FINE-TUNING PRE-TRAINTED SENTIMENT ANALYSIS USING

(BERT, ROBERTA) MODELS

TEAM MEMBER

810021106001 : A.ABDUL BAZIR

PHASE 2 : PROJECT SUBMISSION

BERT model:

BERT is super exciting algorithm and not only for me, but for the whole community of NLP(Natural Language Processing).

It's super powerful. It's super interesting. And I'm really glad to share it with you, how to use it and how it works.

Before going to BERT, let's Just take a look on NLP in a more general way. NLP(natural language processing) is the part of A.I. that deals with human language. Actually, It's pretty much everywhere. For instance, the Web or your search engine uses NLP to optimize the results. On a recent note, the vocal assistance's like Siri or Alexa use NLP technique to understand what we say. NLP is also used in email box for spam detection. NLP is used in the translators that are widely used. Chatbots are built using NLP.

So there is a lot of research done in this field, and BERT is an advanced algorithm for NLP released by Google.

Here are few points about BERT.

Provides a better understanding of words and sentences in the context

Already implemented in Google search engine for instance

BERT is a tool that is supposed to understand the language and provide what we call a language modeling.

What is BERT?

BERT stands for Bidirectional Encoder Representations from Transformers. I will split this full form into three parts.

1 . Encoder Representations: BERT is a language modeling system which is pre-trained with huge corpus and huge processing power. We just need use this pre-trained model and fine-tune it for our need.

What is Language modeling?

It means that BERT gives the best, the most efficient and the most flexible representation for words and sequences. For example, We give a sentence or two sentences to BERT and it will generate a single vector or a list of vector that can be used as per our need.

2. From Transformer: Transformer is basic building block for the architecture of BERT. So Google developed transformer which was designed to tackle many sequences to sequence tasks to build a translator or a chatbot. Then they use the same transformer in a small part and use it even smarter way to create BERT.

3. Bidirectional: Most of the time in NLP project, we want to predict the next words in our sentences. And to predict next word i.e the right part of the sentence, we would need access to the left part of the sentence. Even sometimes we have access only to the right part, and we need to predict left part of the sentence. Special case is when some model is trained with left parts and right parts of the sentences separately and then concatenate both. So it becomes pseudo bidirectional. BERT uses left and right context when dealing with a words and It achieves to have a fully bidirectional model. This means it has access to whole context or sentence to predict words,. This is what make BERT more powerful.

Having learned above concept, Let's jump to the points where BERT can applied.

Application of BERT:

Use the tokenizer to process the data

Use BERT as an embedding layer

Fine tune BERT, the core of your model

In this blog, we will learn about BERT's tokenizer for data processing **(sentiment Analyzer).**

Sentiment Analyzer:

In this project, we will try to improve our personal model (in this case CNN for classification) using tokenizer with BERT.

So to start with, we will first build a classification model to assess if a tweet is positive or negative in terms of feeling..

Fine tuning BERT model for sentiment analysis :

Step 1 : Import necessary libraries

Step 2 : Load the dataset

Step 3 : Pre-processing

Step 4 : Tokenization & Encoding

Step 5 : Build the classification model

Step 6 : Evaluate the model

Step 7 : prediction using user inputs

Step 1: Import the necessary libraries

Python3

```
import os
import shutil
import tarfile
import tensorflow as tf
from transformers import BertTokenizer, TFBertForSequenceClassification
import pandas as pd
from bs4 import BeautifulSoup
import re
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.offline as pyo
import plotly.graph_objects as go
from wordcloud import WordCloud, STOPWORDS
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
```

Step 2: Load the dataset

Python3

```
# Get the current working directory
current_folder = os.getcwd()
```

```
dataset = tf.keras.utils.get_file(
    fname ="aclImdb.tar.gz",
    origin ="http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz",
    cache_dir=   current_folder,
    extract = True)
```

check the dataset folder

Python3

```
dataset_path = os.path.dirname(dataset)
# Check the dataset
os.listdir(dataset_path)
```

