


BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI, PILANI CAMPUS

A REPORT ON SENTIMENT ANALYSIS

CS F429: NATURAL LANGUAGE PROCESSING

 Sentiment_Analysis_2020B5A70951P.ipynb



DATASET: Amazon Alexa Reviews

<https://www.kaggle.com/datasets/sid321axn/amazon-alexa-reviews/data>

MEMBERS:

Swati Dubey: 2022H1030127P

Prakhar Pandey: 2020B5A70951P

1. Introduction

Sentiment Analysis focuses on understanding and classifying the sentiments of the text. It is used in various domains for multiple purposes. It has a role in extracting information, understanding public sentiment and improving user experiences in different applications. Over the years using advanced deep learning methods has greatly improved the precision and effectiveness of analyzing emotions. This study explores the use of **Google's state-of-the-art transformer-based NLP model, BERT (Bidirectional Encoder Representations from Transformers)**, for sentiment analysis tasks. Sentiment analysis and other NLP tasks have been transformed by BERT's capacity to extract bidirectional contextual information from the text and better understand the subtleties and context of language data.

The purpose of this study is to examine BERT's ability to understand emotions, in text by analyzing and categorizing expressions of sentiment(1-5) in a dataset. By using BERT's trained contextual embeddings this study will explore how adjusting and refining the BERT model architecture can improve the accuracy of sentiment analysis.

2. Data Preprocessing/Preparation

We have preprocessed the data using two different techniques, one for zero-shot, one for few-shot and fine-tuning.

2.1 Zero-Shot Processing

The `preprocess_text` function we used for raw data in the column **verified reviews** is designed to clean and preprocess text data so that the tokenizer and model

(<https://huggingface.co/nlptown/bert-base-multilingual-uncased-sentiment>) imported can easily compute sentiments.

Fig 2.1.1

Importing the tokenizer and model.

```
tokenizer = AutoTokenizer.from_pretrained('nlptown/bert-base-multilingual-uncased-sentiment')
model = AutoModelForSequenceClassification.from_pretrained('nlptown/bert-base-multilingual-uncased-sentiment')
```

Fig 2.1.2

Dataset Head.

```
1 df = pd.read_csv('amazon_alexa.tsv', sep='\t')
```

Python

```
1 df.head()
```

Python

	rating	date	variation	verified_reviews	feedback
0	5	31-Jul-18	Charcoal Fabric	Love my Echo!	1
1	5	31-Jul-18	Charcoal Fabric	Loved it!	1
2	4	31-Jul-18	Walnut Finish	Sometimes while playing a game, you can answer...	1
3	5	31-Jul-18	Charcoal Fabric	I have had a lot of fun with this thing. My 4 ...	1
4	5	31-Jul-18	Charcoal Fabric	Music	1

It takes a string of text as input and performs several preprocessing steps:

- 1. Punctuation removal:** The text is converted to lowercase and split into words. Then, all punctuation is removed from each word using the **translate** and **maketrans** functions.
- 2. Stopword removal:** Stopwords (commonly used words like **the**, **is**, **in**, **etc.**) are removed from the list of words. This is done using a list of English stopwords from the **nltk.corpus.stopwords** module.
- 3. Lemmatization:** Each word is lemmatized, or reduced to its base or dictionary form. This is done using the **WordNetLemmatizer** from the **nltk.stem** module. For example, **running** would be lemmatized to **run**.
- 4. Joining the text:** Finally, the preprocessed words are joined back together into a single string, with spaces between each word.

The function returns this cleaned and preprocessed text. This preprocessing can help improve the performance of text classification algorithms by reducing the dimensionality of the data and removing noise.

Fig 2.1.1

An example of what preprocessed text looks like.....

```
1 df['text'].head()
0      love echo
1      loved
2  sometimes playing game answer question correct...
3  lot fun thing 4 yr old learns dinosaur control...
4      music
Name: text, dtype: object
```

2.2 Few-Shot/Fine-Tuning Processing:

The preprocessor that we used for fine-tuning in our code is a preprocessor(<https://www.kaggle.com/models/tensorflow/bert/frameworks/tensorFlow2/variation/s/en-uncased-preprocess/versions/3?tfhub-redirect=true>) specifically designed for the BERT

model. It's often referred to as a BERT preprocessor. This preprocessor prepares the input data in a way that's suitable for the BERT model.

