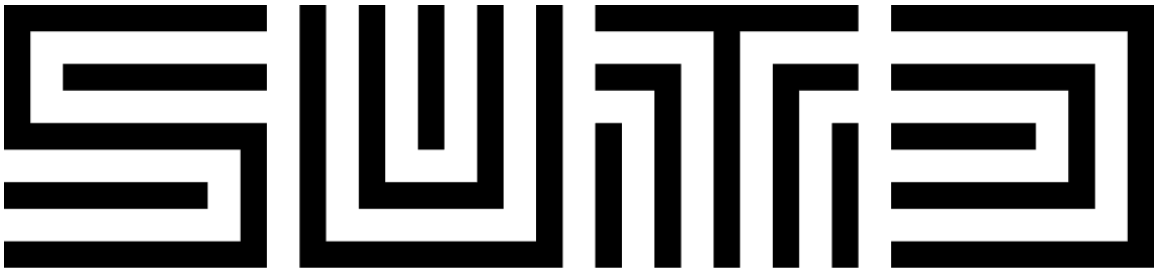

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.040 Natural Language Processing

Fall 2024

Design Project

Link to the GitHub repository:

https://github.com/jordanleewei/nlp-final-project

Advised by:

Professor Lu Wei

Chen Huang

Submitted by:

Anutham Mukunthan, 1006202

Jordan Lee Wei, 1005906

Shwetha Iyer, 1006308

Model Description

Architecture Overview

Our team wanted to pursue a transformer-based model as we had learned about them through Homework 3 and the associated lectures. This motivated us to build an encoder and decoder and combine them to form a transformer model. However during the experimentation phase, the team felt there was no need for a decoder so large with masked attention, as we were predicting only one class and not multiple words (as was the original use case of transformers). Hence, we decided to make it less complicated and pursue an encoder-only approach instead of an encoder-decoder approach.

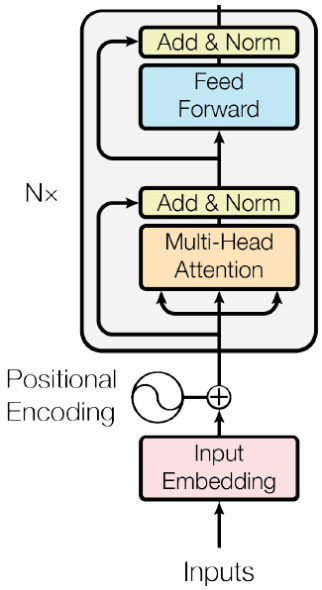

Figure 1 - Encoder architecture from "Attention is all you need (A Vaswani et al., 2017)"

Using the learnings from Homework 3, we built the attention mechanism, and added normalization and fully connected layers as described in Figure 1.

Embedding Layer

Pre-trained embeddings (GloVe) were used to initialize the token embeddings to get a fixed-dimensional vector representation, as was done for both the CNN and BiRNN models.

Positional Encoding

Typically positional encodings are used to provide the position of the word embeddings in the sentence. This is very useful for the attention mechanism as it guides where the model needs to focus on to extract the right features.

Encoder Blocks

Each encoder block consists of multi-head self-attention and position-wise feed-forward networks:

- Multi-Head Attention helps to capture different aspects of the sequence.
- Feed-Forward Network (FFN): A two-layer linear network with a ReLU activation transforms the attention outputs, providing a non-linear transformation.
- Normalization helps stabilize training and improve the model's representational capabilities.

In this project, we used one encoder layer with appropriate hyperparameters. If needed, the number of layers can be increased.

Aggregation + Classification

After processing the sequence, we aggregate token representations (e.g., by taking the mean over the sequence length). The resulting vector is then passed through a fully connected layer to produce a binary sentiment prediction (positive or negative).

Training Settings

Dataset

- **Training/Test Data Split:** Split into 25000 data points for training and 25000 for testing.
- **Preprocessing**
    - Tokenization at the word level using GloVe
    - Filtering out low-frequency words (min frequency=5).
    - Padding/truncating reviews to a fixed length (e.g., 500 tokens).
    - Constructing vocabulary and mapping words to indices.

