ST449 Project

Sentiment Analysis using RNNs and Transformer-Based Models

Candidate Number: 14565

Abstract

In this paper we analyse the performance of different types of machine learning algorithms at classifying sentiment in text data. We use traditional Bag-of-Words based model -Naive Bayes, word2vec based model – Long Short-Term Memory Neural Network and transformer-based models- ALBERT, BERT-Expert and ELECTRA. We define the architecture of the different models in detail, apply them to classify real world data and compare their performance and accuracy again each other.

I. Introduction

Sentiment analysis is an application of natural language processing. It is a supervised machine learning task that attempts to predict emotion from text. In today's world, sentiment has a myriad of real-world applications in marketing, politics and forecasting. It analyses text to provide valuable insights that cannot be discerned from numerical data. Over time many different kinds of algorithms have been used to perform sentiment analysis. Some of the older algorithms follow a rule-based approach, such as the Naive Bayes Algorithm. More recent developments in the machine learning field have led to the rise of Deep Learning algorithms to classify sentiment. The BERT model introduced by Google in 2018, has provided state-of-the-art results for many natural language processing tasks. In addition to this, recurrent neural networks have also been shown to perform well for solving NLP tasks, given their ability to incorporate previous information to inform their future decisions.

○ Data

In this paper, we analyse the performance of various sentiment analysis algorithms on the task of classifying IMDB movie reviews. The dataset is made available by Stanford's AI Lab. It is a balanced dataset that contains 50,000 movie reviews. Each review is given a label 1 if it is a positive review and a label of 0 if it is a negative review. We start our analysis by using a Multinomial Naive Bayes Model for the classification task. We then use a Long Short Term Memory Neural Network, ALBERT, BERT-Expert and ELECTRA to classify our empirical data. The focus remains on whether the new variations of the BERT model such as ALBERT, BERT-Expert and ELECTRA offer improvements in prediction accuracy and computational time when compared again RNN's and simpler algorithms such as Naive Bayes.

II. Naive Bayes

i. Model Introduction

One of the oldest algorithms to be used for sentiment analysis is the Naive Bayes. The algorithm is based on the bayes theorem, which says that conditional probability can be decomposed into:

$$p(C_k|x) = \frac{p(x|C_k)\, p(C_k)}{p(x)}$$

There is also an assumption of conditional independence, which means that in the context of looking a movie reviews, the occurrence of one word in a review does not affect the probability of seeing any other word in the review. The Naive Bayes Algorithm uses a Bag-of-Words Model to classify text. The model assigns a unique number to each word in the text corpus and then represents each review as a numerical vector. By using this model, Naive Bayes ignores all grammar rules, context and order of words when performing its classification. This is in stark contrast to models like BERT which take context into account when performing classification.

Let's refer to each review as a feature and each sentiment category as a label. Using the Bayes Theorem, the probability of belonging to a label is:

$$P(L|features) = \frac{P(features|L)P(L)}{P(features)}$$

As we have two labels – negative and positive, one way to classify the reviews would be to compute the posterior.

$$\frac{P(L_1|features)}{P(L_2|features)} = \frac{P(features|L_1)P(L_1)}{P(features|L_2)P(L_2)}$$

The "naive" in Naive Bayes implies that the features are conditionally independent. So, we have that:

$$P(features|L_1) = P(feature_1|L_1) * P(feature_2|L_1) * \ldots * P(feature_n|L_1)$$

Now we need to specify a generative distribution for $P(features|L_1)$. In this project. we will use the multinomial Naive Bayes Model which specifies a generative distribution based on the multinomial distribution. Thus, the probability $P(feature_1|L_1)$ is based on word count frequency. One problem arises when we calculate likelihood this way. If feature 1 is not present in label 1, then the probability $P(feature_1|L_1)$ will be to zero. This would make the entire likelihood zero which means that the likelihood contributions from other features will be discarded. To circumvent this problem, Naive Bayes usually adds a pseudo count to all word frequencies. (Kibriya, Frank, Pfahringer, & Holmes, 2004)

ii. Data Processing and Model Fit

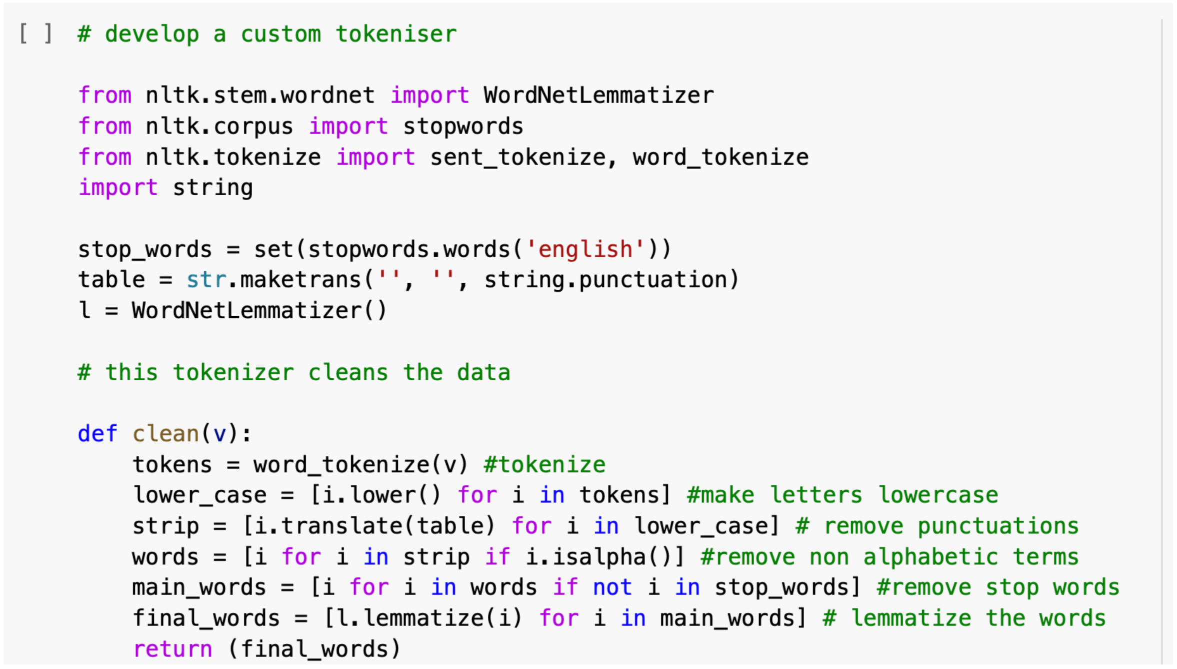
We start by importing the imdb reviews as a pandas dataframe and splitting the data into traiing and testing.

```
#import data as a tfds dataset
dataset, info = tfds.load('imdb_reviews', with_info=True,
                          as_supervised=True)

#convert to a pandas dataframe and split into train and test
pd_dataset_train = tfds.as_dataframe(dataset['train'], info)
pd_dataset_test = tfds.as_dataframe(dataset['test'], info)
```

```
[ ] #take a look at the first few rows for training data
    pd_dataset_train.head
```

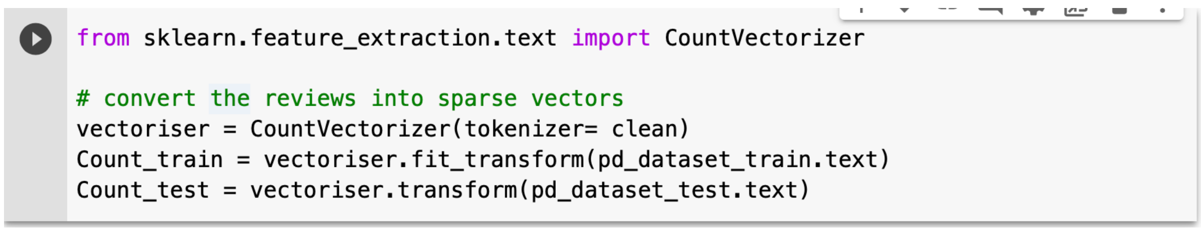
```
<bound method NDFrame.head of        label                                              text
0          0  b"This was an absolutely terrible movie. Don't...
1          0  b'I have been known to fall asleep during film...
2          0  b'Mann photographs the Alberta Rocky Mountains...
3          1  b'This is the kind of film for a snowy Sunday ...
4          1  b'As others have mentioned, all the women that...
...      ...                                                ...
24995      0  b'I have a severe problem with this show, seve...
24996      1  b'The year is 1964. Ernesto "Che" Guevara, hav...
24997      0  b'Okay. So I just got back. Before I start my ...
24998      0  b'When I saw this trailer on TV I was surprise...
24999      1  b'First of all, Riget is wonderful. Good comed...

[25000 rows x 2 columns]>
```

Both the training and testing datasets contain 25,000 reviews each. We first need to clean and tokenize the reviews, so that the Naive Bayes classifier is able to focus on the relevant words when classifying a review. We start by defining a function named "clean" that takes a review as input. It tokenises the words in the review, makes all letters lowercase, removes punctuations and removes non-alphabetic terms. The function then removes "stopwords", common English words, from the review test. This is done because stop words do not contribute any information towards which category the review belongs to. We also lemmatize the review, which means that it groups all inflections of a word as the root word.