The BERT preprocessor performs the following steps:

1. Tokenization:

The input text is tokenized into words, subwords, or characters based on the BERT model's vocabulary. This is done using a **WordPiece tokenizer**.

2. Adding special tokens:

BERT requires special tokens to be added to the input data. Specifically, each sequence is prefixed with a **[CLS] (classification) token**, and sentence pairs are separated by a **[SEP] (separator) token**. One important point to note is that it treats each sequence (single-sentence or multiple) as a single input and adds **[CLS](ID 101)** to the beginning of the sequence and **[SEP](ID 102)** to the end.

3. Creating input sequences:

The tokenized inputs are then converted into sequences of fixed length (**128 tokens for smallBERT in this case**). If a text sequence is shorter than the fixed length, it's padded with **[PAD] tokens**.

4. Creating input embeddings:

BERT uses three types of input embeddings: **token embeddings**, **segment embeddings**, and **position embeddings**. The preprocessor creates these embeddings as follows:

Token Embeddings (*input_word_ids*): These are the IDs for each token in the tokenized text. Each ID corresponds to a token in the BERT model's vocabulary.

Segment Embeddings (*input_type_ids*): These are binary masks identifying the two sentence types in the model (0 and 1). In single sentence tasks, these are all 0.

Position Embeddings (*input_mask*): This is a mask of 1s and 0s where 1s represent real tokens and 0s represent padding ([PAD]) tokens.

This preprocessing makes the input data suitable for the BERT model, which requires fixed-length input sequences and specific types of embeddings.

For example for the text **“this is such an amazing movie!”**. The following are the keys obtained. [101] is [CLS] and [102] is [SEP].

The final step in the data preparation involved the creation of TensorFlow datasets from the features and labels in each set. These datasets were batched for efficient training and evaluation of the model.

A noteworthy optimization technique employed here is caching. By caching the datasets to memory, the time taken to load data during each epoch of training is significantly reduced. This is particularly beneficial when the data can fit into memory, as it allows the model to access the data faster, thereby speeding up the training process.

Prefetching was also used, which overlaps the preprocessing and model execution of a training step. While the model is executing training step `s`, the input pipeline is reading the data for step `s+1`. This improves the performance of the model training.

3. Ground Truth

The dataset taken here is a review. The customers has given their honest review with ratings. The dataset we have used has 3150 rows with 5 columns. Column **‘rating’** and **‘verified_reviews’** are the only useful ones to train the model and finding accuracy. The rating is categorized into 5 classes from 1 to 5 stars. The data is skewed with majority reviews with 5 and 4 values. Only **8%** of the reviews are **2** and **1** stars. This will affect the training of the model.

If we divide the ratings into classes, 4 and 5 being **‘positive’**, 1 and 2 being **‘negative’** and 3 finally being **‘neutral’**, we can see the distribution of various sentiment classes. The result is in the below diagram.

Fig 3.1

Class Distribution.