Hyperparameters

- Embedding Dimension: 100
- Number of Heads: 5
- Hidden Size (Model Dimension): 100
- FeedForward Network Hidden Dimensions: 256
- Number of Encoder Layers: 1
- Dropout: 0.5
- Optimizer: Adam
- Learning Rate: 0.001
- Batch Size: 64
- Number of Epochs: 5
- Weight_decay: 1e-4

Hardware used was Nvidia T4 GPU on Google Colaboratory.

Alternative Experiments

This section details the other experiments conducted for the design challenge:

- Experiment 1: We utilized an `EncoderBlockModel` configured with a vocabulary size equal to the length of the vocabulary, an embedding size of 100, 100 hidden units, 5 attention heads, a feed-forward network with 64 hidden units, 2 layers, a dropout rate of 0.5, and 2 output classes. This yielded the results:

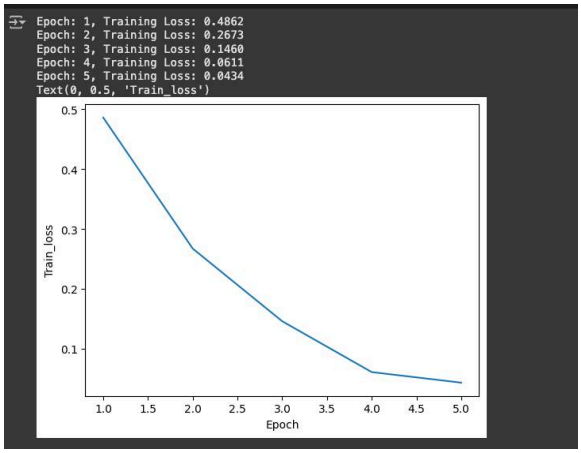

Figure 2 - Training the model for Experiment 1

```
positive
negative
(0.7704199249649686, 0.8855866154006027, 0.68176, 0.79684)
```

Figure 3 - Results from Experiment 1 when tested on sample sentences and test_data

```
[34] cal_metrics2(net, test_iter_final)

(0.8794996376130781, 0.948471514590088, 0.8198788849406936, 0.887775)
```

Figure 4 - Results from Experiment 1 when tested on test_data_movie.csv

- Experiment 2: In this experiment, we implemented an `EncoderBlockModel` with the following configuration: a vocabulary size matching the length of the vocabulary, an embedding size of 100, 100 hidden units, 5 attention heads, a feed-forward network with 256 hidden units, 2 layers, a dropout rate of 0.5, and 2 output classes.

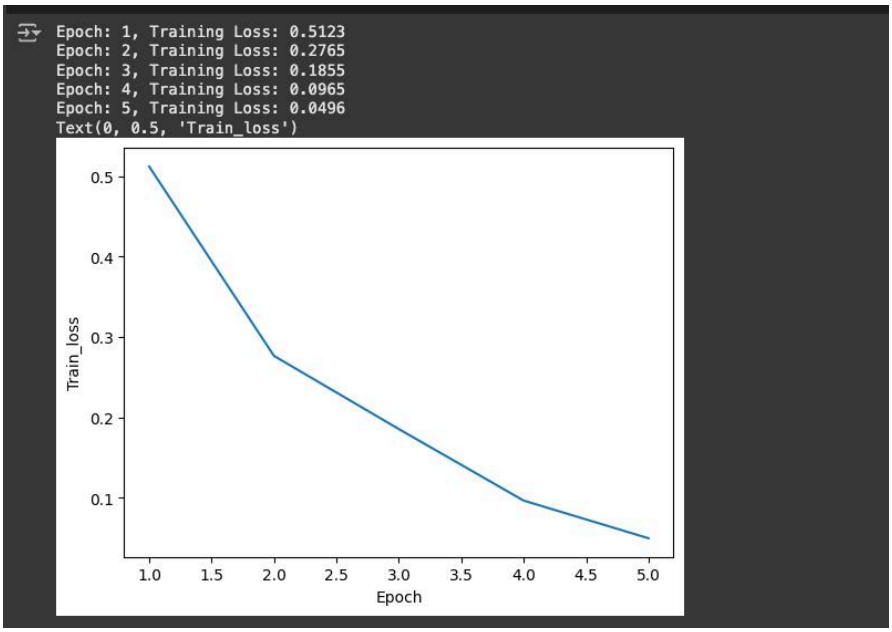

Figure 5 - Training the model for Experiment 2

```
positive
negative
(0.8235579645659736, 0.8329242349860907, 0.8144, 0.82552)
```

