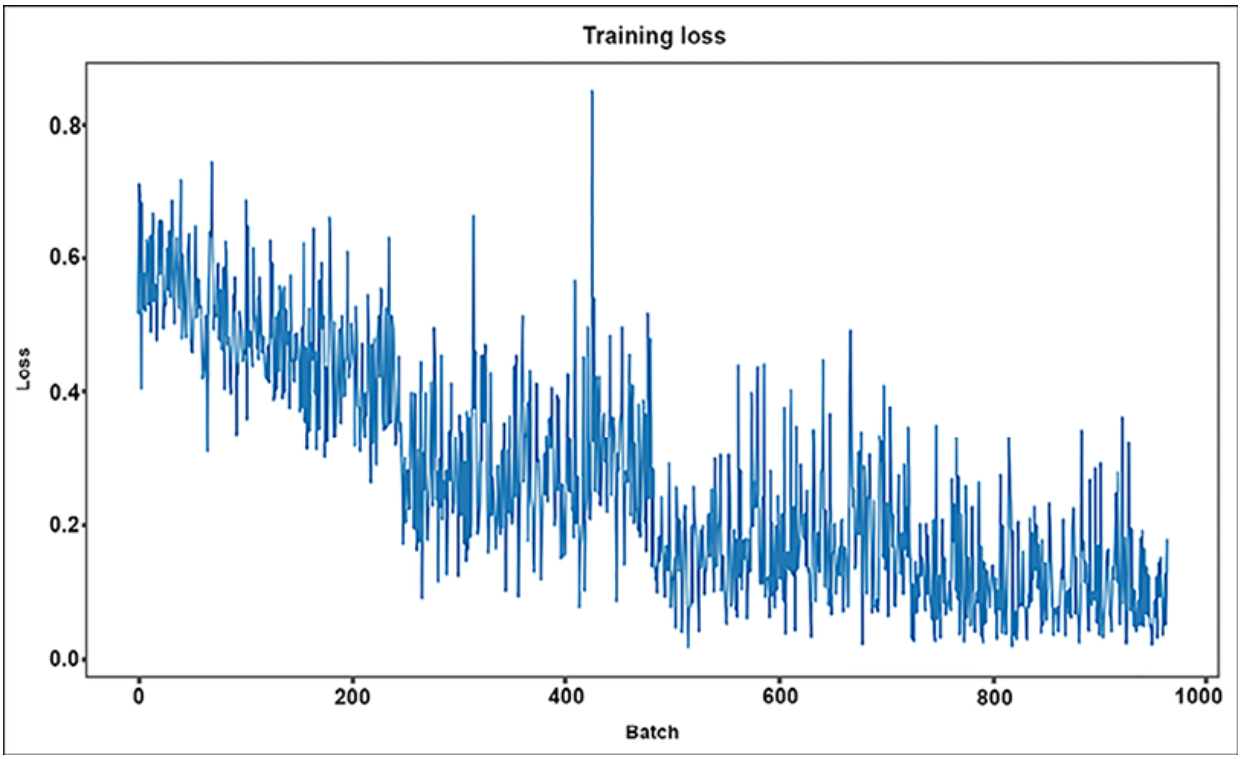

Figure 5.8: Training loss per batch

The model has been fine-tuned. We can now run predictions.

Predicting and evaluating using the holdout dataset

The BERT downstream model has been trained with the `in_domain_train.tsv` dataset. The program will now make predictions using the holdout (testing and evaluation) dataset in the `out_of_domain_dev.tsv` file. The goal is to predict whether the sentence is grammatically correct.

The following excerpts of the code show the main steps of the evaluation process.

First, the data preparation process applied to the training data is repeated in the part of the code for the holdout dataset:

```
df = pd.read_csv("out_of_domain_dev.tsv", delimiter='\t', header=None,
df.shape
```

The output of `df.shape` displays the number of lines to assess and the shape of the dataset:

```
(516, 4)
```

Then, the evaluation process begins by creating the sentences to assess and their labels:

```
# Create sentence and label lists
sentences = df.sentence.values
# We need to add special tokens at the beginning and end of each sente
sentences = ["[CLS] " + sentence + " [SEP]" for sentence in sentences]
```

```
labels = df.label.values
tokenized_texts = [tokenizer.tokenize(sent) for sent in sentences]
.../...
```

The program then runs batch predictions using the `dataloader`:

```
# Predict
for batch in prediction_dataloader:
  # Add batch to GPU
  batch = tuple(t.to(device) for t in batch)
  # Unpack the inputs from our dataloader
  b_input_ids, b_input_mask, b_labels = batch
  # Telling the model not to compute or store gradients, saving memory
with torch.no_grad():
    # Forward pass, calculate logit predictions
logits = model(b_input_ids, token_type_ids=None, attention_mask=b_inpu
```

The logits and labels of the predictions are moved to the CPU:

```
 # Move logits and labels to CPU
  logits =  logits['logits'].detach().cpu().numpy()
  label_ids = b_labels.to('cpu').numpy()
```

Then, the assessment is made by choosing the class with the highest probability:

```
 # The predicted class is the one with the highest probability
  batch_predictions = np.argmax(probabilities, axis=1)
```

Let's go deeper into this process in the following subsection.