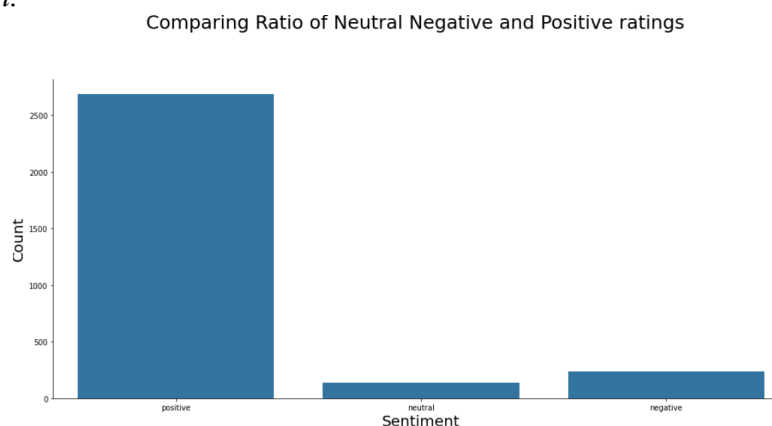
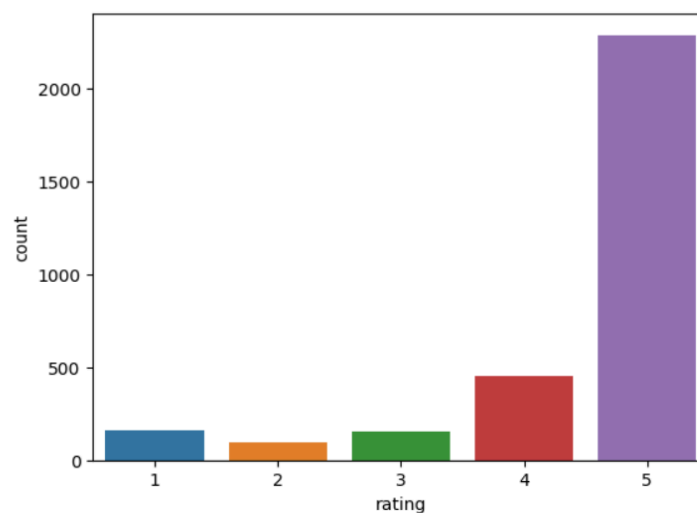


Fig 3.2

Rating Distribution.



Rating column statistics. Mean rating is 4.46.

Fig 3.3

[illegible]

4. Network Details

4.1 Zero-Shot

The model we have used for zero-shot testing is the ‘**bert base multilingual uncased sentiment**’ model. It is a variant of the BERT model. Here are the key details of its architecture:

Component	Description
Model Type	BERT
Version	Base
Transformer Blocks	12
Hidden Layers	768
Attention Heads	12
Pre-training	Multilingual product reviews
Case Sensitivity	Uncased (insensitive to case)

This model is specifically designed for sequence classification tasks, such as sentiment analysis. The 'base' version of BERT is used, which includes 12 transformer blocks. Each transformer block comprises self-attention mechanisms, which allow the model to weigh the importance of words in the input sequence.

The model has 768 hidden layers, enabling it to learn complex representations of the input data. It also uses 12 attention heads, allowing it to focus on different parts of the input for each head, providing a more nuanced understanding of the input data.

The model is pre-trained on multilingual product reviews, which equips it with a broad understanding of language and sentiment. The 'uncased' aspect means the model does not differentiate between upper and lower case characters, which is typically beneficial for sentiment analysis tasks.

4.2 Fine-Tuning

The model architecture used here is a custom classifier built on top of a pre-trained BERT model. The model is constructed using TensorFlow's Keras API, which allows for a clear and concise definition of the model layers.

4.2.1 Network Architecture

1. Input Layer:

The model begins with an input layer that accepts a scalar string. This layer is defined using `tf.keras.layers.Input` and is named `"text"`. The input dimension here is dependant on raw input text.

2. Pre-processing Layer:

Following the input layer, a pre-processing layer is defined using a pre-trained TensorFlow Hub model specified by `tfhub_handle_preprocess` which is the same preprocessor explained earlier. This layer transforms the raw text input into a format that can be fed into the BERT model. It tokenizes the text and converts the tokens into integer IDs, matching the vocabulary of the pre-trained BERT model. The output dimension of this layer is dependent on the sequence length after tokenization.

3. BERT Encoder:

The preprocessed text is then passed to the BERT encoder. This encoder is a pre-trained TensorFlow Hub model specified by `tfhub_handle_encoder`. The specific model used here is `'small_bert/bert_en_uncased_L-4_H-512_A-8'`, which is a smaller version of the standard BERT model with 4 transformer blocks, a hidden size of 512, and 8 attention heads. The encoder is set to be trainable, allowing the weights of the BERT model to be fine-tuned on the specific task at hand. The BERT model outputs a dictionary with three keys: `'pooled_output'`, `'sequence_output'`, and `'encoder_outputs'`. The `'pooled_output'` is a fixed-length representation of the input text with shape `'[batch_size, H]'`, where `'H'` is 512 for this model. The `'sequence_output'` has shape `'[batch_size, seq_length, H]'`, representing each input token in the context. The `'encoder_outputs'` are the intermediate activations of the `'L'` Transformer blocks, each with shape `'[batch_size, seq_length, 1024]'`.