Step 3: Preprocessing

Python3

```
sentiment_counts = train_df['sentiment'].value_counts()


fig =px.bar(x= {0:'Negative',1:'Positive'},
            y= sentiment_counts.values,
            color=sentiment_counts.index,
            color_discrete_sequence =   px.colors.qualitative.Dark24,
```

```
            title='<b>Sentiments Counts')


fig.update_layout(title='Sentiments Counts',
                  xaxis_title='Sentiment',
                  yaxis_title='Counts',
                  template='plotly_dark')


# Show the bar chart
fig.show()
pyo.plot(fig, filename = 'Sentiments Counts.html', auto_open = True)
```

Text Cleaning

Python3

```
def text_cleaning(text):
    soup = BeautifulSoup(text, "html.parser")
    text = re.sub(r'\[[^]]*\]', '', soup.get_text())
    pattern = r"[^a-zA-Z0-9\s,']"
    text = re.sub(pattern, '', text)
    return text
```

Apply text_cleaning

Python3

```
# Train dataset
train_df['Cleaned_sentence'] = train_df['sentence'].apply(text_cleaning).tolist()
# Test dataset
test_df['Cleaned_sentence'] = test_df['sentence'].apply(text_cleaning)
```

Separate input text and target sentiment of both train and test

Python3

```
# Training data
#Reviews = "[CLS] " +train_df['Cleaned_sentence'] + "[SEP]"
Reviews = train_df['Cleaned_sentence']
Target = train_df['sentiment']


# Test data
#test_reviews =  "[CLS] " +test_df['Cleaned_sentence'] + "[SEP]"
test_reviews = test_df['Cleaned_sentence']
test_targets = test_df['sentiment']
```

Split TEST data into test and validation

Python3

```
x_val, x_test, y_val, y_test = train_test_split(test_reviews,
```

```
                                                    test_targets,
                                                    test_size=0.5,
                                                    stratify = test_targets)
```

Step 4: Tokenization & Encoding

BERT tokenization is used to convert the raw text into numerical inputs that can be fed into the BERT model. It tokenized the text and performs some preprocessing to prepare the text for the model's input format. Let's understand some of the key features of the BERT tokenization model.

1. BERT tokenizer splits the words into subwords or workpieces. For example, the word "geeksforgeeks" can be split into "geeks" "##for", and"##geeks". The "##" prefix indicates that the subword is a continuation of the previous one. It reduces the vocabulary size and helps the model to deal with rare or unknown words.
2. BERT tokenizer adds special tokens like [CLS], [SEP], and [MASK] to the sequence. These tokens have special meanings like :
    - [CLS] is used for classifications and to represent the entire input in the case of sentiment analysis,
    - [SEP] is used as a separator i.e. to mark the boundaries between different sentences or segments,
    - [MASK] is used for masking i.e. to hide some tokens from the model during pre-training.
3. BERT tokenizer gives their components as outputs:
    - input_ids: The numerical identifiers of the vocabulary tokens
    - token_type_ids: It identifies which segment or sentence each token belongs to.
    - attention_mask: It flags that inform the model which tokens to pay attention to and which to disregard.

Load the pre-trained BERT tokenizer

Python3

```
#Tokenize and encode the data using the BERT tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)
```

Apply the BERT tokenization in training, testing and validation dataset

Python3

```
max_len= 128
# Tokenize and encode the sentences
X_train_encoded = tokenizer.batch_encode_plus(Reviews.tolist(),
                                              padding=True,
                                              truncation=True,
                                              max_length = max_len,
                                              return_tensors='tf')

X_val_encoded = tokenizer.batch_encode_plus(x_val.tolist(),
                                              padding=True,
                                              truncation=True,
                                              max_length = max_len,
                                              return_tensors='tf')

X_test_encoded = tokenizer.batch_encode_plus(x_test.tolist(),
                                              padding=True,
                                              truncation=True,
```

```
                                                    max_length = max_len,
                                                    return_tensors='tf')
```