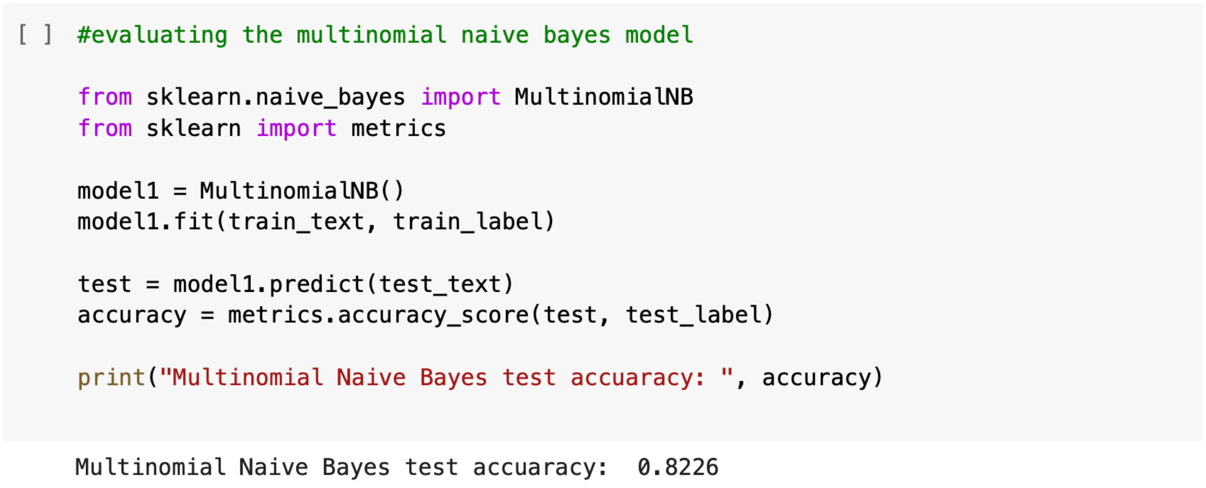
```
# develop a custom tokeniser

from nltk.stem.wordnet import WordNetLemmatizer
from nltk.corpus import stopwords
from nltk.tokenize import sent_tokenize, word_tokenize
import string

stop_words = set(stopwords.words('english'))
table = str.maketrans('', '', string.punctuation)
l = WordNetLemmatizer()

# this tokenizer cleans the data

def clean(v):
    tokens = word_tokenize(v) #tokenize
    lower_case = [i.lower() for i in tokens] #make letters lowercase
    strip = [i.translate(table) for i in lower_case] # remove punctuations
    words = [i for i in strip if i.isalpha()] #remove non alphabetic terms
    main_words = [i for i in words if not i in stop_words] #remove stop words
    final_words = [l.lemmatize(i) for i in main_words] # lemmatize the words
    return (final_words)
```

We then convert our reviews into sparse vectors using the CountVectorizer. It converts the text from the reviews into fixed length numerical vectors. Here the length of the vector is equal to the length of the vocabulary from the dataset. The value of each position in the vector represents the frequency with which the corresponding word occurs in the review. When building the CountVectorizer we specify our self-built function "clean" as the custom tokenizer. Thus, CountVectorizer cleans, tokenizes and vectorizes the inputs.



```
from sklearn.feature_extraction.text import CountVectorizer

# convert the reviews into sparse vectors
vectoriser = CountVectorizer(tokenizer= clean)
Count_train = vectoriser.fit_transform(pd_dataset_train.text)
Count_test = vectoriser.transform(pd_dataset_test.text)
```

We import the Multinomial Naive Bayes model from the sklearn library. We fit the model to training data and then evaluate its accuracy by predicting the labels associated with the testing data. The testing accuracy is 82%.



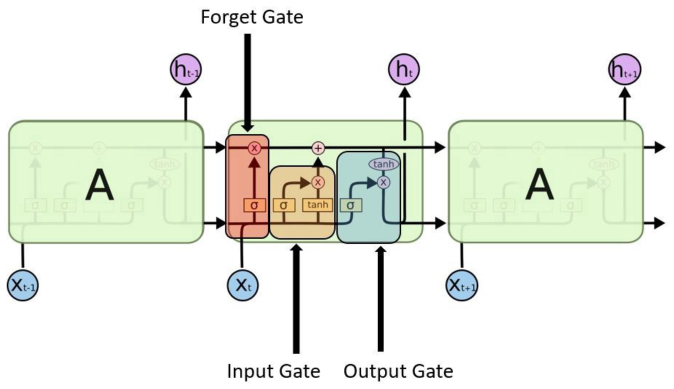
```
[ ] #evaluating the multinomial naive bayes model

    from sklearn.naive_bayes import MultinomialNB
    from sklearn import metrics

    model1 = MultinomialNB()
    model1.fit(train_text, train_label)

    test = model1.predict(test_text)
    accuracy = metrics.accuracy_score(test, test_label)

    print("Multinomial Naive Bayes test accuaracy: ", accuracy)
```

```
Multinomial Naive Bayes test accuaracy:  0.8226
```

III. LSTM Neural Networks

i. Introduction

Long Short-Term memory neural networks are a subset of Recurrent Neural Network that overcome the vanishing gradient problem often seen in recurrent neural networks. Due to their ability to process input based on their memory of previous classifications, they are particularly well suited to natural language processing tasks where the inputs are not independent of one-another. The following image shows the basic architecture of an LSTM network:



Between each LSTM cell, there are two states that are transferred from one cell to the next. The first of these states is the cell state. The cell state is the long-term memory of the network. Cell state is constantly modified by the input gate and the forget gate. The second state is the hidden state. Hidden state is the working memory which contains information from the immediate previous cell. The contents of the hidden state are overwritten by the output gate at each cell. (Minaee, Azimi, & Abdolrashidi, 2019)

So, each LSTM cell takes three inputs: the cell state, the last hidden state and the observation. Let's define some important terms first:

$$h_t = Hidden\ state\ at\ time\ t$$
$$C_t = Cell\ state\ at\ time\ t$$
$$X_t = Input\ at\ time\ t$$

These inputs then pass through the following gates:

Input gate – The main task of the input gate is to decide which values from the input should be stored in the long-term memory. For that it uses a sigmoid activation function and has a range between 0 and 1. The tanh function is used to allow the cell state to forget by assigning values weights from -1 to 1. The equations are as follows:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\widetilde{C_t} = \tanh\,(W_C \ \cdot [h_{t-1}, x_t] + b_C$$

Forget Gate – The main task of the forget gate is to decide which information should be forgotten from the cell state. This is done using the sigmoid activation function. The function is applied to the previous hidden state and the current input. For each number in the previous cell state, it gives out a 1 if the information is to be retained and it gives out a 0 if the information is to be discarded. The equations are as follows:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

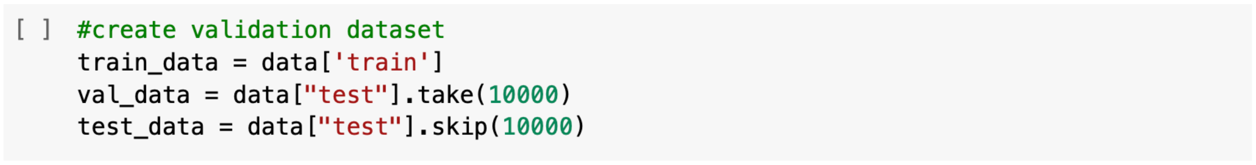
Output Gate – The main task of the output gate is to decide the output for the next cell. It uses the current input and the current cell state. The sigmoid activation function is used to decide the values to keep. The values then pass through the tanh function to be weighted according to their importance. The equations are as follows:

$$o_t = \sigma(W_o\ [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh\,(C_t)$$

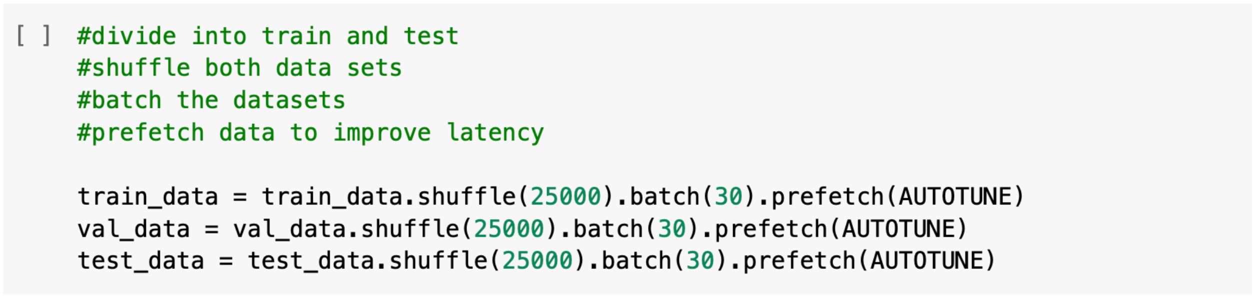
The first activation is the forget gate which decides which information should be forgotten from long term memory. We then move on to thee input gate which decides which new information should be added to the cell state. Then the output gate decides how the next hidden state is updated.

ii. Data Processing

We start by loading the data as a TensorFlow dataset. The next step is to shuffle and split the dataset into 25000 training reviews, 10,000 validation reviews and 15,000 testing reviews. Using prefetch allows the dataset to be read faster during training.