4. Dropout Layer (Regularization):

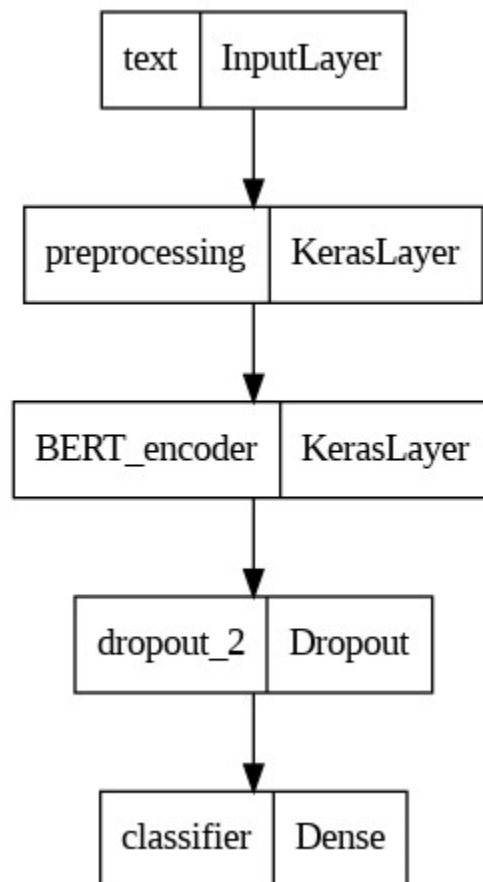
To prevent overfitting, a dropout layer is added after the BERT encoder. The dropout rate is specified as a parameter when building the model. During training, this layer randomly sets a fraction of its input (defined by **dropout_rate**) units to 0, which helps prevent overfitting. The input and output dimensions of this layer are both 512.

5. Output Layer:

Finally, a dense output layer with 5 units is added. This layer uses a softmax activation function, making the model's output a set of probabilities summing to 1. Each unit corresponds to one of the 5 classes that the model is trained to predict. The input dimension of this layer is 512, and the output dimension is 5.

Fig 4.2.1

Network Architecture.



4.2.2 Metrics and loss function

The loss function is set as `'Categorical Crossentropy'`. This is a popular choice for multi-class classification problems. The Categorical Cross-Entropy loss calculates the dissimilarity between the predicted probability distribution outputted by the model and the true distribution of the labels. The model's objective during training is to minimize this loss.

The performance of the model is evaluated using the `'Categorical Accuracy'` metric. This metric computes the mean accuracy rate across all predictions. In the context of multi-class classification, it checks if the maximal true value index matches the index of the maximal predicted value. It's a valuable metric for providing a clear understanding of the model's performance.

4.2.3 Optimizer (*AdamW*)

For the purpose of our model, the chosen optimizer is [`"AdamW"`](#), which is an extension of the popular "Adam" optimizer. AdamW is designed to improve the generalization of models by introducing weight decay for regularization, which is not present in the standard Adam optimizer. **This choice of optimizer aligns with the original training process of BERT, ensuring consistency in the fine-tuning phase.**

Here we have used a learning rate scheduler. This scheduler sets an initial learning rate (`'init_lr'`), which decreases linearly over the course of the training. This approach is commonly used in training large transformer models like BERT, as it helps in stabilizing the training process and achieving better performance.

A warm-up phase is also included at the start of the training. During this phase, which spans the first 10% of the training steps (`'num_warmup_steps'`), the learning rate increases linearly from zero to the initial learning rate. This warm-up phase helps in mitigating the impact of large gradient updates at the start of training, which can destabilize the model.