Check the encoded dataset

Python3

```
k = 0
print('Training Comments -->>',Reviews[k])
print('\nInput Ids -->>\n',X_train_encoded['input_ids'][k])
print('\nDecoded Ids -->>\n',tokenizer.decode(X_train_encoded['input_ids'][k]))
print('\nAttention Mask -->>\n',X_train_encoded['attention_mask'][k])
print('\nLabels -->>',Target[k])
```

Step 5: Build the classification model

Lad the model

Python3

```
# Intialize the model
model = TFBertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)
```

Step 6:Evaluate the model

Python3

```
#Evaluate the model on the test data
```

```
test_loss, test_accuracy = model.evaluate(
    [X_test_encoded['input_ids'], X_test_encoded['token_type_ids'], X_test_encoded['attention_mas
    y_test
)
print(f'Test loss: {test_loss}, Test accuracy: {test_accuracy}')
```

Save the model and tokenizer to the local folder

Python3

```
path = 'path-to-save'
# Save tokenizer
tokenizer.save_pretrained(path +'/Tokenizer')

# Save model
model.save_pretrained(path +'/Model')
```

Load the model and tokenizer from the local folder

Python3

```
# Load tokenizer
bert_tokenizer = BertTokenizer.from_pretrained(path +'/Tokenizer')

# Load model
bert_model = TFBertForSequenceClassification.from_pretrained(path +'/Model')
```

Predict the sentiment of the test dataset

Python3

```
pred = bert_model.predict(
    [X_test_encoded['input_ids'], X_test_encoded['token_type_ids'], X_test_encoded['attention_mas

# pred is of type TFSequenceClassifierOutput
logits = pred.logits

# Use argmax along the appropriate axis to get the predicted labels
pred_labels = tf.argmax(logits, axis=1)

# Convert the predicted labels to a NumPy array
pred_labels = pred_labels.numpy()

label = {
    1: 'positive',
    0: 'Negative'
}

# Map the predicted labels to their corresponding strings using the label dictionary
pred_labels = [label[i] for i in pred_labels]
Actual = [label[i] for i in y_test]

print('Predicted Label :', pred_labels[:10])
print('Actual Label    :', Actual[:10])
```

Step 7: Prediction with user inputs

Python3

```
def Get_sentiment(Review, Tokenizer=bert_tokenizer, Model=bert_model):
    # Convert Review to a list if it's not already a list
    if not isinstance(Review, list):
        Review = [Review]

    Input_ids, Token_type_ids, Attention_mask = Tokenizer.batch_encode_plus(Review,
                                                                            padding=True,
                                                                            truncation=True,
                                                                            max_length=128,
                                                                            return_tensors='tf')
    prediction = Model.predict([Input_ids, Token_type_ids, Attention_mask])

    # Use argmax along the appropriate axis to get the predicted labels
    pred_labels = tf.argmax(prediction.logits, axis=1)

    # Convert the TensorFlow tensor to a NumPy array and then to a list to get the predicted sent
    pred_labels = [label[i] for i in pred_labels.numpy().tolist()]
    return pred_labels
```

RoBERTa model:

RoBERTa (short for "Robustly Optimized BERT Approach") is a variant of the BERT (Bidirectional Encoder Representations from Transformers) model, which was developed by researchers at Facebook AI. Like BERT, RoBERTa is a transformer-based language model that uses self-attention to process input sequences and generate contextualized representations of words in a sentence.

One key difference between RoBERTa and BERT is that RoBERTa was trained on a much larger dataset and using a more effective training procedure. In particular, RoBERTa was trained on a dataset of 160GB of text, which is more than 10 times larger than the dataset used to train BERT. Additionally, RoBERTa uses a dynamic masking technique during training that helps the model learn more robust and generalizable representations of words.