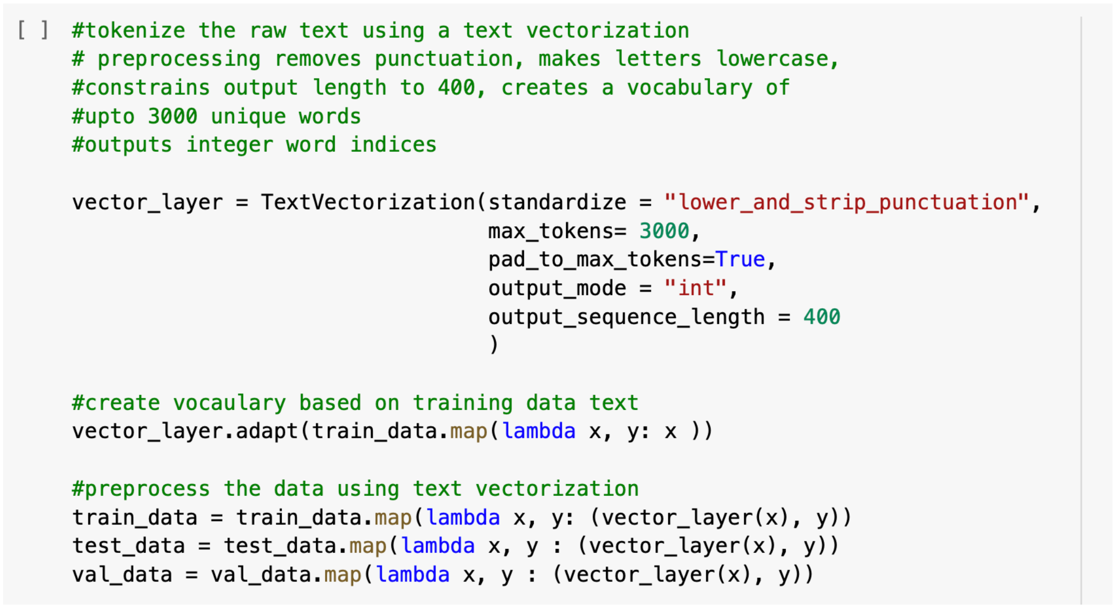


```
[ ]  #create validation dataset
     train_data = data['train']
     val_data = data["test"].take(10000)
     test_data = data["test"].skip(10000)
```



```
[ ]  #divide into train and test
     #shuffle both data sets
     #batch the datasets
     #prefetch data to improve latency

     train_data = train_data.shuffle(25000).batch(30).prefetch(AUTOTUNE)
     val_data = val_data.shuffle(25000).batch(30).prefetch(AUTOTUNE)
     test_data = test_data.shuffle(25000).batch(30).prefetch(AUTOTUNE)
```

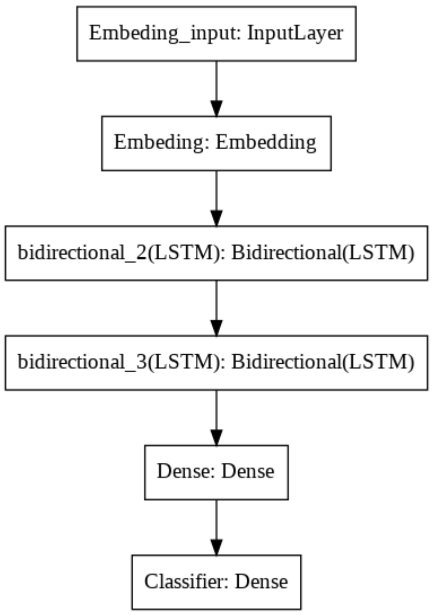
We then use TextVectorization to transform the text input into tokens. The reviews are truncated to the first 400 words to reduce computational burden. In the process of this transformation, it also makes all words lowercase and strips all the punctuation. By using the adapt() method on training data, we create a vocabulary of the 3000 most frequently used words in the training reviews. We then tokenize the training, validation and testing reviews based on this vocabulary.



```
[ ]  #tokenize the raw text using a text vectorization
     # preprocessing removes punctuation, makes letters lowercase,
     #constrains output length to 400, creates a vocabulary of
     #upto 3000 unique words
     #outputs integer word indices

     vector_layer = TextVectorization(standardize = "lower_and_strip_punctuation",
                                      max_tokens= 3000,
                                      pad_to_max_tokens=True,
                                      output_mode = "int",
                                      output_sequence_length = 400
                                      )

     #create vocaulary based on training data text
     vector_layer.adapt(train_data.map(lambda x, y: x ))

     #preprocess the data using text vectorization
     train_data = train_data.map(lambda x, y: (vector_layer(x), y))
     test_data = test_data.map(lambda x, y : (vector_layer(x), y))
     val_data = val_data.map(lambda x, y : (vector_layer(x), y))
```

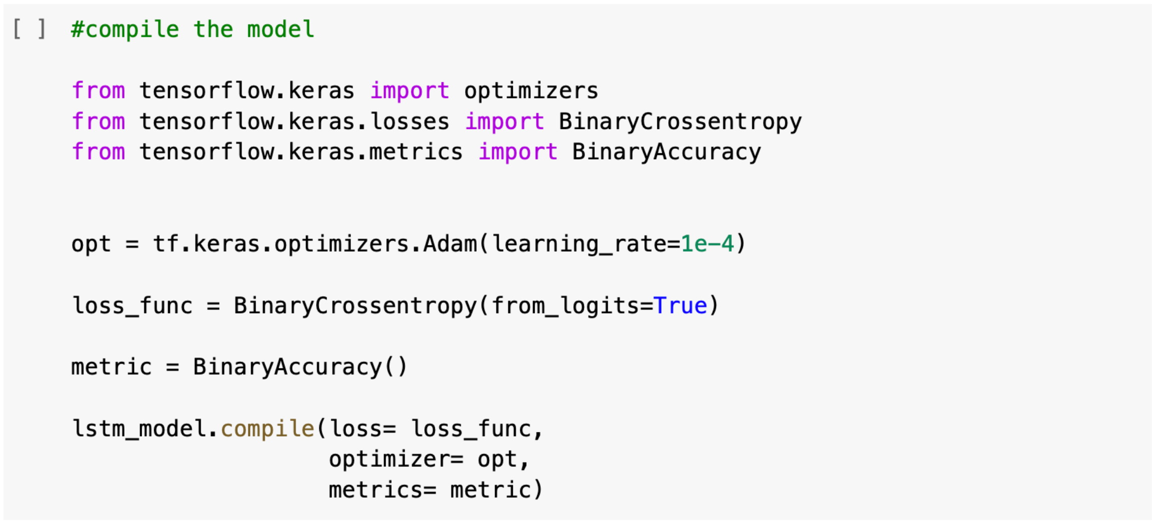
We then build the LSTM model as a Keras Seqential model. The first layer is an embedding layer that that converts the input sequences into sequence of trainable dense vectors. The size of each dense vector is the size of the vocabulary, which in our case is 3000. The next row layers are bidirectional LSTM layers. Bidirectional LSTM layers train LSTM twice, once on the given input sequence and then again on the reversed input sequence. This helps them gain a deeper understanding of the input context. The Bidirectional LSTM layers use Kernel regularization and dropout for regularization. Dropout prevents overfitting by randomly dropping some outputs during training. Kernel regularizer applies and L1 and L2 penalty to the layers kernel. The bidirectional layers are followed by a dense layer. Finally, the last layer is a classification layer with a sigmoid activation.