Exploring the prediction process

It is interesting to dig into the output of the prediction process. Let's take a look at the code that follows `batch_predictions`.

We can sum up the process by creating five steps and adding a function of our own:

```
    print(f"Sentence: {sentence}")
```

This will display the sentence to assess. For example:

```
Sentence: somebody just left - guess who.
```

```
    print(f"Prediction: {logits[i]}")
```

Now, we can display the fine-tuned model's prediction:

```
Prediction: [-2.6922598  2.1979954]
```

The first element, `-2.6922598`, is the first class's logit value, representing incorrect sentences in this case.

The second element, `2.1979954`, is the second class's logit value, which, in this case, represents the acceptable sentences.

Let's apply a softmax function to interpret the result.

```
    print(f"Sofmax probabilities", softmax(logits[i]))
```

A softmax function was added to the notebook just above the prediction cell:

```
#Softmax logits
import numpy as np
def softmax(logits):
        e = np.exp(logits)
         return e / np.sum(e)
```

The result is an interpretable array of two probabilities that sum up to 1:

```
Sofmax probabilities [0.00746338 0.9925366 ]
```

The first element, the *unacceptable* class, has a probability of `0.00746338.`

The second element, the *acceptable* class, has a probability of `0.9925366.`

We can see that the probability of the acceptable class exceeds that of the unacceptable class.

We can now look at the results produced by the fine-tuned model.

```
print(f"Prediction: {batch_predictions[i]}")
```

The prediction of the model for the sentence is:

```
Prediction: 1
```

This matches the detailed analysis of the output we made.

```
    print(f"True label: {label_ids[i]}")
```

The label of the dataset is:

```
True label: 1
```

We can see that, in this case, the prediction matches the true label in the dataset.

Once the predictions are made, the raw predictions and their true labels are stored:

```
        predictions.append(logits)
        true_labels.append(label_ids)
```

The program can now evaluate the predictions.

Evaluating using the Matthews correlation coefficient

The **Matthews Correlation Coefficient** (**MCC**) was initially designed to measure the quality of binary classifications and can be modified to be a multi-class correlation coefficient. A two-class classification can be made with four probabilities for each prediction:

- *TP* = True Positive
- *TN* = True Negative

- *FP* = False Positive
- *FN* = False Negative

Brian W. Matthews, a biochemist, designed it in 1975, inspired by his predecessors' *phi* function. Since then, it has evolved into various formats, such as the following:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

The value produced by the MCC is between `-1` and `+1`. `+1` is the maximum positive value of a prediction. `-1` is an inverse prediction. `0` is an average random prediction.

Linguistic acceptability can be measured with the MCC.

The MCC is imported from `sklearn.metrics`:

```
# Import and evaluate each test batch using Matthew's correlation coef
from sklearn.metrics import matthews_corrcoef
```

A set of predictions is created:

```
matthews_set = []
```

The MCC value is calculated and stored in `matthews_set`:

```
for i in range(len(true_labels)):
  matthews = matthews_corrcoef(true_labels[i],
                 np.argmax(predictions[i], axis=1).flatten())
  matthews_set.append(matthews)
```

You may see messages due to library and module version changes.

The final score will be based on the entire test set.

Matthews correlation coefficient evaluation for the whole dataset

The MCC is a practical way to evaluate a classification model.

The program will now aggregate the true values for the whole dataset:

```
# Flatten the predictions and true values for aggregate Matthew's eval
flat_predictions = [item for sublist in predictions for item in sublis
flat_predictions = np.argmax(flat_predictions, axis=1).flatten()
flat_true_labels = [item for sublist in true_labels for item in sublis
matthews_corrcoef(flat_true_labels, flat_predictions)
```

The metric compares the labels (0 or 1) provided by the evaluation dataset and the labels produced by the prediction using the MCC. The result might vary from one run to another because of the stochastic nature of model predictions.

The result, in this case, is:

```
MCC: 0.524309707232222
```

The Matthews correlation coefficient measures the quality of binary classifications with an output value ranging from `-1` to `+1`:

- `+1` means that the predictions were 100% accurate.

- `0` is the same as a random calculation.
- `-1` indicates that the predictions are totally false.

In this case, an MCC of 0.54 is relatively efficient. The model has learned a significant amount of information but can be improved.