RoBERTa has been shown to outperform BERT and other state-of-the-art models on a variety of natural language processing tasks, including language translation, text classification, and question answering. It has also been used as a base model for many other successful NLP models and has become a popular choice for research and industry applications.

Overall, RoBERTa is a powerful and effective language model that has made significant contributions to the field of NLP and has helped to drive progress in a wide range of applications.

RoBERTa stands for Robustly Optimized BERT Pre-training Approach. It was presented by researchers at Facebook and Washington University. The goal of this paper was to optimize the training of BERT architecture in order to take lesser time during pre-training.

Modifications to BERT:

RoBERTa has almost similar architecture as compare to BERT, but in order to improve the results on BERT architecture, the authors made some simple design changes in its architecture and training procedure. These changes are:

Removing the Next Sentence Prediction (NSP) objective: In the next sentence prediction, the model is trained to predict whether the observed document segments come from the same or distinct documents via an auxiliary Next Sentence Prediction (NSP) loss. The authors experimented with removing/adding of

NSP loss to different versions and concluded that removing the NSP loss matches or slightly improves downstream task performance

Training with bigger batch sizes & longer sequences: Originally BERT is trained for 1M steps with a batch size of 256 sequences. In this paper, the authors trained the model with 125 steps of 2K sequences and 31K steps with 8k sequences of batch size. This has two advantages, the large batches improves perplexity on masked language modelling objective and as well as end-task accuracy. Large batches are also easier to parallelize via distributed parallel training.

Dynamically changing the masking pattern: In BERT architecture, the masking is performed once during data preprocessing, resulting in a single static mask. To avoid using the single static mask, training data is duplicated and masked 10 times, each time with a different mask strategy over 40 epochs thus having 4 epochs with the same mask. This strategy is compared with dynamic masking in which different masking is generated every time we pass data into the model.

Datasets Used:

The following are the datasets used to train ROBERTa model:

1 . BOOK CORPUS and English Wikipedia dataset: This data also used for training BERT architecture, this data contains 16GB of text.

2 . CC-NEWS: This data contains 63 million English news articles crawled between September 2016 and February 2019. The size of this dataset is 76 GB after filtering.

3 . OPENWEBTEXT: This dataset contains web content extracted from the URLs shared on Reddit with at least 3 upvotes. The size of this dataset is 38 GB.

4. STORIES: This dataset contains a subset of Common Crawl data filtered to match the story-like style of Winograd NLP task. This dataset contains 31 GB of text.

Fine-Tuning RoBERTa model for sentiment analysis:

Step 1 : Importing python libraries and preparing the environment

Step 2 : Importing and pre- processing the domain data

Step 3 : preparing the dataset and dataloader

Step 4 : creating the Neural network for Fine Tuning

Step 5 : Fine Tuning the model

Step 5 : Validating the model performance

Step 6 : Saving the model and artifacts for Interferene in future

Importing Python Libraries and preparing the environment

At this step we will be importing the libraries and modules needed to run our script. Libraries are:

- Pandas
- Pytorch
- Pytorch Utils for Dataset and Dataloader
- Transformers
- tqdm
- sklearn
- Robert Model and Tokenizer

Followed by that we will preapre the device for CUDA exeeceution. This configuration is needed if you want to leverage on onboard GPU.

In [ ]:

```
!pip install transformers==3.0.2
```

In [ ]:

```
# Importing the libraries needed
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import torch
import seaborn as sns
import transformers
import json
from tqdm import tqdm
from torch.utils.data import Dataset, DataLoader
from transformers import RobertaModel, RobertaTokenizer
import logging
logging.basicConfig(level=logging.ERROR)
```

In [ ]:

```
# Setting up the device for GPU usage

from torch import cuda
device = 'cuda' if cuda.is_available() else 'cpu'
```

In [ ]:

```
train = pd.read_csv('train.tsv', delimiter='\t')
```

In [ ]:

```
train.shape
```

Out[ ]:

```
(156060, 4)
```

In [ ]:

```
train.head()
```

Out[ ]:

	PhraseId	SentenceId	Phrase	Sentiment
0	1	1	A series of escapades demonstrating the adage ...	1
1	2	1	A series of escapades demonstrating the adage ...	2
2	3	1	A series	2
3	4	1	A	2
4	5	1	series	2

In [ ]:

```
train['Sentiment'].unique()
```

Out[ ]:

```
array([1, 2, 3, 4, 0])
```

In [ ]:

```
train.describe()
```

Out[ ]:

	PhraseId	SentenceId	Sentiment
count	156060.000000	156060.000000	156060.000000
mean	78030.500000	4079.732744	2.063578
std	45050.785842	2502.764394	0.893832
min	1.000000	1.000000	0.000000
25%	39015.750000	1861.750000	2.000000
50%	78030.500000	4017.000000	2.000000
75%	117045.250000	6244.000000	3.000000

	PhraseId	SentenceId	Sentiment
max	156060.000000	8544.000000	4.000000

In [ ]:

```
new_df = train[['Phrase', 'Sentiment']]
```

Preparing the Dataset and Dataloader

I will start with defining few key variables that will be used later during the training/fine tuning stage. Followed by creation of Dataset class - This defines how the text is pre-processed before sending it to the neural network. I will also define the Dataloader that will feed the data in batches to the neural network for suitable training and processing. Dataset and Dataloader are constructs of the PyTorch library for defining and controlling the data pre-processing and its passage to neural network.

SentimentData Dataset Class

- This class is defined to accept the Dataframe as input and generate tokenized output that is used by the Roberta model for training.
- I am using the Roberta tokenizer to tokenize the data in the TITLE column of the dataframe.
- The tokenizer uses the encode_plus method to perform tokenization and generate the necessary outputs, namely: ids, attention_mask
- target is the encoded category on the news headline.
- The *SentimentData* class is used to create 2 datasets, for training and for validation.
- *Training Dataset* is used to fine tune the model: **80% of the original data**
- *Validation Dataset* is used to evaluate the performance of the model. The model has not seen this data during training.

Dataloader

- Dataloader is used to for creating training and validation dataloader that load data to the neural network in a defined manner. This is needed because all the data from the dataset cannot be loaded to the memory at once, hence the amount of dataloaded to the memory and then passed to the neural network needs to be controlled.
- This control is achieved using the parameters such as batch_size and max_len.
- Training and Validation dataloaders are used in the training and validation part of the flow respectively

In [ ]:

```
# Defining some key variables that will be used later on in the training
MAX_LEN = 256
TRAIN_BATCH_SIZE = 8
VALID_BATCH_SIZE = 4
# EPOCHS = 1
LEARNING_RATE = 1e-05
tokenizer = RobertaTokenizer.from_pretrained('roberta-base', truncation=True,
do_lower_case=True)
```

```
HBox(children=(FloatProgress(value=0.0, description='Downloading', max=898823
.0, style=ProgressStyle(descripti…
```

```
HBox(children=(FloatProgress(value=0.0, description='Downloading', max=456318
.0, style=ProgressStyle(descripti…
```

In [ ]:

```
class SentimentData(Dataset):
    def __init__(self, dataframe, tokenizer, max_len):
        self.tokenizer = tokenizer
        self.data = dataframe
        self.text = dataframe.Phrase
        self.targets = self.data.Sentiment
        self.max_len = max_len

    def __len__(self):
        return len(self.text)

    def __getitem__(self, index):
        text = str(self.text[index])
        text = " ".join(text.split())

        inputs = self.tokenizer.encode_plus(
            text,
            None,
            add_special_tokens=True,
            max_length=self.max_len,
            pad_to_max_length=True,
            return_token_type_ids=True
        )
        ids = inputs['input_ids']
        mask = inputs['attention_mask']
        token_type_ids = inputs["token_type_ids"]


        return {
            'ids': torch.tensor(ids, dtype=torch.long),
            'mask': torch.tensor(mask, dtype=torch.long),
            'token_type_ids': torch.tensor(token_type_ids, dtype=torch.long),
            'targets': torch.tensor(self.targets[index], dtype=torch.float)
        }
```