<u>LSTM Architecture</u>


Embeding_input: InputLayer
Embeding: Embedding
bidirectional_2(LSTM): Bidirectional(LSTM)
bidirectional_3(LSTM): Bidirectional(LSTM)
Dense: Dense
Classifier: Dense

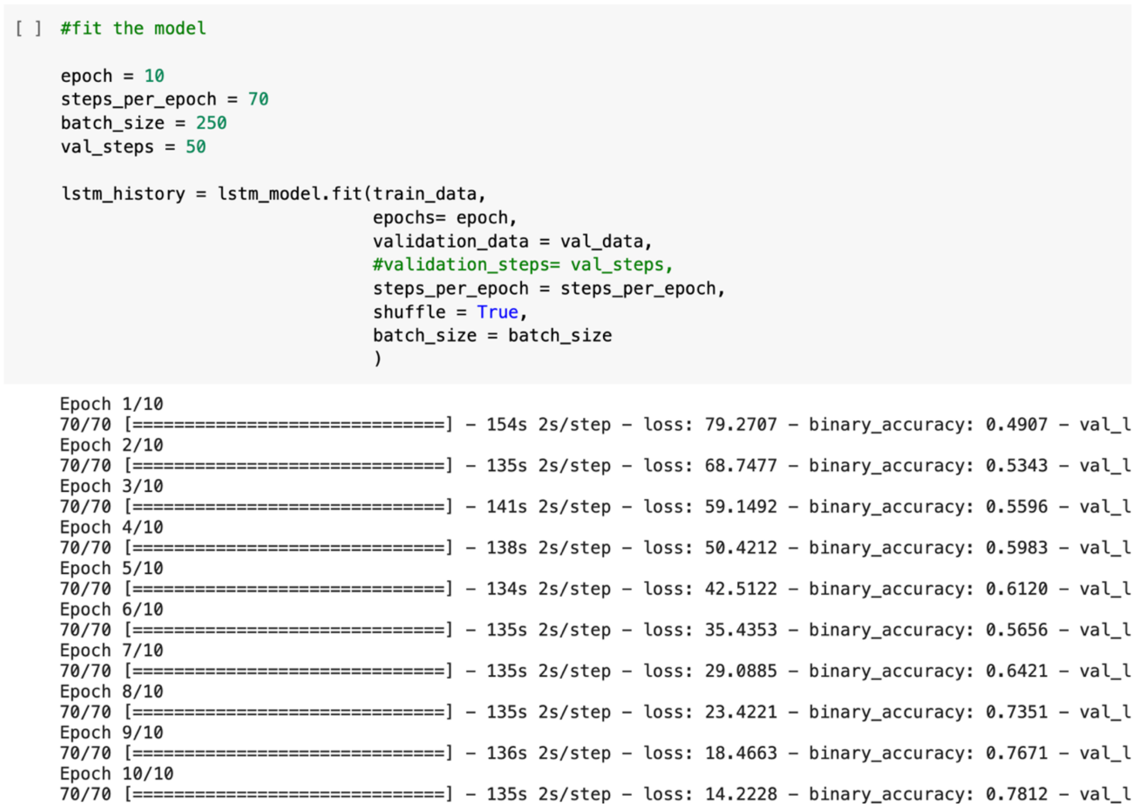
iii. Model Fit and Compile

We compile the model using Adam as the optimizer with an initial learning rate of 1e-4. Binary Cross entropy is used as the loss function given its good performance when training binary classification tasks. Binary Accuracy is used to measure the performance of the model.



```
[ ] #compile the model

    from tensorflow.keras import optimizers
    from tensorflow.keras.losses import BinaryCrossentropy
    from tensorflow.keras.metrics import BinaryAccuracy


    opt = tf.keras.optimizers.Adam(learning_rate=1e-4)

    loss_func = BinaryCrossentropy(from_logits=True)

    metric = BinaryAccuracy()

    lstm_model.compile(loss= loss_func,
                       optimizer= opt,
                       metrics= metric)
```

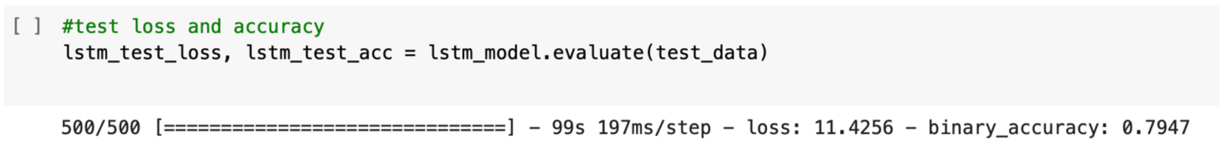
The model is trained for 10 epochs. The training data is split into batches of 250 each and each epoch trains for 70 steps. Validation is done for 50 steps to see how well the model is doing during training.



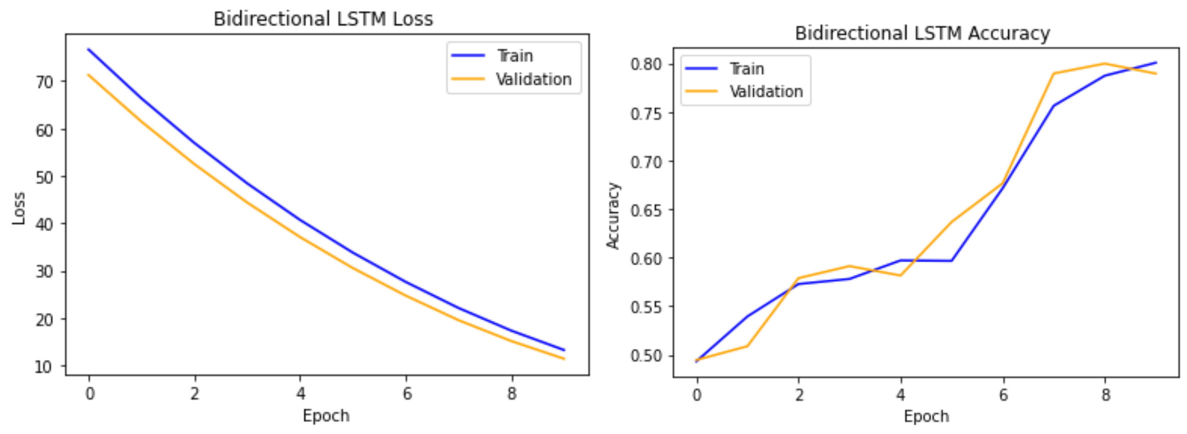
```
[ ] #fit the model

    epoch = 10
    steps_per_epoch = 70
    batch_size = 250
    val_steps = 50

    lstm_history = lstm_model.fit(train_data,
                                  epochs= epoch,
                                  validation_data = val_data,
                                  #validation_steps= val_steps,
                                  steps_per_epoch = steps_per_epoch,
                                  shuffle = True,
                                  batch_size = batch_size
                                  )
```

```
    Epoch 1/10
    70/70 [==============================] - 154s 2s/step - loss: 79.2707 - binary_accuracy: 0.4907 - val_l
    Epoch 2/10
    70/70 [==============================] - 135s 2s/step - loss: 68.7477 - binary_accuracy: 0.5343 - val_l
    Epoch 3/10
    70/70 [==============================] - 141s 2s/step - loss: 59.1492 - binary_accuracy: 0.5596 - val_l
    Epoch 4/10
    70/70 [==============================] - 138s 2s/step - loss: 50.4212 - binary_accuracy: 0.5983 - val_l
    Epoch 5/10
    70/70 [==============================] - 134s 2s/step - loss: 42.5122 - binary_accuracy: 0.6120 - val_l
    Epoch 6/10
    70/70 [==============================] - 135s 2s/step - loss: 35.4353 - binary_accuracy: 0.5656 - val_l
    Epoch 7/10
    70/70 [==============================] - 135s 2s/step - loss: 29.0885 - binary_accuracy: 0.6421 - val_l
    Epoch 8/10
    70/70 [==============================] - 135s 2s/step - loss: 23.4221 - binary_accuracy: 0.7351 - val_l
    Epoch 9/10
    70/70 [==============================] - 136s 2s/step - loss: 18.4663 - binary_accuracy: 0.7671 - val_l
    Epoch 10/10
    70/70 [==============================] - 135s 2s/step - loss: 14.2228 - binary_accuracy: 0.7812 - val_l
```

We then evaluate the model on testing data and plot the training accuracy and loss. The test accuracy is 79.4% and validation accuracy converges around 8 epochs.



```
[ ] #test loss and accuracy
    lstm_test_loss, lstm_test_acc = lstm_model.evaluate(test_data)

    500/500 [==============================] - 99s 197ms/step - loss: 11.4256 - binary_accuracy: 0.7947
```



