SENTIMENT ANALYSIS ON THE IMDB DATASET

Guefa Nguimnang Valdes
valdesguefa@gmail.com

Abstract

this report aims to explain the approaches that have been set up to do sentiment analysis on user reviews, these reviews being in the Large Movie Review Dataset (IMBD). For this purpose, we have realized two CNN models; LSTM model and the pre-trained google Bert-base-uncased (hugging face transformers) which is a large deep learning model trained on huge corpus as Wikipedia and tends to have understanding of how language works, in this work we are focusing on supervised learning. the BERT model was deployed using the gradio tool at the link https://38195.gradio.app

I. Introduction

Deep Learning has emerged as an effective megadata analysis tool that uses complex algorithms and artificial neural networks to train machines/computers to learn from experience, classify and recognize data just as a human brain does. In Deep Learning, a convolutional neural network or CNN is a type of artificial neural network widely used for image recognition and text classification like sentiment analysis. Deep Learning classifies texts according to their context. CNNs play a major role in various tasks such as image processing problems, computer vision tasks such as localization and segmentation, video analysis, obstacle recognition in autonomous cars, speech recognition and sentiment analysis in natural language processing. As CNNs play an important role in these fast-growing and emerging fields, they are very popular in Deep Learning.
Furthermore, a typical neural network will have an input layer, hidden layers and an output layer. CNNs are inspired by the architecture of the brain. Just as a neuron in the brain processes and passes information to the neurons in the next layer, artificial neurons or nodes in CNNs take inputs, process them and send the result as output. During training, the neural network adjusts its weights in order to provide the correct output based on the given input. Theoretically, at the end of training, the neural network should be able to infer the right output even for inputs that were not provided during training, which is known as generalization.

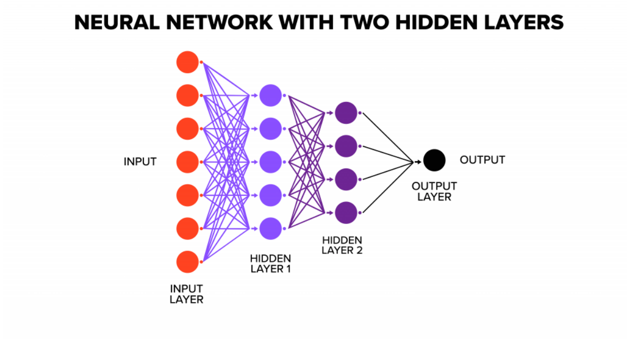


fig1 :deep learning and deep neural network

A transformer model is a neural network that learns context and thus meaning by tracking relationships in sequential data like the words in this sentence. Transformer models apply an evolving set of mathematical techniques, called attention or self-attention, to detect subtle ways even distant data elements in a series influence and depend on each other.
Most competitive neural sequence transduction models have an encoder-decoder structure. Here, the encoder maps an input sequence of symbol representations (x1, ..., xn) to a sequence of continuous representations z = (z1, ..., zn). Given z, the decoder then generates an output sequence (y1, ..., ym) of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next. The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure bellow, respectively

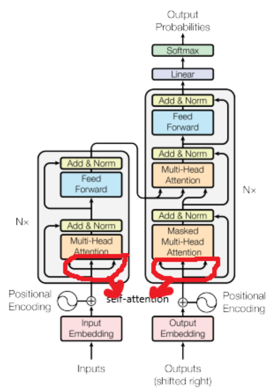


Fig2 : https://arxiv.org/pdf/1706.03762.pdf

Encoder [2]: The encoder is composed of a stack of N = 6 identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, positionwise fully connected feed-forward network. We employ a residual connection around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is LayerNorm (x + Sublayer(x)), where Sublayer(x) is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension d_{model}= 512.
Decoder [2]: The decoder is also composed of a stack of N = 6 identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.
Transformers use positional encoders to tag data elements coming in and out of the network. Attention units follow these tags, calculating a kind of algebraic map of how each element relates to the others. Attention queries are typically executed in parallel by calculating a matrix of equations in what’s called multi-headed attention. With these tools, computers can see the same patterns humans see.