In [ ]:

```
train_size = 0.8
train_data=new_df.sample(frac=train_size,random_state=200)
test_data=new_df.drop(train_data.index).reset_index(drop=True)
train_data = train_data.reset_index(drop=True)


print("FULL Dataset: {}".format(new_df.shape))
print("TRAIN Dataset: {}".format(train_data.shape))
print("TEST Dataset: {}".format(test_data.shape))

training_set = SentimentData(train_data, tokenizer, MAX_LEN)
testing_set = SentimentData(test_data, tokenizer, MAX_LEN)
```

```
FULL Dataset: (156060, 2)
TRAIN Dataset: (124848, 2)
```

```
TEST Dataset: (31212, 2)
```

In [ ]:

```
train_params = {'batch_size': TRAIN_BATCH_SIZE,
                'shuffle': True,
                'num_workers': 0
                }

test_params = {'batch_size': VALID_BATCH_SIZE,
                'shuffle': True,
                'num_workers': 0
                }

training_loader = DataLoader(training_set, **train_params)
testing_loader = DataLoader(testing_set, **test_params)
```

Creating the Neural Network for Fine Tuning

Neural Network

- We will be creating a neural network with the RobertaClass.
- This network will have the Roberta Language model followed by a dropout and finally a Linear layer to obtain the final outputs.
- The data will be fed to the Roberta Language model as defined in the dataset.
- Final layer outputs is what will be compared to the Sentiment category to determine the accuracy of models prediction.
- We will initiate an instance of the network called model. This instance will be used for training and then to save the final trained model for future inference.

Loss Function and Optimizer

- Loss Function and Optimizer and defined in the next cell.
- The Loss Function is used the calculate the difference in the output created by the model and the actual output.
- Optimizer is used to update the weights of the neural network to improve its performance.

In [ ]:

```
class RobertaClass(torch.nn.Module):
    def __init__(self):
        super(RobertaClass, self).__init__()
        self.l1 = RobertaModel.from_pretrained("roberta-base")
        self.pre_classifier = torch.nn.Linear(768, 768)
        self.dropout = torch.nn.Dropout(0.3)
        self.classifier = torch.nn.Linear(768, 5)

    def forward(self, input_ids, attention_mask, token_type_ids):
        output_1 = self.l1(input_ids=input_ids,
attention_mask=attention_mask, token_type_ids=token_type_ids)
        hidden_state = output_1[0]
        pooler = hidden_state[:, 0]
        pooler = self.pre_classifier(pooler)
        pooler = torch.nn.ReLU()(pooler)
```

```
        pooler = self.dropout(pooler)
        output = self.classifier(pooler)
        return output
```