IV. Transformer Based Models

i. BERT Architecture

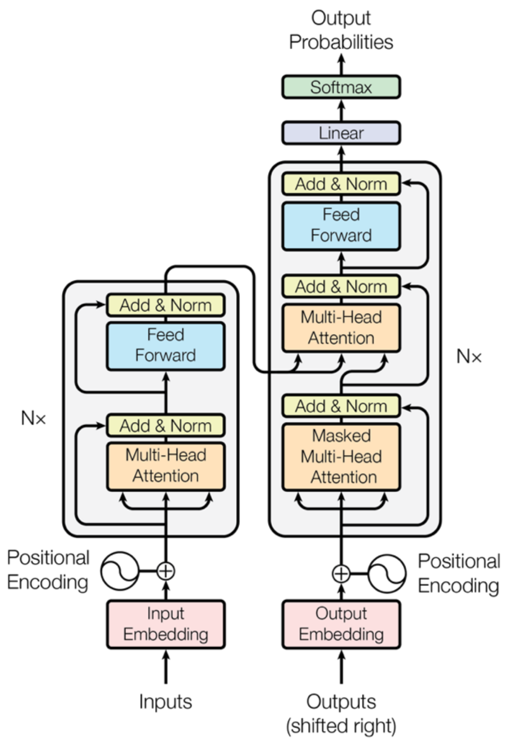
When the Bidirectional Encoder Representations from Transformers (BERT) was first released by Google AI in 2018, it was shown to improve performance for a variety of natural language processing tasks such as answering queries, sentiment classification, natural language inference etc. (Song, et al., 2020)

Devlin, et al. (2018) describe BERT as a model designed to pretrain deep bidirectional representations from unlabeled text by conditioning on both right and left context. Let's break down this definition.

Pretraining means that the BERT model has already been trained on a large amount of unlabelled data for different tasks. Pretraining word embeddings have been shown to significantly improve a model's performance when compared to a model that trains word embeddings from scratch. As we train our model on a large text corpus it develops a deeper understanding of language peculiarities. Thus, pretraining proves particularly crucial to BERT's functionality

Bidirectional means that unlike other deep learning models like ELMo which uses unidirectional training to grasp language representations, BERT computes language representation both left to right and right to left. Devlin, et al. (2018) stress the importance of bidirectional pre-training to truly grasp a deeper understanding of language.

Transformers form the foundation of the BERT model. Transformers are architectures that transform a sequence into another predicted sequence using an encoder and a decoder. They were first introduced by Vaswani, et al. (2017). It transforms sequences based on an attention mechanism which decides at each step which parts of the input sequence are important. Vaswani, et al. (2017) use the following image the explain the architecture of the transformer:



On the left, we have the encoder and, on the right, we have the decoder. The encoder consists of a multiheaded attention layer and a feed forward neural network. These encoders can be stacked on top of each other. Each encoder takes as input a list of vectors and maps it to an output sequence representation. The input sequence is first passed through a multi- head attention layer. This means that as the layer processes each word in the input sequence, multi-attention allows for the layer to look at the word from different positions and this helps find a better encoding for the word. The feed forward layer estimates hierarchical features for the model. The decoder contains an extra layer of masked multiheaded attention which helps it focus on the key parts of the inputs. Bert's purpose is to analyse the given input – it is not concerned with transforming the given input into another sequence. Thus, only transformer encoders are used in Bert's architecture.

To perform bidirectional training on the text inputs, Bert uses two strategies: Masked Language Models and Next sentence prediction. Masked Language Models (MLM), hide 15% of the tokens in each input sequence and replace them with the token [MASK]. The model then predicts the identity of the masked tokens based on the context provided by the not masked tokens. This gives the model the advantage of being bidirectional. BERT also uses next sentence prediction to gain a deeper understanding of word embedings. During NSP, the model receives a pair of sentences as inputs. Half the time the second sentence is the succeeding sentence to the first one. The other half of the time, the second sentence is random. The model then has to learn to predict if the second sentence follows the first one or not. We make the assumption that if the second sentence is random that it will not have a connection to the first sentence.

The base BERT architecture is built on transformers. The base bert model from Tensorflow is made up of 12 hidden layers (transformers layers), a hiddensize of 768 and 12 attention heads. This means that the bert base architecture is made up of 12 encoder layers, 12 attention heads and feedforward networks have 768 units.

Finally, the pretrained BERT model need to be "fine-tuned" for the particular classification task. This can be achieved by adding an additional output layer. This way we can achieve the efficiency of state-of-the-art models without substantial training.

ALERT, BERT- Expert and ELECTRA all apply different variations to the base bert architecture

- **ALBERT Architecture**

ALBERT is considered a lighter version of Google's BERT model. ALBERT makes two key changes to BERT's architecture to achiever impressive results. First, it factorises embedding parametrizations. Second, it implements cross layer parameter sharing. Thus, AlBERT reduces the number of parameters used, thereby making computation easier and decreasing the time taken for training Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., & Soricut, R. (2019).

First, let's focus on the factorization. At the input level, ALBERT splits the big vocabulary embedding matrix into two smaller matrices. The input layer embeddings are separated from the hidden layer embeddings. This reduces the number of parameters while casing only a small drop in performance. With regards to parameter sharing, it is often observed that in transformer based neural networks, have different layers that have to pick up how to perform similar operations. Albert eliminates this need by sharing parameters between the feed forward and attention layers.

For our implementation we use a base version of albert with 12 encoder layers, 12 attention heads and 768 feed forward network units. The base version of Albert has 12 million parameters as compared to 108 million parameters in the base Bert model.

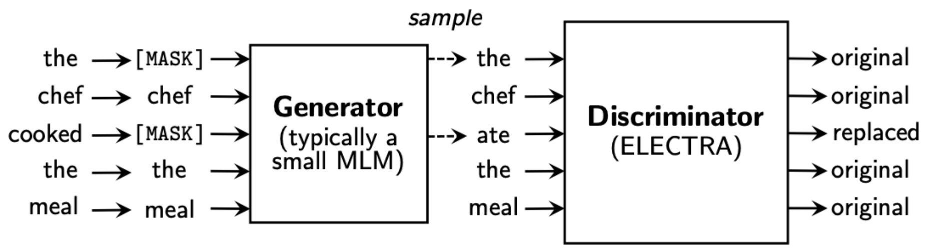
- **Bert Expert Architecture**

This model utilises the same architecture as the base Bert model, but some changes have been made to how the model is pre trained. The model is specifically trained for sentiment analysis on the Stanford sentiment tree bank dataset and English Wikipedia. Taking this into account, the model is expected to perform particularly well on sentiment analysis.

- **ELECTRA Architecture**

Electra stands for Efficiently Learning an Encoder that Classifies Token Replacements Accurately. The model uses the same architecture as the base BERT model but it uses a different method to pre train the model. It was first introduced by Clark, et al. (2020).

As we discussed before BERT uses masked language models and nest sentence prediction to pretrain the model. ELECTRA takes a different approach, as its name suggests, instead of MLM it uses "token replacements" for pretraining. The following image helps explain it better:



During pretraining, token replacement employs a network called a generator. Generator is typically a small MLM, that randomly replaces the masked tokens with either the original token value or a fake token. The ELECTRA model, is used as a discriminator and has to spot which tokens are false and which are true.

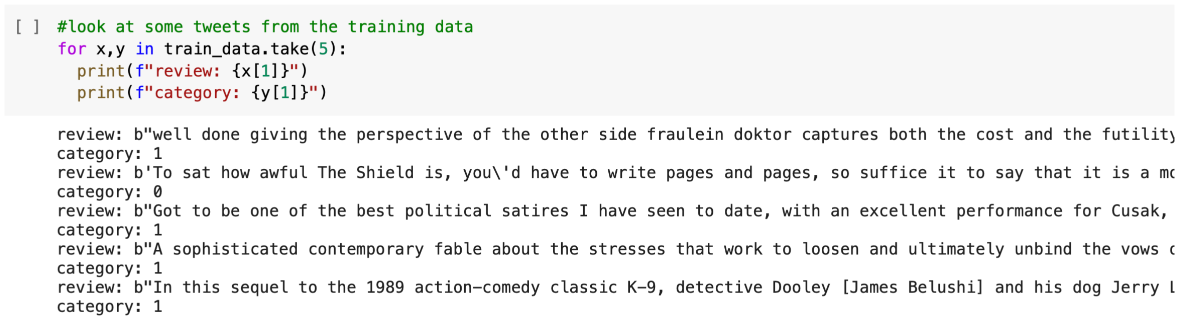
According to Clark, K., Luong et al. (2020), this approach to pre training helps the model perform better, because it forces the model to consider how genuine each input token is instead of focusing only on the masked tokens.

ii. Data Pre-Processing for Transformer Based Models

Before fitting any of the transformer-based models, it is important that the input text is in the correct format. BERT requires that all text input to be converted into numeric tensors. For this, we use preprocessing models from TensorFlow hub. Each transformer-based model comes with its own preprocessor that converts the input text into fixed length tensors. The preprocessor adds start and end padding tokens to each review and output three parts –