II. Methodology

III.1 Pre-processing Data

a) Data Collection and annotation

the constitution of our dataset is made thanks to the comments contained in each .txt file provided in the assignment. these txt files are grouped in directories whose name indicates the class of comments it contains. The base dataset contains 50,000 reviews divided equally into 25k training sets and 25k test sets. The overall distribution of labels is balanced (25k pos and 25k neg), it also contains an additional 50k unlabelled comments for unsupervised learning.

b) Remove punctuation

Punctuation can provide a grammatical context for a sentence, which helps our understanding. But for
our vectorizer, which counts the number of words and not the context, it does not add value, so we remove all special characters. special characters. e.g.: What is your name? =>What is your name.
moreover, we remove the html strips and the line break characters which do not impact on the meaning of the sentence. after that we put all the text in capital letters.

c) Tokenization

Tokenizing separates text into units such as sentences or words. It gives structure to previously unstructured text. Here, tokens can be either words, characters, or subwords. Hence, tokenization can be broadly classified into 3 types – word, character, and subword (n-gram characters) tokenization. in our work we use Word Tokenization e.g.: Hello World → 'Hello, 'World '. after that we apply an encoding on each of these tokens.

d) Remove stopwords

Stopwords are common words that will likely appear in any text. They don't tell us much about our data so we remove them. eg:
I like reading, so i read →'like', 'reading', 'read'.

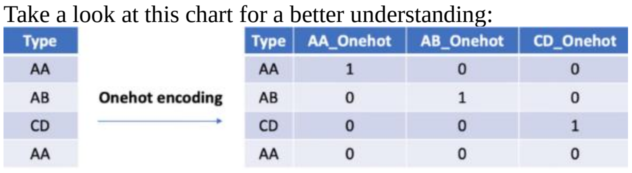
e) Lemmatisation

Lemmatization is the process of grouping together the different inflected forms of a word so they can be as a single item. Lemmatization is similar to stemming but it brings context to the words. So, it links words with similar meanings to one word.eg: analyzed, analyzing → analyze

III.2 Feature Generation

a) Vectorizing Data

Vectorizing is the process of encoding text as integers i.e. numeric form to create feature vectors so that machine learning algorithms can understand our data. For LSTM and CNN models we used One Hot Encoding on each token of our dataset. With one-hot, we convert each token value into a new categorical column and assign a binary value of 1 or 0 to those columns. Each integer value is represented as a binary vector. All the values are zero, and the index is marked with a 1.
Take a look at this chart for a better understanding:


Onehot encoding

Type
AA
AB
CD
AA

Type	AA_Onehot	AB_Onehot	CD_Onehot
AA	1	0	0
AB	0	1	0
CD	0	0	1
AA	0	0	0

Fig3: One-Hot encoding

we use one hot encoding because One hot encoding makes our training data more useful and expressive, and it can be rescaled easily. By using numeric values, we more easily determine a probability for our values. In particular, a hot coding is used for our output values, as it provides more nuanced predictions than single labels. it is also less computationally expensive than the Word Embedding on large data sets.
Transformers [2] used sinusoidal positional encoding. The formula is written below where **pos** is positional indices of words in the sentences, **d** is embedding vector dimension and **i** is the position of indices in that embedding vector. Using Sin and Cosine waves for even and odd indices removes the duplicate embedding values (cosine wave is zero more than one-time similar way sin wave).

$$PE_{(pos,2i)} = \sin\,(pos\backslash 10000^{2i/d})$$
$$PE_{(pos,2i+1)} = \cos\,(pos\backslash 10000^{2i/d})$$

In BERT, we do not have to give sinusoidal positional encoding, the model itself learns positional integration during the learning phase. BERT uses the concept of word tokenizer which is nothing else than dividing some words into subwords. For example, in the image below, the word "sleep" is tokenized into "sleep" and "##ing".