Figure 6 - Results from Experiment 2 when tested on sample sentences and test_data

```
cal_metrics2(net, test_iter_final)
```

```
(0.9072881014697, 0.912477853707922, 0.9021570491967369, 0.9079)
```

Figure 7 - Results from Experiment 2 when tested on test_data_movie.csv

- Experiment 3: For this experiment, we used an `EncoderBlockModel` configured with the following parameters: a vocabulary size matching the length of the vocabulary, an embedding size of 100, 100 hidden units, 5 attention heads, a feed-forward network with 128 hidden units, 1 layer, a dropout rate of 0.5, and 2 output classes.

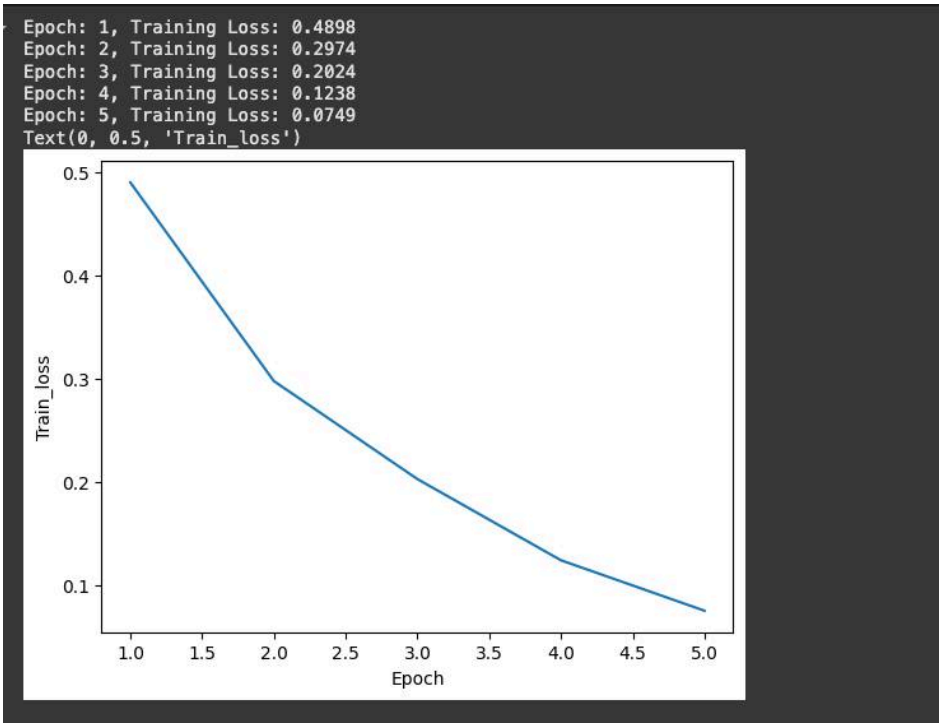

Figure 8 - Training the model for Experiment 3

```
positive
negative
(0.8585808986005402, 0.8787904171553024, 0.83928, 0.86176)
```

Figure 9 - Results from Experiment 3 when tested on sample sentences and test_data

```
44] cal_metrics2(net, test_iter_final)

(0.9258930158165986, 0.9395610057708161, 0.91261698613683, 0.927025)
```