1. Input word ids – unique number assigned to each word
2. Input mask - distinguish padding from non-padding tokens
3. Input Type Ids – distinguish between different sentences in the same text

Once we import the IMDB dataset as a TensorFlow dataset, we split that data into training, testing and validation sets. 25,000 reviews are used for training, 10,000 are used for validation and 15,000 are reserved for testing. The following is a sample of some of the reviews:

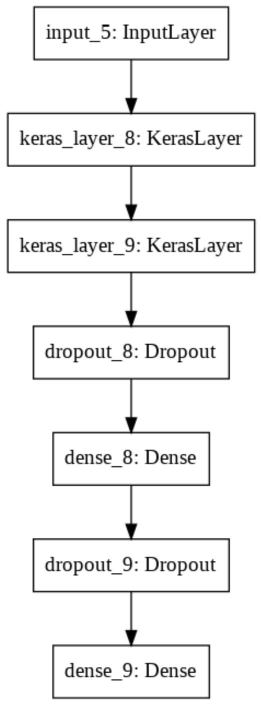


```
[ ] #look at some tweets from the training data
    for x,y in train_data.take(5):
      print(f"review: {x[1]}")
      print(f"category: {y[1]}")
```

```
review: b"well done giving the perspective of the other side fraulein doktor captures both the cost and the futility
category: 1
review: b'To sat how awful The Shield is, you\'d have to write pages and pages, so suffice it to say that it is a mo
category: 0
review: b"Got to be one of the best political satires I have seen to date, with an excellent performance for Cusak,
category: 1
review: b"A sophisticated contemporary fable about the stresses that work to loosen and ultimately unbind the vows o
category: 1
review: b"In this sequel to the 1989 action-comedy classic K-9, detective Dooley [James Belushi] and his dog Jerry L
category: 1
```

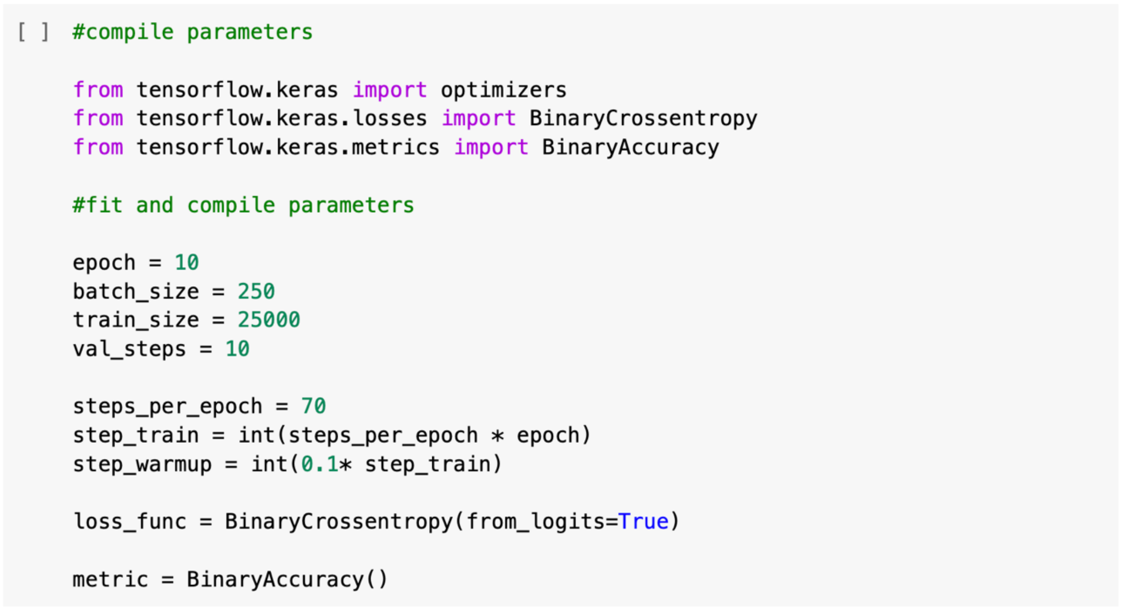
Since different kinds of transformer based models and preprocessors can be accessed using different URLs from TensorFlow Hub, we write a function called "bert_model" that takes a model url and a preprocessor url as input and output a fine-tuned transformer based model. In addition to an existing model, we add dropout layers to the architecture to prevent overfitting. We also add a dense layer and an output layer with sigmoid activation for the final classification. The following image encapsulates this model architecture:

Transformer Model Architecture



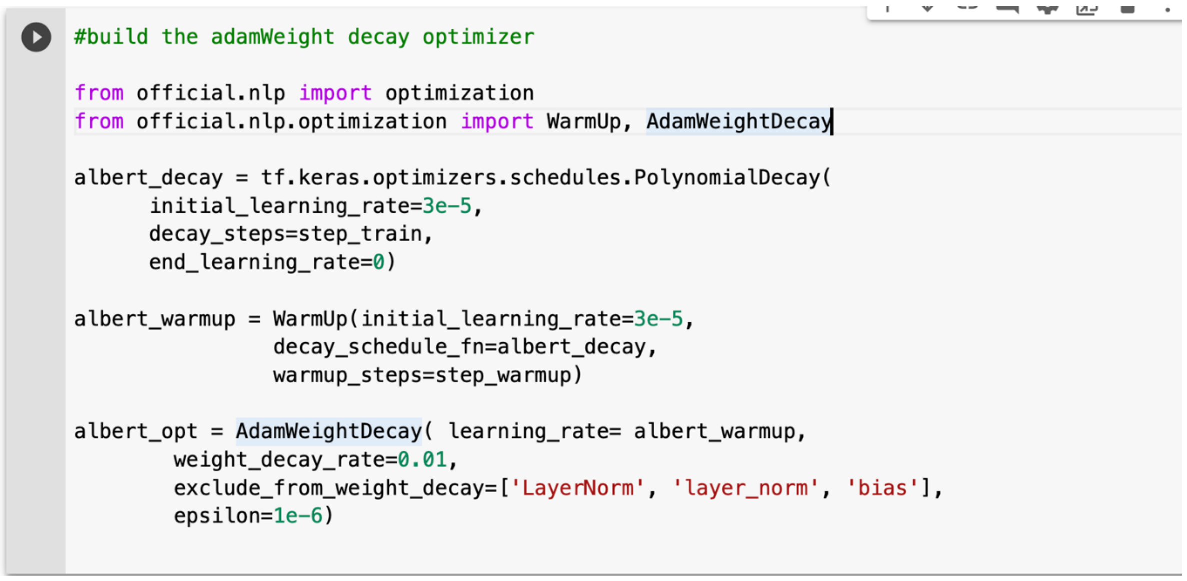
iii. Model Fit and compile

We train all our transformer-based models –for 10 epochs. We batch the data into sets of 250 each and run each epoch for 70 steps. Validation is done at the end of each epoch for 10 steps. This helps us see how well the algorithm is actually learning. We also define the number of training and warmup steps which are needed in building up the AdamW optimizer



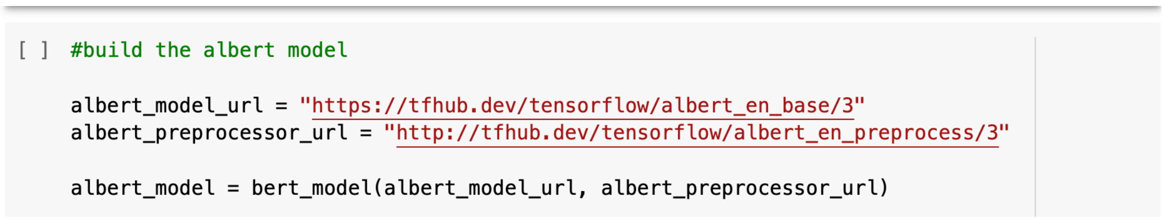
```
[ ] #compile parameters

    from tensorflow.keras import optimizers
    from tensorflow.keras.losses import BinaryCrossentropy
    from tensorflow.keras.metrics import BinaryAccuracy

    #fit and compile parameters

    epoch = 10
    batch_size = 250
    train_size = 25000
    val_steps = 10

    steps_per_epoch = 70
    step_train = int(steps_per_epoch * epoch)
    step_warmup = int(0.1* step_train)

    loss_func = BinaryCrossentropy(from_logits=True)

    metric = BinaryAccuracy()
```