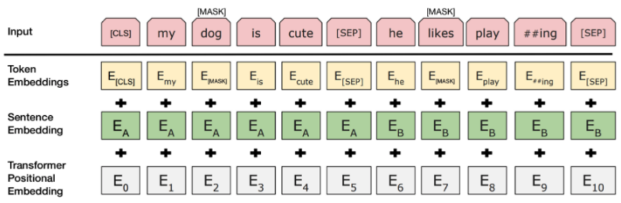


Fig3: BERT [*Devlin et al*., 2018]

III.3 Building model

a) BERT

In our work, we use the basic version of Bert, namely Bert Base. BERT is basically a trained Transformer Encoder stack. Both BERT model sizes have a large number of encoder layers (which the paper calls Transformer Blocks) – twelve for the Base version, and twenty-four for the Large version. These also have larger feedforward-networks (768 and 1024 hidden units respectively), and more attention heads (12 and 16 respectively) than the default configuration in the reference implementation of the Transformer in the initial paper (6 encoder layers, 512 hidden units, and 8 attention heads).

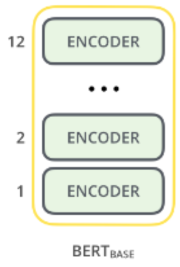
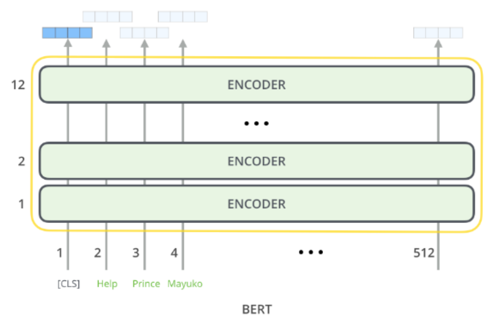


Fig4: Bert Base Encoder

The first input token is supplied with a special [CLS] token. BERT takes a sequence of words as input which keep flowing up the stack. Each layer applies self-attention, and passes its results through a feed-forward network, and then hands it off to the next encoder.

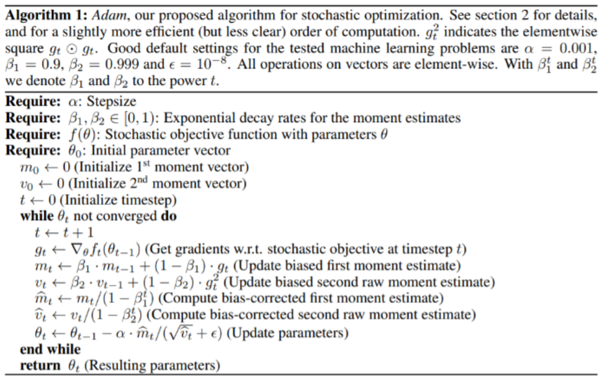
SENTIMENT ANALYSIS ON THE IMDB DATASET

Guefa Nguimnang Valdes
valdesguefa@gmail.com



Each position outputs a vector of size hidden_size (768 in BERT Base). For the sentence classification we've worked, we focus on the output of only the first position (that we passed the special [CLS] token to). this vector is used to perform our classification task.
during the different training, we use the Adam optimizer [1] with learning rate lr=1e-5, ε=1e-8.

$$\theta_{n+1} = \theta_n - \frac{\alpha}{\sqrt{\hat{v}_n + \epsilon}}\hat{m}_n$$



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Fig5: Adam Algorithm [3]

Adaptive Moment Estimation (Adam) [1] is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t Adam also keeps an exponentially decaying average of past gradients m_t similar to momentum. Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface [3]. m_t And v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.
As $m_t$and $v_t$are initialized as vectors of 0, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e β_1 and β_2 are close to 1). They counteract these biases by computing bias-corrected first and second moment estimates: $\widehat{m}_t$ and $\hat{v}_t$. They then use these to update the parameters, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\varepsilon + \sqrt{\hat{v}_t}}\widehat{m}_t$$

b) LSTM

Long Short-Term Memory Network or LSTM, is a variation of a recurrent neural network (RNN) that is quite effective in predicting the long sequences of data like sentences and stock prices over a period of time.
It differs from a normal feedforward network because there is a feedback loop in its architecture. It also includes a special unit known as a memory cell to withhold the past information for a longer time for making an effective prediction.
In fact, LSTM with its memory cells is an improved version of traditional RNNs which cannot predict using such a long sequence of data and run into the problem of vanishing gradient.