In [ ]:

```
model = RobertaClass()
model.to(device)
```

Out[ ]:

```
RobertaClass(
  (l1): RobertaModel(
    (embeddings): RobertaEmbeddings(
      (word_embeddings): Embedding(50265, 768, padding_idx=1)
      (position_embeddings): Embedding(514, 768, padding_idx=1)
      (token_type_embeddings): Embedding(1, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (1): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
```

```
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (2): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (3): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
```

```
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (4): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (5): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
```

```
            (intermediate): BertIntermediate(
              (dense): Linear(in_features=768, out_features=3072, bias=True)
            )
            (output): BertOutput(
              (dense): Linear(in_features=3072, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (6): BertLayer(
            (attention): BertAttention(
              (self): BertSelfAttention(
                (query): Linear(in_features=768, out_features=768, bias=True)
                (key): Linear(in_features=768, out_features=768, bias=True)
                (value): Linear(in_features=768, out_features=768, bias=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
              (output): BertSelfOutput(
                (dense): Linear(in_features=768, out_features=768, bias=True)
                (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
            (intermediate): BertIntermediate(
              (dense): Linear(in_features=768, out_features=3072, bias=True)
            )
            (output): BertOutput(
              (dense): Linear(in_features=3072, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (7): BertLayer(
            (attention): BertAttention(
              (self): BertSelfAttention(
                (query): Linear(in_features=768, out_features=768, bias=True)
                (key): Linear(in_features=768, out_features=768, bias=True)
                (value): Linear(in_features=768, out_features=768, bias=True)
                (dropout): Dropout(p=0.1, inplace=False)
              )
              (output): BertSelfOutput(
                (dense): Linear(in_features=768, out_features=768, bias=True)
                (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
            (intermediate): BertIntermediate(
              (dense): Linear(in_features=768, out_features=3072, bias=True)
            )
```

```
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (8): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (9): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
```

```
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (10): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (11): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=Tr
ue)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True
)
            (dropout): Dropout(p=0.1, inplace=False)
```

```
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (pre_classifier): Linear(in_features=768, out_features=768, bias=True)
  (dropout): Dropout(p=0.3, inplace=False)
  (classifier): Linear(in_features=768, out_features=5, bias=True)
)
```

Fine Tuning the Model

After all the effort of loading and preparing the data and datasets, creating the model and defining its loss and optimizer. This is probably the easier steps in the process.

Here we define a training function that trains the model on the training dataset created above, specified number of times (EPOCH), An epoch defines how many times the complete data will be passed through the network.

Following events happen in this function to fine tune the neural network:

- The dataloader passes data to the model based on the batch size.
- Subsequent output from the model and the actual category are compared to calculate the loss.
- Loss value is used to optimize the weights of the neurons in the network.
- After every 5000 steps the loss value is printed in the console.

As you can see just in 1 epoch by the final step the model was working with a loss of 0.8141926634122427.

In [ ]:

```
# Creating the loss function and optimizer
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(params =  model.parameters(), lr=LEARNING_RATE)
```

In [ ]:

```
def calcuate_accuracy(preds, targets):
    n_correct = (preds==targets).sum().item()
    return n_correct
```

In [ ]:

```
# Defining the training function on the 80% of the dataset for tuning the
distilbert model

def train(epoch):
    tr_loss = 0
    n_correct = 0
    nb_tr_steps = 0
    nb_tr_examples = 0
    model.train()
    for _,data in tqdm(enumerate(training_loader, 0)):
        ids = data['ids'].to(device, dtype = torch.long)
```

```
        mask = data['mask'].to(device, dtype = torch.long)
        token_type_ids = data['token_type_ids'].to(device, dtype =
torch.long)
        targets = data['targets'].to(device, dtype = torch.long)

        outputs = model(ids, mask, token_type_ids)
        loss = loss_function(outputs, targets)
        tr_loss += loss.item()
        big_val, big_idx = torch.max(outputs.data, dim=1)
        n_correct += calcuate_accuracy(big_idx, targets)

        nb_tr_steps += 1
        nb_tr_examples+=targets.size(0)

        if _%5000==0:
            loss_step = tr_loss/nb_tr_steps
            accu_step = (n_correct*100)/nb_tr_examples
            print(f"Training Loss per 5000 steps: {loss_step}")
            print(f"Training Accuracy per 5000 steps: {accu_step}")

        optimizer.zero_grad()
        loss.backward()
        # # When using GPU
        optimizer.step()

    print(f'The Total Accuracy for Epoch {epoch}:
{(n_correct*100)/nb_tr_examples}')
    epoch_loss = tr_loss/nb_tr_steps
    epoch_accu = (n_correct*100)/nb_tr_examples
    print(f"Training Loss Epoch: {epoch_loss}")
    print(f"Training Accuracy Epoch: {epoch_accu}")

    return
```