With this final evaluation of the fine-tuning of the BERT model, we have an overall view of the fine-tuning framework.

We will now interact with the model to explore its scope and limits.

Building a Python interface to interact with the model

In this section, we will first save the model and then build an interface to interact with our trained model.

Let's first save the model if we choose to.

Saving the model

The following code will save the model's files:

```
# Specify a directory to save your model and tokenizer
save_directory = "/content/model"
# If your model is wrapped in DataParallel, access the original model
if isinstance(model, torch.nn.DataParallel):
    model.module.save_pretrained(save_directory)
else:
    model.save_pretrained(save_directory)
# Save the tokenizer
tokenizer.save_pretrained(save_directory)
```

The saved `/content/model` directory contains:

- `tokenizer_config.json`: Configuration details specific to the tokenizer.
- `special_tokens_map.json`: Mappings for any special tokens.
- `vocab.txt`: The vocabulary of tokens that the tokenizer can recognize.
- `added_tokens.json`: Any tokens that were added to the tokenizer after its initial creation.

When a model is wrapped in `DataParallel` for multi-GPU training in PyTorch, the original model becomes an attribute of the `DataParallel` object.

Remember to save your files outside of the Colab environment, as files in the Colab environment are discarded once the session ends. The saved model can be large.

In this case, one solution is to mount Google Drive:

```
from google.colab import drive
drive.mount('/content/drive')
```

Then, save the files in the `/content/model` directory to Google Drive.

The files can be loaded in another session, as shown in this excerpt of the code:

```
from transformers import BertTokenizer, BertForSequenceClassification
# Directory where the model and tokenizer were saved
load_directory = "/content/model/"
…/…
```

The next step is to build an interface to interact with the trained model.

Creating an interface for the trained model

First, make sure the model is in evaluation mode:

```
from transformers import BertTokenizer, BertForSequenceClassification
import torch
model.eval()  # set the model to evaluation mode
```

Now, we create a function to interact with the model, as shown in the following excerpt:

```
def predict(sentence, model, tokenizer):
    # Add [CLS] and [SEP] tokens
    sentence = "[CLS] " + sentence + " [SEP]"
# Tokenize the sentence
    tokenized_text = tokenizer.tokenize(sentence)
…/…
```

We can sum up the function as follows:

1. The function takes in three parameters: a sentence, a model, and a tokenizer.
2. The sentence is preprocessed by adding [CLS] and [SEP] tokens, which are special tokens used in BERT-like models.
3. The sentence is then tokenized into sub-words using the provided tokenizer.

4. The tokens are converted into corresponding IDs from the model's vocabulary.
5. A segment ID is defined for each token, set to 0 since there's only one sequence.
6. The lists of token IDs and segment IDs are converted into PyTorch tensors.
7. A prediction is made using the model without updating its gradients.
8. The model outputs logits, which represent raw prediction values.
9. The label with the highest logit is selected as the predicted label.
10. The predicted label is returned.

We can now interact with the model.

Interacting with the model

To build the interface, install `ipywidgets`:

```
import ipywidgets as widgets
from IPython.display import display
def model_predict_interface(sentence):
    prediction = predict(sentence, model, tokenizer)
    if prediction == 0:
        return "Grammatically Incorrect"
elif prediction == 1:
        return "Grammatically Correct"
else:
        return f"Label: {prediction}"
text_input = widgets.Textarea(
    placeholder='Type something',
    description='Sentence:',
    disabled=False,
```

```
    layout=widgets.Layout(width='100%', height='50px')  # Adjust width
)
output_label = widgets.Label(
    value='',
    layout=widgets.Layout(width='100%', height='25px'),  # Adjust widt
    style={'description_width': 'initial'}
)
def on_text_submit(change):
    output_label.value = model_predict_interface(change.new)
text_input.observe(on_text_submit, names='value')
display(text_input, output_label)
```

Finally, we interact by entering sentences that are classified in real time as correct or incorrect, as shown in *Figure 5.9*:

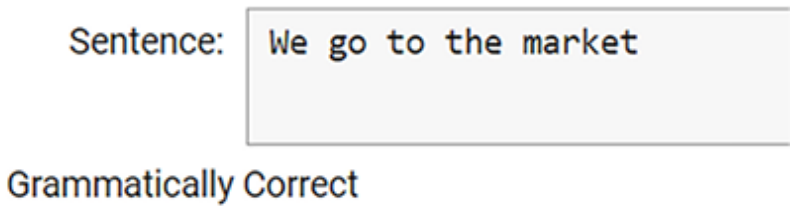

Figure 5.9: A grammatically correct sentence

The following is a grammatically incorrect example:

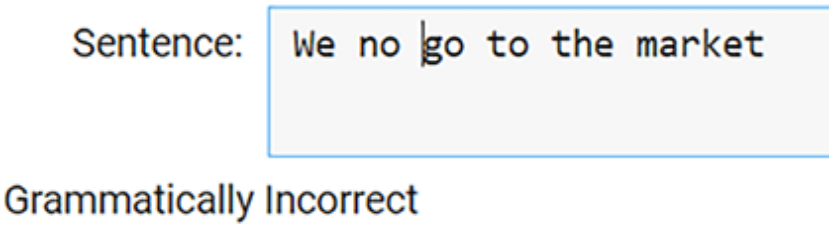

Figure 5.10: A grammatically incorrect sentence

Enter more examples to explore the scope and find the limits when the model is generalized.

We have now been through a complete training process and interacted with our trained model!

Let's now summarize our fine-tuning journey.

Summary

In this chapter, we explored the process of fine-tuning a transformer model. We achieved this by implementing the fine-tuning process of a pretrained Hugging Face BERT model.

We began by analyzing the architecture of BERT, which only uses the encoder stack of transformers and uses bidirectional attention. BERT was designed as a two-step framework. The first step of the framework is to pretrain a model. The second step is to fine-tune the model.

We then configured a fine-tuning BERT model for an *acceptability judgment* downstream task. The fine-tuning process went through all phases of the process.

We installed the Hugging Face transformers and considered the hardware constraints, including selecting CUDA as the device for torch. We retrieved the CoLA dataset from GitHub. We loaded and created in-domain (training data) sentences, label lists, and BERT tokens.

The training data was processed with the BERT tokenizer and other data preparation functions, including the attention masks. Then, the data was split into training and validation sets. We selected a batch size to optimize the process and created an iterator.

We then loaded a Hugging Face BERT uncased base model. The process could have been implemented with another tokenizer and another Hugging Face pretrained model such as GPT-2, T5, RoBERTa, or others.

We grouped the parameters, defined the hyperparameters, and ran the training loop, including evaluations during the process.

Once the training was over, we used the out-of-domain holdout dataset to evaluate the output of the fined-tuned model. We explored the prediction process and implemented the Matthews correlation coefficient to evaluate the results.

Finally, we built an interface with `ipywidgets` to interact with our trained model.

Fine-tuning a pretrained model takes fewer machine resources than training a model for downstream tasks from scratch. Fine-tuned models can perform a variety of tasks. However, in some cases, fine-tuning a model is insufficient to reach our goals. For example, we might need to pretrain a transformer model practically from scratch.

In the next chapter, *Chapter 6*, *Pretraining a Transformer from Scratch through RoBERTa,* we will build and pretrain a transformer model from scratch.

Questions

1. BERT stands for Bidirectional Encoder Representations from Transformers. (True/False)
2. BERT is a two-step framework. *Step 1* is pretraining. *Step 2* is fine-tuning. (True/False)

3. Fine-tuning a BERT model implies training parameters from scratch. (True/False)
4. BERT only pretrains using all downstream tasks. (True/False)
5. BERT pretrains with **MLM**. (True/False)
6. BERT pretrains with **NSP**. (True/False)
7. BERT pretrains on mathematical functions. (True/False)
8. A question-answer task is a downstream task. (True/False)
9. A BERT pretraining model does not require tokenization. (True/False)
10. Fine-tuning a BERT model takes less time than pretraining. (True/False)

References

- *Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin*, 2017, *Attention Is All You Need*: https://arxiv.org/abs/1706.03762
- *Alex Warstadt, Amanpreet Singh, Samuel R. Bowman*, 2018, *Neural Network Acceptability Judgments*: https://arxiv.org/abs/1805.12471
- The **Corpus of Linguistic Acceptability** (**CoLA**): https://nyu-mll.github.io/CoLA/
- Documentation on Hugging Face models:
  - https://huggingface.co/transformers/pretrained_models.html
  - https://huggingface.co/transformers/model_doc/bert.html

- https://huggingface.co/transformers/model_doc/roberta.html
- https://huggingface.co/transformers/model_doc/distilbert.html

Further reading

- *Vladislav Mosin, Igor Samenko, Alexey Tikhonov, Borislav Kozlovskii, and Ivan P. Yamshchikov*, 2021, *Fine-Tuning Transformers: Vocabulary Transfer*: https://arxiv.org/abs/2112.14569
- *Yi Tay, Mostafa Dehghani, Jinfeng Rao, William Fedus, Samira Abnar, Hyung Won Chung, Sharan Narang, Dani Yogatama, Ashish Vaswani, and Donald Metzler*, 2022, *Scale Efficiently: Insights from Pre-training and Fine-tuning Transformers*: https://arxiv.org/abs/2109.10686

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

https://www.packt.link/Transformers