The total number of training steps (`'num_train_steps'`) is calculated as the product of the number of epochs and the number of steps per epoch. The number of steps per epoch is determined by the size of the training dataset (`'train_ds'`).

In summary, our code sets up an AdamW optimizer with a learning rate that starts with a warm-up phase and then decays linearly. This setup is consistent with the original BERT training

process and is designed to achieve effective fine-tuning of the model. **The AdamW optimizer used here is the novel mechanism that results in high training/testing accuracies!**

5. Results

5.1 Zero-Shot

For the zero-shot testing, we took the pre-trained-base-bert model and tested it on our review dataset. Two key metrics were calculated to evaluate the performance of the sentiment analysis model: **accuracy and mean squared error (MSE)**.

The accuracy is calculated using the `'accuracy_score'` function from the `'sklearn.metrics'` module. This function compares the model's predicted sentiments (`df['predicted_sentiment']`) with the actual ratings (`df['rating']`). The accuracy is the proportion of predictions that exactly match the actual ratings. In this case, **the model achieved an accuracy of approximately 63.49%**.

The mean squared error (MSE) is a measure of the average squared difference between the model's predictions and the actual values. Before calculating the MSE, the predicted sentiments and actual ratings are normalized to a range between 0 and 1. This is done by subtracting the minimum value and dividing by the range of the values. The MSE is then calculated as the mean of the squared differences between the normalized predicted sentiments and actual ratings. The reported **MSE for this model is approximately 8.36%**.

The model demonstrated a reasonable level of accuracy in predicting sentiment, correctly classifying around 63.49% of the samples. However, the MSE of 8.36% indicates that there is still room for improvement in the model's predictive performance.

The accuracy obtained here is quite similar to the accuracy reported by the hugging-face model library page(<https://huggingface.co/nlptown/bert-base-multilingual-uncased-sentiment>).

Fig 5.1.1

Accuracies obtained by the Hugging Face library community.

Accuracy

The fine-tuned model obtained the following accuracy on 5,000 held-out product reviews in each of the languages:

- Accuracy (exact) is the exact match for the number of stars.
- Accuracy (off-by-1) is the percentage of reviews where the number of stars the model predicts differs by a maximum of 1 from the number given by the human reviewer.

Language	Accuracy (exact)	Accuracy (off-by-1)
English	67%	95%
Dutch	57%	93%
German	61%	94%
French	59%	94%
Italian	59%	95%
Spanish	58%	95%

5.2 Fine-Tuning

5.2.1 Varying initial-learning rate

Dropout Rate = 0.15, Batch Size = 32

initial-learning rate	epochs	testing-loss	testing-accuracy
3e - 3	15	0.921	0.73
3e - 4	15	1.138	0.838
3e - 5	15	0.574	0.819
5e - 5	15	0.72	0.844
3e - 6	15	0.684	0.743

Fig 5.2.1.1

Initial Learning Rate = $3e-3$.