In [ ]:

```
EPOCHS = 1
for epoch in range(EPOCHS):
    train(epoch)
```

```
1it [00:00,  4.04it/s]
Training Loss per 5000 steps: 1.2416878938674927
Training Accuracy per 5000 steps: 62.5
5001it [18:40,  4.42it/s]
Training Loss per 5000 steps: 0.8735729315070672
Training Accuracy per 5000 steps: 64.37212557488502
10001it [37:22,  4.40it/s]
Training Loss per 5000 steps: 0.8366646968724489
Training Accuracy per 5000 steps: 65.3947105289471
15001it [56:03,  4.43it/s]
Training Loss per 5000 steps: 0.8169289541139411
Training Accuracy per 5000 steps: 66.141423905073
15606it [58:18,  4.46it/s]
The Total Accuracy for Epoch 0: 66.24695629885942
Training Loss Epoch: 0.8141926634122427
```

```
Training Accuracy Epoch: 66.24695629885942
```

Validating the Model

During the validation stage we pass the unseen data(Testing Dataset) to the model. This step determines how good the model performs on the unseen data.

This unseen data is the 20% of train.tsv which was seperated during the Dataset creation stage. During the validation stage the weights of the model are not updated. Only the final output is compared to the actual value. This comparison is then used to calcuate the accuracy of the model.

As you can see the model is predicting the correct category of a given sample to a 69.47% accuracy which can further be improved by training more.

In [ ]:

```
def valid(model, testing_loader):
    model.eval()
    n_correct = 0; n_wrong = 0; total = 0; tr_loss=0; nb_tr_steps=0;
nb_tr_examples=0
    with torch.no_grad():
        for _, data in tqdm(enumerate(testing_loader, 0)):
            ids = data['ids'].to(device, dtype = torch.long)
            mask = data['mask'].to(device, dtype = torch.long)
            token_type_ids = data['token_type_ids'].to(device,
dtype=torch.long)
            targets = data['targets'].to(device, dtype = torch.long)
            outputs = model(ids, mask, token_type_ids).squeeze()
            loss = loss_function(outputs, targets)
            tr_loss += loss.item()
            big_val, big_idx = torch.max(outputs.data, dim=1)
            n_correct += calcuate_accuracy(big_idx, targets)

            nb_tr_steps += 1
            nb_tr_examples+=targets.size(0)

            if _%5000==0:
                loss_step = tr_loss/nb_tr_steps
                accu_step = (n_correct*100)/nb_tr_examples
                print(f"Validation Loss per 100 steps: {loss_step}")
                print(f"Validation Accuracy per 100 steps: {accu_step}")
    epoch_loss = tr_loss/nb_tr_steps
    epoch_accu = (n_correct*100)/nb_tr_examples
    print(f"Validation Loss Epoch: {epoch_loss}")
    print(f"Validation Accuracy Epoch: {epoch_accu}")

    return epoch_accu
```

In [ ]:

```
acc = valid(model, testing_loader)
print("Accuracy on test data = %0.2f%%" % acc)
```

```
3it [00:00, 23.17it/s]
Validation Loss per 100 steps: 0.6547155380249023
Validation Accuracy per 100 steps: 75.0
```

```
5004it [03:13, 25.87it/s]
Validation Loss per 100 steps: 0.736690901120772
Validation Accuracy per 100 steps: 69.22115576884623
7803it [05:01, 25.89it/s]
Validation Loss Epoch: 0.7332612214877096
Validation Accuracy Epoch: 69.46687171600666
Accuracy on test data = 69.47%
```

Saving the Trained Model Artifacts for inference

This is the final step in the process of fine tuning the model.

The model and its vocabulary are saved locally. These files are then used in the future to make inference on new inputs of news headlines.

In [ ]:

```
output_model_file = 'pytorch_roberta_sentiment.bin'
output_vocab_file = './'

model_to_save = model
torch.save(model_to_save, output_model_file)
tokenizer.save_vocabulary(output_vocab_file)

print('All files saved')
print('This tutorial is completed')
```

```
All files saved
This tutorial is completed
```