For all our transformer-based models - ALBERT, BERT-Expert and ELECTRA, we use AdamW optimizer. This is a variation of the standard Adam optimizer that incorporates weight decay and not L2 regularisation. Learning rate warmup is also used to avoid overfitting early in the training period. Based on various literature, we select the initial learning rate to be 3e-5. Cross Entropy Loss is used to optimize the model. This particular choice is made because Cross entropy loss works particularly well when training for binary classification tasks. Binary Accuracy is used as the metric to measure how often the predicted labels match the true labels



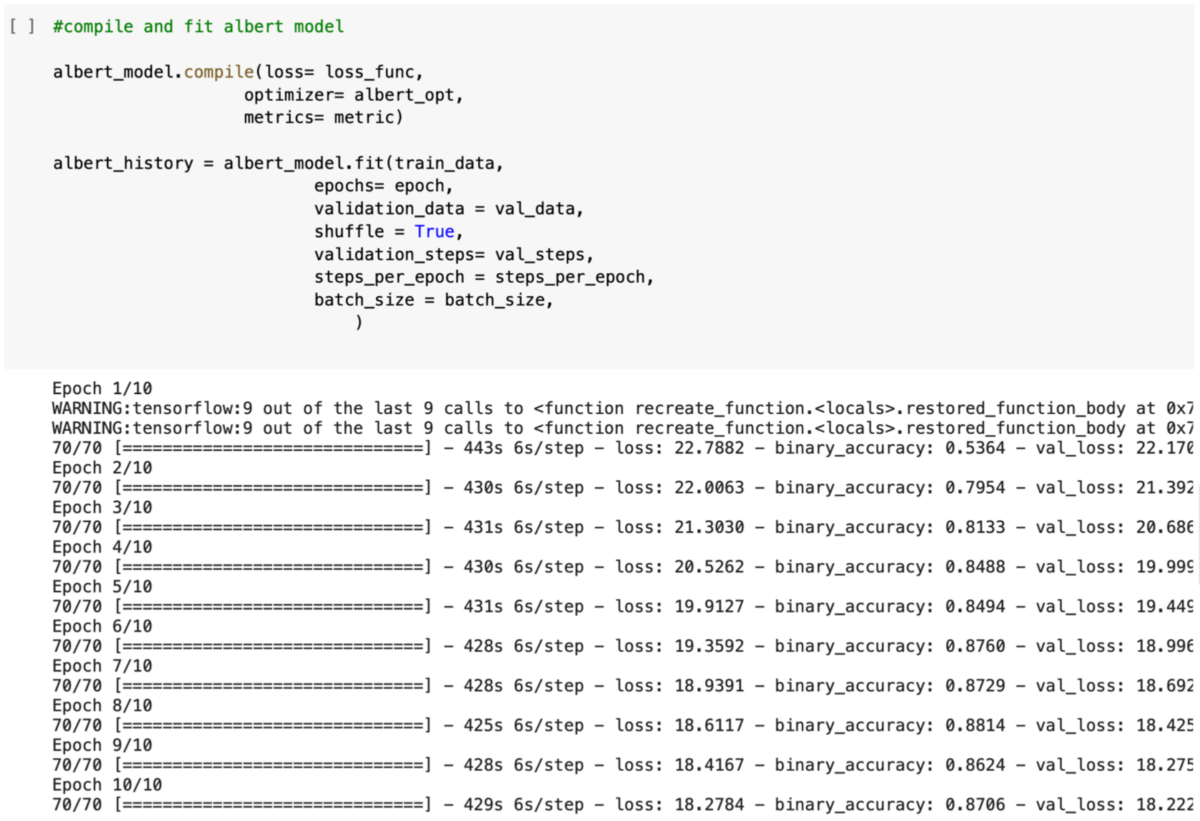
```
#build the adamWeight decay optimizer

from official.nlp import optimization
from official.nlp.optimization import WarmUp, AdamWeightDecay

albert_decay = tf.keras.optimizers.schedules.PolynomialDecay(
      initial_learning_rate=3e-5,
      decay_steps=step_train,
      end_learning_rate=0)

albert_warmup = WarmUp(initial_learning_rate=3e-5,
                decay_schedule_fn=albert_decay,
                warmup_steps=step_warmup)

albert_opt = AdamWeightDecay( learning_rate= albert_warmup,
        weight_decay_rate=0.01,
        exclude_from_weight_decay=['LayerNorm', 'layer_norm', 'bias'],
        epsilon=1e-6)
```

The model building process for all three models is the same. We first define the specific model using the “bert_model” function. For example, to build the ALBERT model we first save the path to the ALBERT model and the preprocessor of the ALERT model in different variables



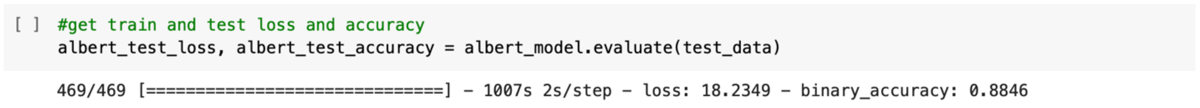
```
[ ]  #build the albert model

     albert_model_url = "https://tfhub.dev/tensorflow/albert_en_base/3"
     albert_preprocessor_url = "http://tfhub.dev/tensorflow/albert_en_preprocess/3"

     albert_model = bert_model(albert_model_url, albert_preprocessor_url)
```

We then compile the model with AdamW optimizer, cross entropy loss and Binary Accuracy. The model is fit to the training data to begin training



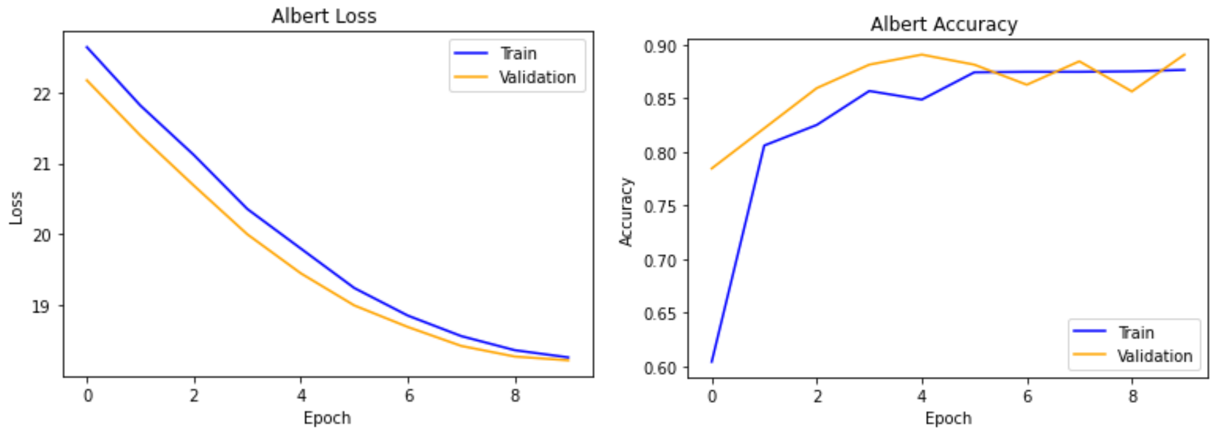
```
[ ]  #compile and fit albert model

     albert_model.compile(loss= loss_func,
                          optimizer= albert_opt,
                          metrics= metric)

     albert_history = albert_model.fit(train_data,
                                       epochs= epoch,
                                       validation_data = val_data,
                                       shuffle = True,
                                       validation_steps= val_steps,
                                       steps_per_epoch = steps_per_epoch,
                                       batch_size = batch_size,
                                           )

     Epoch 1/10
     WARNING:tensorflow:9 out of the last 9 calls to <function recreate_function.<locals>.restored_function_body at 0x7
     WARNING:tensorflow:9 out of the last 9 calls to <function recreate_function.<locals>.restored_function_body at 0x7
     70/70 [==============================] - 443s 6s/step - loss: 22.7882 - binary_accuracy: 0.5364 - val_loss: 22.170
     Epoch 2/10
     70/70 [==============================] - 430s 6s/step - loss: 22.0063 - binary_accuracy: 0.7954 - val_loss: 21.392
     Epoch 3/10
     70/70 [==============================] - 431s 6s/step - loss: 21.3030 - binary_accuracy: 0.8133 - val_loss: 20.686
     Epoch 4/10
     70/70 [==============================] - 430s 6s/step - loss: 20.5262 - binary_accuracy: 0.8488 - val_loss: 19.999
     Epoch 5/10
     70/70 [==============================] - 431s 6s/step - loss: 19.9127 - binary_accuracy: 0.8494 - val_loss: 19.449
     Epoch 6/10
     70/70 [==============================] - 428s 6s/step - loss: 19.3592 - binary_accuracy: 0.8760 - val_loss: 18.996
     Epoch 7/10
     70/70 [==============================] - 428s 6s/step - loss: 18.9391 - binary_accuracy: 0.8729 - val_loss: 18.692
     Epoch 8/10
     70/70 [==============================] - 425s 6s/step - loss: 18.6117 - binary_accuracy: 0.8814 - val_loss: 18.425
     Epoch 9/10
     70/70 [==============================] - 428s 6s/step - loss: 18.4167 - binary_accuracy: 0.8624 - val_loss: 18.275
     Epoch 10/10
     70/70 [==============================] - 429s 6s/step - loss: 18.2784 - binary_accuracy: 0.8706 - val_loss: 18.222
```