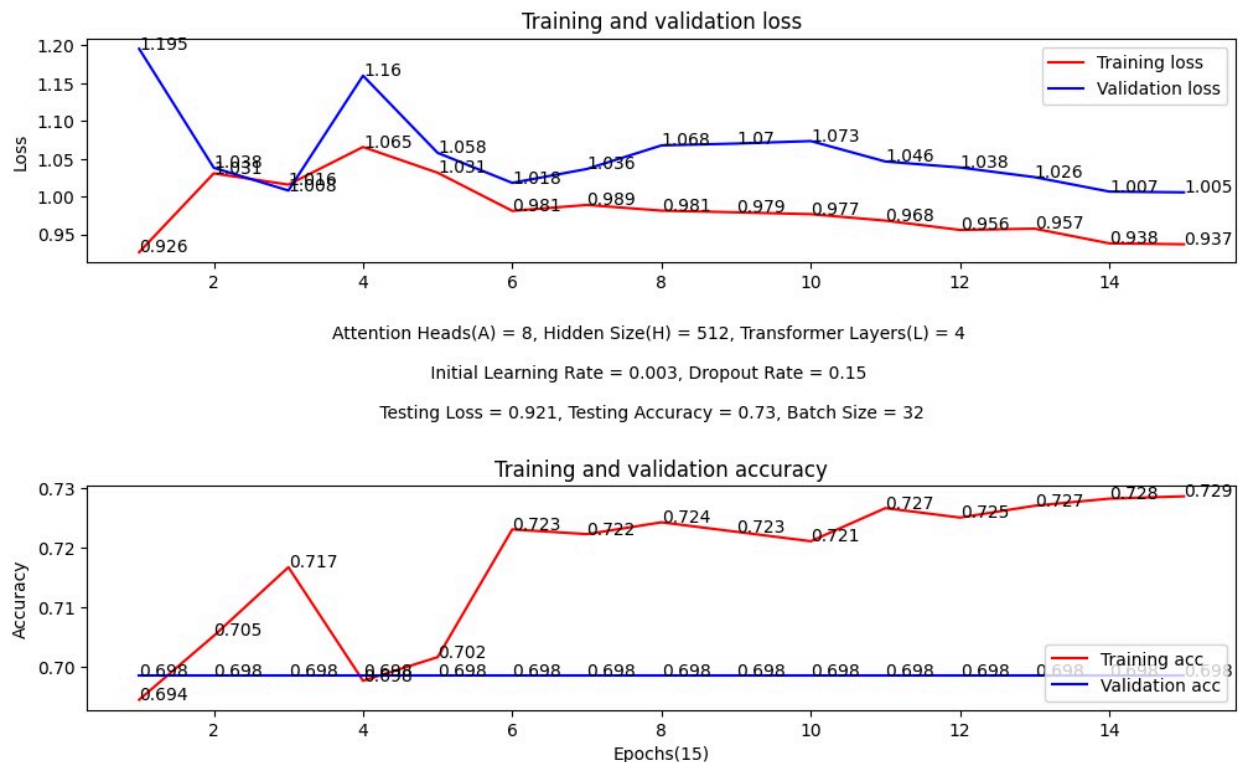


Fig 5.2.1.2

Initial Learning Rate = $3e-4$.