Figure 10 - Results from Experiment 3 when tested on test_data_movie.csv

- Experiment 4: In this experiment, OpenAI's *text-embedding-3-small* model was used to generate embeddings for text data, which were then fed into an LSTM network for binary classification. The LSTM model had 128 hidden units, 3 layers, a dropout rate of 30%, and an output layer with 2 units for binary classification. The model was trained for 5 epochs, but during the testing phase, processing the embeddings for the test data became too time-consuming, due to the large size of the dataset. Even batch processing the dataset was proving to be impractical. This caused the test phase to be unfeasible, even though the model was successfully trained on the training data.

```
Epoch 1, Loss: 0.1858345013888329
Epoch 2, Loss: 0.1466468918099142
Epoch 3, Loss: 0.139286418756246 17
Epoch 4, Loss: 0.1309277027919221 5
Epoch 5, Loss: 0.1244708628851034 4
```

Figure 11 - Training the model for Experiment 4

Performance and Evaluation

We decided to choose a model with 1 encoder layer and 256 feed forward network units as it gave the best result from our experiments. After training, we evaluated our model on the test set (*test_data_movie.csv*).

Final Results:

Without Positional Encodings	With Positional Encodings
• F1 Score: 0.927	F1 Score - 0.918
• Precision: 0.945	Precision - 0.930
• Recall: 0.909	Recall - 0.903
• Accuracy: 0.928	Accuracy - 0.919

Issues and Challenges Faced

1. Adding more encoder layers caused the model to converge quickly but caused overfitting. When testing on the datasets, there was a decrease in the evaluation metric scores.
2. Adding positional encodings seemed to reduce the model's predictive capacity as can be seen in the results. This can be because sentiment analysis is more based on the combination of the words used than where they are placed. As such, passing positional encodings seems to hamper its predictions marginally by reducing the model's ability to capture some semantic relations.
3. Additionally, the cost of computing also proved to be an issue as we lacked a local GPU, forcing us to rely on Google Colaboratory which often timed out during training and evaluation.

Code Execution Instructions

Kindly refer to the README file (we used Google Colaboratory).

Link to the GitHub repository:
https://github.com/jordanleewei/nlp-final-project/blob/main/README.md

Alternatively, the execution instructions can be found below:

- Model trained in `Python 3.10.12` - Ensure that all these libraries are installed. If not please do so and then run the file - `math torch d2l pandas csv os matplotlib.`

For Google Colaboratory:

- In the Google Colaboratory environment, run `%pip install d2l.`
- Set runtime to t4 GPU and restart the runtime.

- Put *new_approach.py*, *test_data_movie.csv, and encoder_block_model.pt* (pretrained weights) in the "Files" section in the Google Colaboratory environment.
- The pretrained model weights should be present in the zip file of the repository.
- If the weights are present, run *python new_approach.py* The user will be prompted to input "train" or "infer". Enter "infer" and continue to observe model results.
- If the weights are not present, or if you wish to train again, run "*python new_approach.py*" The user will be prompted to input "train" or "infer". Enter "train" and continue. The model will train and run inference on the IMDB test set as well as the provided test set and return the evaluation metrics. The new weights will be saved and can be used for future inference.

For local environments,

- Download the zip file.
- Install all relevant modules using `pip install math torch d2l pandas csv os matplotlib`, in `Python 3.10.12.`
- The pretrained model weights should be present in the zip file.
- Follow the below steps with a terminal open in the folder containing the files:
    - If the weights are present, run python new_approach.py The user will be prompted to input "train" or "infer". Enter "infer" and continue to observe model results.
    - If the weights are not present, or if you wish to train again, run "python new_approach.py" The user will be prompted to input "train" or "infer". Enter "train" and continue. The model will train and run inference on the IMDB test set as well as the provided test set and return the evaluation metrics. The new weights will be saved and can be used for future inference.

Breakdown of Contributions

The below table details the contributions of each team member:

Student	ID	Contribution
Anutham Mukunthan	1006202	- Part 2 (RNN) - Implemented Encoder Block Model (final presented approach). - Ran experiments. - Final report writing.
Jordan Lee Wei	1005906	- Part 1 (Pre-processing) - Implemented evaluation functions. - Extracted prediction output CSV files. - Final report writing.
Shwetha Iyer	1006308	- Part 3 (CNN) - Ran experiments. - Exploring other embeddings (BERT and OpenAI). - Final report writing.

Table 1 - Contributions of individual team members