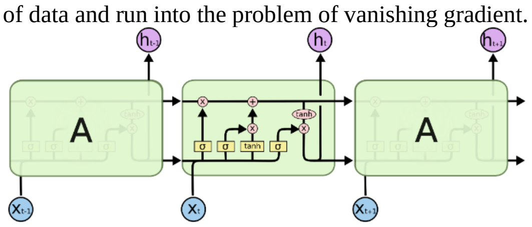
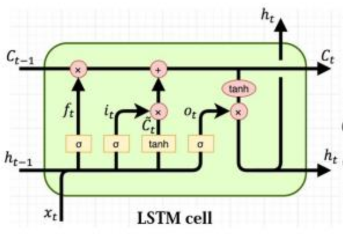


Fig6: **Long Short-Term Memory Network**

$$\sigma(x) = \frac{1}{1+e^{x}} \; and \tanh(x) = \frac{(e^{x}-e^{-x})}{(e^{x}+e^{-x})}$$

In this work, Long Short-Term Memory (LSTM) cells are used as the nodes of recurrent neural networks (see Figure 7). In an LSTM cell there are extra gates, namely the input, forget and output gate that are used in order to decide which signals are going to be forwarded to another node. W is the recurrent connection between the previous hidden layer and current hidden layer. U is the weight matrix that connects the inputs to the hidden layer. $\tilde{C}$ is a candidate hidden state that is computed based on the current input and the previous hidden state. C is the internal memory of the unit, which is a combination of the previous memory, multiplied by the forget gate, and the newly computed hidden state, multiplied by the input gate. The equations that describe the behaviour of all gates in the LSTM cell are described in Figure 7.



$$i_t = \sigma(x_t U^i + h_{t-1} W^i)$$
$$f_t = \sigma(x_t U^f + h_{t-1} W^f)$$
$$o_t = \sigma(x_t U^o + h_{t-1} W^o)$$
$$\tilde{C}_t = \tanh(x_t U^g + h_{t-1} W^g)$$
$$C_t = \sigma(f_t * C_{t-1} + i_t * \tilde{C}_t)$$
$$h_t = \tanh(C_t) * o_t$$

Fig7 : Structure of the LSTM cell and equations that describe the gates of an LSTM cell.[4]

then we put in output of LSTM a dense layer having a sigmoid activation. this layer provides a single output indicating the probability that the input element is positive (label is 1). the architecture of the model is as follows.

SENTIMENT ANALYSIS ON THE IMDB DATASET

Guefa Nguimnang Valdes
valdesguefa@gmail.com

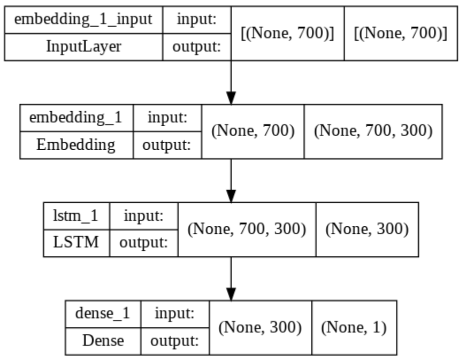
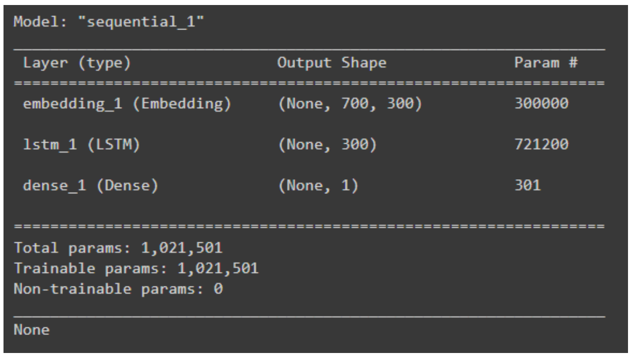