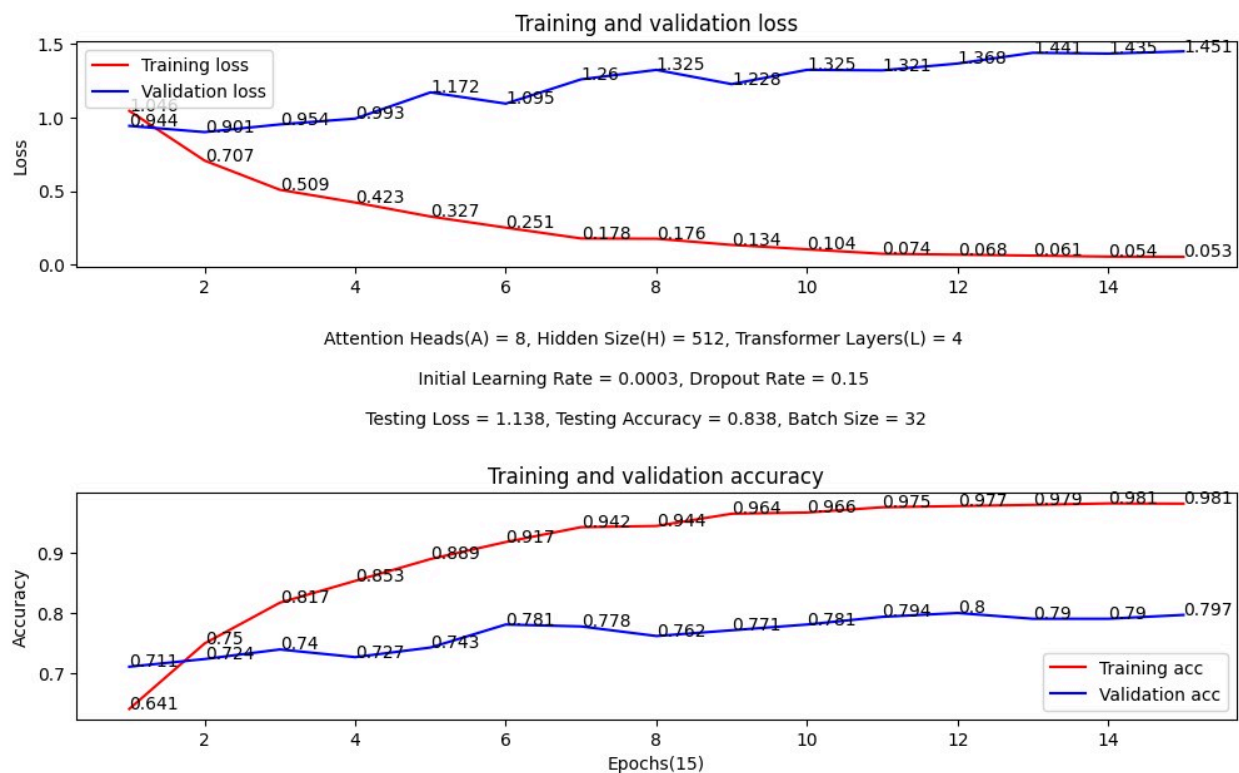


Fig 5.2.1.3

Initial Learning Rate = $3e-5$.