Once training is finished, we evaluate the model on testing data to get testing accuracy. As you can see the testing accuracy for the Albert Model is 88.46%

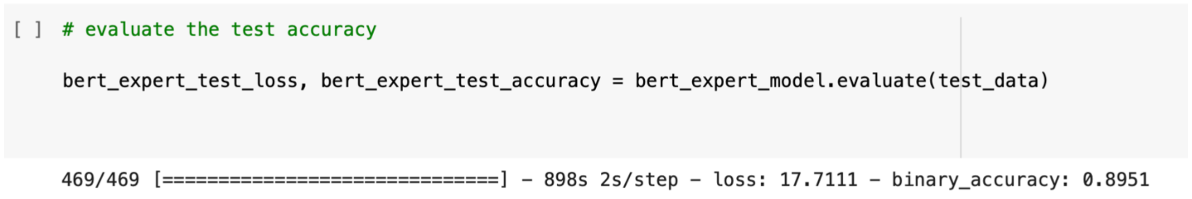


```
[ ] #get train and test loss and accuracy
    albert_test_loss, albert_test_accuracy = albert_model.evaluate(test_data)

    469/469 [==============================] - 1007s 2s/step - loss: 18.2349 - binary_accuracy: 0.8846
```

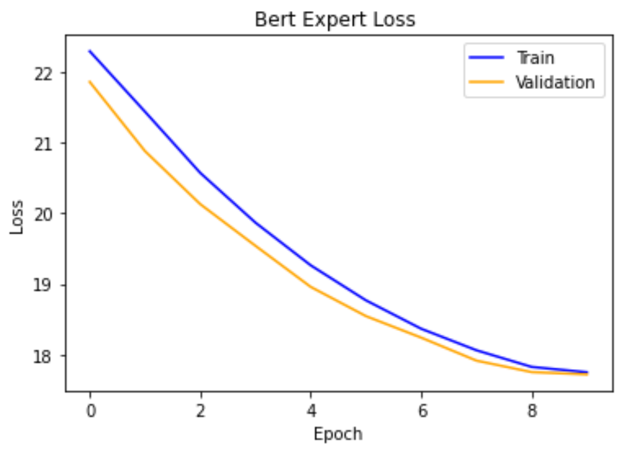
We also plot the loss and accuracy for each epoch to see how to model converges. In case of ALBERT, the validation accuracy stabilizes around the fifth epoch.

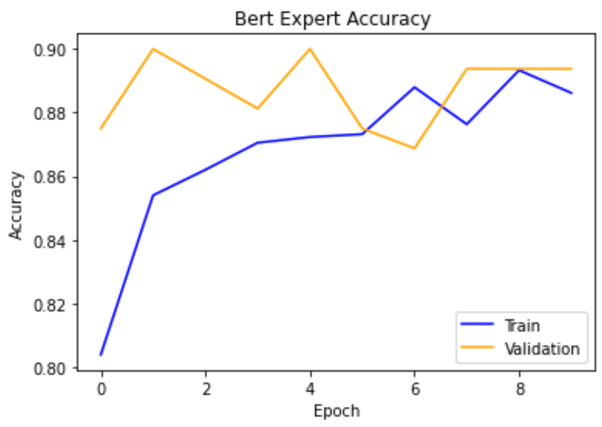


We follow the same process for training the BERT Expert model and achieve 89.5% testing accuracy. The validation accuracy converges around the 8th epoch.

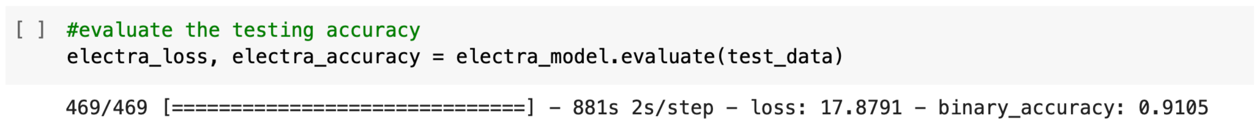


```
[ ] # evaluate the test accuracy

    bert_expert_test_loss, bert_expert_test_accuracy = bert_expert_model.evaluate(test_data)

    469/469 [==============================] - 898s 2s/step - loss: 17.7111 - binary_accuracy: 0.8951
```

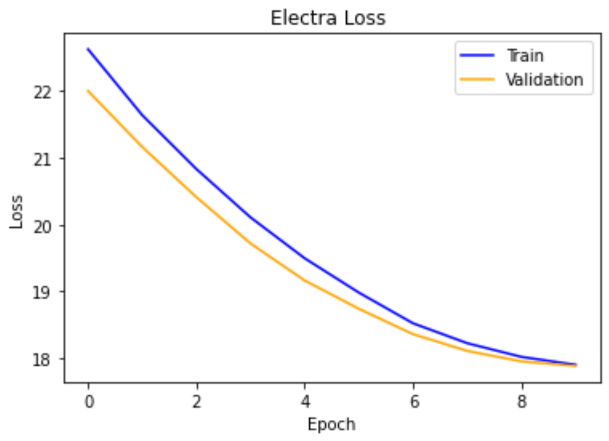


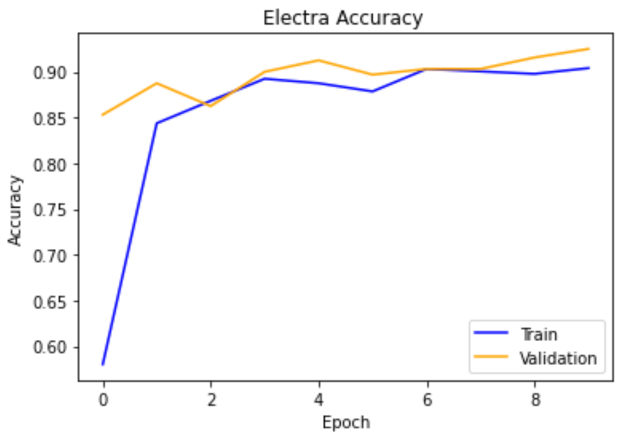


When evaluating ELECTRA using the same methodology, we achieve 91 % testing accuracy, and the validation accuracy converges relatively fast around the 3rd epoch.



```
[ ]  #evaluate the testing accuracy
     electra_loss, electra_accuracy = electra_model.evaluate(test_data)

     469/469 [==============================] - 881s 2s/step - loss: 17.8791 - binary_accuracy: 0.9105
```





V. Results

Our final results are in line with the literature. ELECTRA models perform the best with the highest test accuracy of 91.05%. It is followed by the BERT- Expert model with 89.5% accuracy. LSTM performs the worst with only 79.4% accuracy which puts it behind even the

Naive Bayes model in terms of accuracy performance. However, perhaps with some hyperparameter tuning the LSTM model will be able to outperform the Naive Bayes classifier.

The multi head attention and bidirectional text processing of transformer-based models certainly improve their accuracy well above the traditional sequential and bag-of-words based models. We notice approximately a 10% increase in test accuracy when we switch from traditional models to transformer-based models. The performance of all the transformer-based models remained fairly close to each other – between 88-92%.

The ALBERT model with almost half as many parameters as the other transformer-based models only suffered 2% in terms of accuracy as compared to ELECTRA. This is impressive considering that ALBERT requires a fraction of the computational cost compared to BERT-Expert and ELECTRA.

Our overall results are clear in displaying how transformer-based models have been a breakthrough technology for natural language processing.

References

Song, Y., Wang, J., Liang, Z., Liu, Z., & Jiang, T. (2020). Utilizing BERT intermediate layers for aspect based sentiment analysis and natural language inference. *arXiv preprint arXiv:2002.04815*.

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *arXiv preprint arXiv:1706.03762*.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., & Soricut, R. (2019). Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*.

Clark, K., Luong, M. T., Le, Q. V., & Manning, C. D. (2020). Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*.

Kibriya, A. M., Frank, E., Pfahringer, B., & Holmes, G. (2004, December). Multinomial naive bayes for text categorization revisited. In *Australasian Joint Conference on Artificial Intelligence* (pp. 488-499). Springer, Berlin, Heidelberg.

Minaee, S., Azimi, E., & Abdolrashidi, A. (2019). Deep-sentiment: Sentiment analysis using ensemble of cnn and bi-lstm models. *arXiv preprint arXiv:1904.04206*.