Fig8: LSTM + sigmoid activation



```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 700, 300)          300000

 lstm_1 (LSTM)               (None, 300)               721200

 dense_1 (Dense)             (None, 1)                 301

=================================================================
Total params: 1,021,501
Trainable params: 1,021,501
Non-trainable params: 0
_________________________________________________________________
None
```

Fig9: LSTM + sigmoid activation

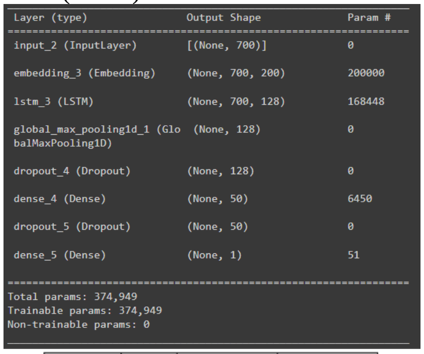
To prevent overfitting, we use Early stopping callback. Early stopping is a method that allows you to specify an arbitrarily large number of training epochs and stop training once the model performance stops improving on the validation dataset.
moreover, it aims to stop the training when the validation loss has started to plateau for the first time after 5 epochs. **Batch size** is a term used in machine learning and refers to the number of training examples utilized in one epoch for this model we use 32 like value of batch size and 10 epochs.

$$sigmoid\ function\ \sigma(x) = P(y = 1/x) = \frac{1}{1+e^{x}}$$

c) First CNN (CNN1)

The first layer is embedding. A word embedding is a learned representation for text where words that have the same meaning have a similar representation. Word embeddings are in fact a class of techniques where individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network, and hence the technique is often lumped into the field of deep learning.
Each word is represented by a real-valued vector, often tens or hundreds of dimensions. This is contrasted to the thousands or millions of dimensions required for sparse word representations, such as a one-hot encoding.
The distributed representation is learned based on the usage of words. This allows words that are used in similar ways to result in having similar representations, naturally capturing their meaning. This can be contrasted with the crisp but fragile representation in a bag of words model where, unless explicitly managed, different words have different representations, regardless of how they are used.

Keras offers an Embedding layer that can be used for neural networks on text data. It requires that the input data be integer encoded so that each word is represented by a unique integer. The Embedding layer is initialized with random weights and will learn an embedding for all of the words in the training dataset. The architecture of the model (CNN1) is as follows:



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 700)]             0

 embedding_3 (Embedding)     (None, 700, 200)          200000

 lstm_3 (LSTM)               (None, 700, 128)          168448

 global_max_pooling1d_1 (Glo  (None, 128)              0
 balMaxPooling1D)