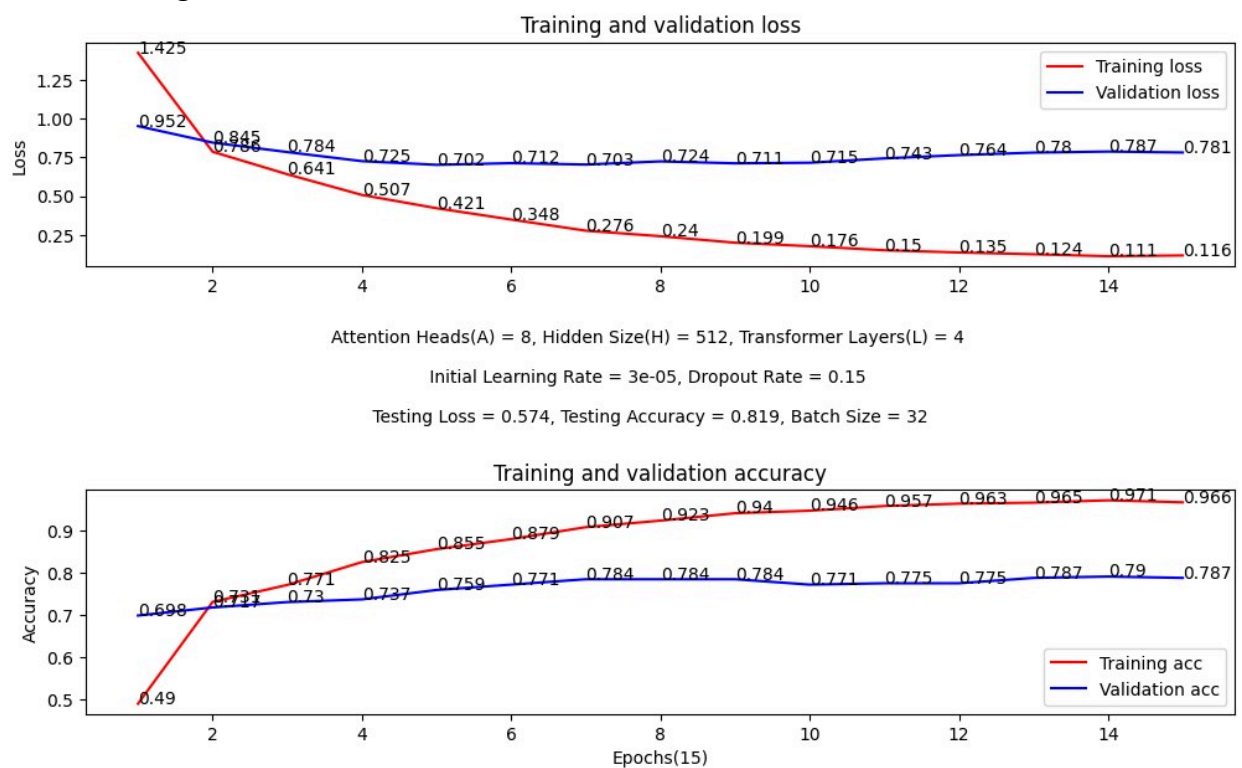
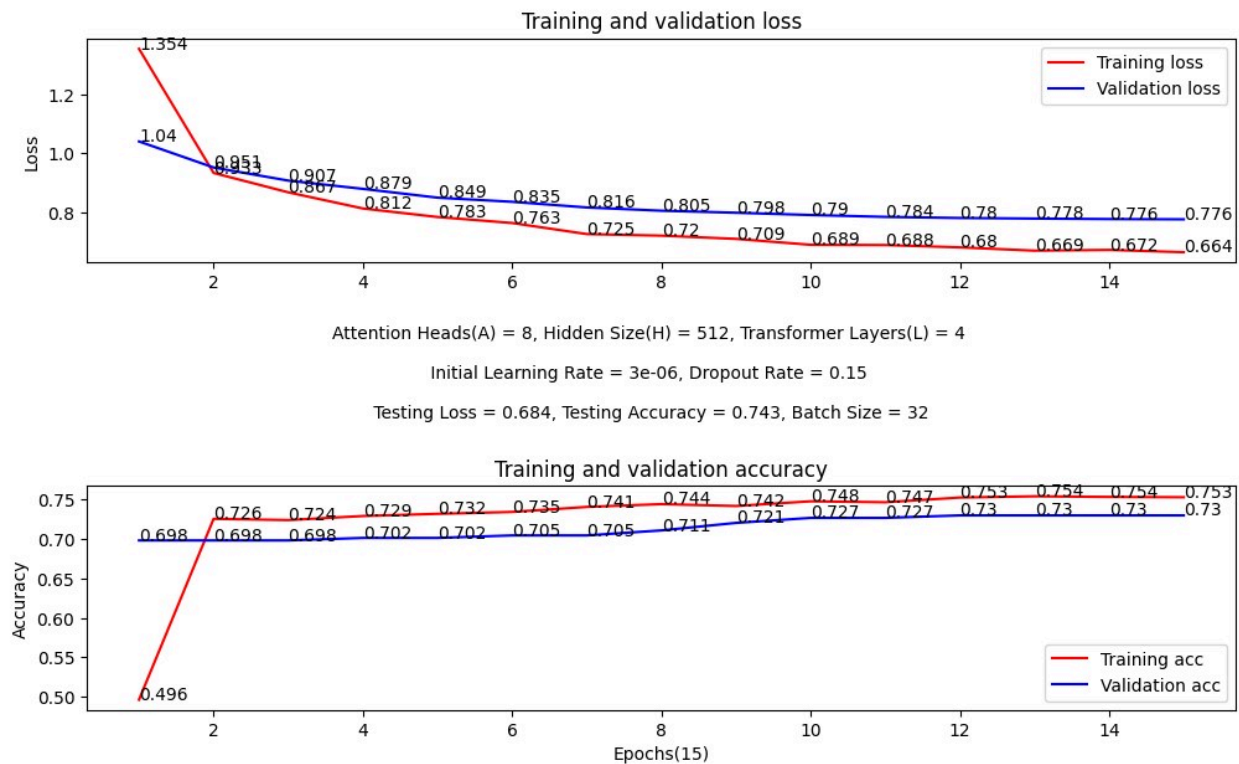


Fig 5.2.1.4

Initial Learning Rate = $3e-6$.



As is clearly visible the **lowest loss()** is achieved when the model is trained with the **initial learning rate** of $3e-05$ and **highest accuracy()** is achieved when the model is trained with the **initial learning rate** of $3e-04$ where **batch-size** is 32 and the **dropout-rate** is 0.15.

5.2.2 Single-training item v.s. Mini-batch

Dropout Rate = 0.15, Initial learning rate = $3e-05$, epochs = 12