 dropout_4 (Dropout)         (None, 128)               0

 dense_4 (Dense)             (None, 50)                6450

 dropout_5 (Dropout)         (None, 50)                0

 dense_5 (Dense)             (None, 1)                 51

=================================================================
Total params: 374,949
Trainable params: 374,949
Non-trainable params: 0
_________________________________________________________________
```

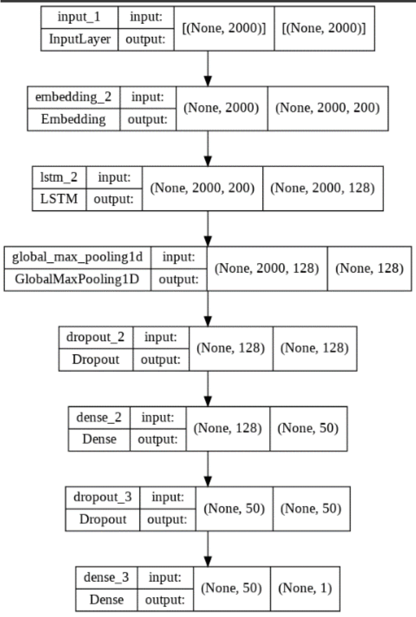


Fig 10 : CNN1

To the previous model, i added GlobalMaxPool1D to downsamples the input representation and Dense layers to generate more features, Dropout to avoid model overfitting.
How GlobalMaxPool1D works:

- Downsamples the input representation by taking the maximum value over the time dimension.
- Global max pooling = ordinary max pooling layer with pool size equals to the size of the input (minus filter size + 1, to be precise).
- Global pooling layers can be used in a variety of cases. Primarily, it can be used to reduce the dimensionality of the feature maps output by some convolutional layer, to replace Flattening and sometimes even Dense layers in your classifier (Christlein et al., 2019). What's more, it can also be used for e.g. word spotting (Sudholt & Fink, 2016). This is due to the property that it allows detecting noise, and thus "large outputs" (e.g. the value 9 in the example above). However, this is also one of

the downsides of Global Max Pooling, and like the regular one, we next cover Global Average Pooling.

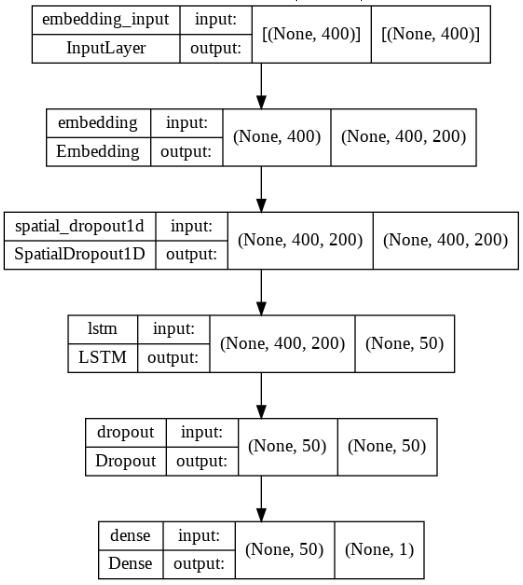
$$\mathrm{Max}\left(\begin{bmatrix}7 & 1 & 2\\ 9 & 2 & 3\\ 8 & 6 & 4\end{bmatrix}\right) \Rightarrow 9$$

To prevent overfitting, we use Early stopping callback. Early stopping is a method that allows you to specify an arbitrarily large number of training epochs and stop training once the model performance stops improving on the validation dataset.

moreover, it aims to stop the training when the validation loss has started to plateau for the first time after 5 epochs.

d) Second CNN (CNN2)

The architecture of the model (CNN2) is as follows:


embedding_input | InputLayer | input: [(None, 400)] | output: [(None, 400)]
embedding | Embedding | input: (None, 400) | output: (None, 400, 200)
spatial_dropout1d | SpatialDropout1D | input: (None, 400, 200) | output: (None, 400, 200)
lstm | LSTM | input: (None, 400, 200) | output: (None, 50)
dropout | Dropout | input: (None, 50) | output: (None, 50)
dense | Dense | input: (None, 50) | output: (None, 1)

a) Training and testing

Throughout this assignment we have used accuracy as a performance measure. Accuracy ranges between 0 and 1. These extreme cases correspond to completely missing the predictions or having always correct predictions. For instance, if our model is able to perfectly predict, the model will have no False Positives or False Negatives, making the numerator be equal to the denominator, bringing the Accuracy to 1. Conversely, if our system is always off, incorrectly predicting each time, the number of True Positives and True Negatives will be zero, making the equation be zero divided by something positive, leading to an Accuracy equal to 0. In real life, Accuracy technically ranges between 0.5 and 1, because if the Accuracy falls below 0.5, we can simply flip the predictions labels to obtain a better prediction. we choose the accuracy of performance measurement because the dataset is balanced (25k pos and 25k neg). for training the model we used 70% for training and 30% for testing

the BERT model was deployed using the gradio tool at the link https://38195.gradio.app

Model	Bert	CNN1	CNN2	LSTM
Accuracy(%)	90	76.87	82.95	76.78

references

[1] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In ICLR, 2015.
[2] https://arxiv.org/pdf/1706.03762.pdf

[3] Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., & Hochreiter, S. (2017). GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium. In Advances in Neural Information Processing Systems 30 (NIPS 2017).