1 True/False Questions (30 pts)

Problem 1.1 – True

Explanation: For self-attention computation, attention scores and attentions weights are calculated based on computing three core variables – key, value and query.

Problem 1.2 – True

Explanation: In the self-attention layer, the attention score is computed by means of matrix multiplication between the query matrix and the transpose of key matrix. The inner product also belongs to cosine similarity.

Problem 1.3 – False

Explanation: We do not need to further apply a normalization on the obtained attention matrix.

Problem 1.4 – True

Explanation: According to the lecture slides, the pre-trained Transformer decoder could learn autoregressively.

Problem 1.5 – True

Explanation: According to lecture slides, BERT has an architecture of Transformer encoder for autoencoding, while GPT has an architecture of Transformer decoder for autoregressive learning.

Problem 1.6 – False

Explanation: BERT's pre-training objectives include (a) masked token prediction and (b) next sentence prediction, instead of sentence order prediction.

Problem 1.7 – False

Explanation: GPT-2 achieves state-of-the-art scores on different domain-specific language modeling tasks. Even though GPT-2 is not trained on any of the data specific to these tasks, it achieves comparable results with the supervised baseline, which is known as the "zero-shot" setting. Meanwhile, BERT is also a zero-shot learner.

Problem 1.8 – False

Explanation: Gradient clipping can be used to alleviate gradient exploding problem, instead of gradient vanishing problem.

Problem 1.9 – False

Explanation: Word embeddings can also have negative values or zeros, because if we want to use cosine similarity to create word embeddings, the output of a cosine function could range from -1 to +1.

Problem 1.10 – False

Explanation: The memory cell of an LSTM is indeed computed by a weighted average of previous memory state and current memory state, where the forget gate controls the previous memory state while the input gate controls the current memory state. However, both the forget gate and the input gate have their own neural network with their respective trainable weights and biases. Therefore, the sum of weights (forget gate value and input gate value) may not strictly be 1. The memory cell of GRU (Gated Recurrent Unit), however, does have the property that the sum of weights is 1.

Reports for Lab 1 and Lab 2 are on the following pages.

2 Lab 1: Implement and train an LSTM for sentiment analysis (35 pts)

(a) (5 pts) Implement your own data loader function. First, read the data from the dataset file on the local disk. Then split the dataset into three sets: train, validation, and test by 7 : 1 : 2 ratio. Finally return `x_train`, `x_valid`, `x_test`, `y_train`, `y_valid` and `y_test`, where `x` represents reviews and `y` represent labels.

(b) (5 pts) Implement the `build_vocab` function to build a vocabulary based on the training corpus. You should first compute the frequency of all the words in the training corpus. Remove the words that are in the `STOP_WORDS`. Then filter the words by their frequency ($\geq$ `min_freq`) and finally generate a corpus variable that contains a list of words.

(c) (5 pts) Implement the `tokenization` function. For each word, find its index in the vocabulary. Return a list of integers that represents the indices of words in the example.

(d) (5 pts) Implement the `__getitem__` function in the `IMDB` class. Given an index i, you should return the i-th review and label. The review is originally a string. Please tokenize it into a sequence of token indices. Use the `max_length` parameter to truncate the sequence so that it contains at most `max_length` tokens. Convert the label string ('positive' / 'negative') to a binary index, such as 'positive' is 1 and 'negative' is 0. Return a dictionary containing three keys: '`ids`', '`length`', '`label`' which represent the list of token ids, the length of the sequence, the binary label.

Note for Questions (a), (b), (c) and (d):

In Lab 1, I wrote my own function named "split_sentence_into_words" to split each review sentence (string) into a list of words (a list of strings), simply using the format of "string.split()". This means that I decided only to split sentence based on if there is a blank space between two words. After splitting, I convert all words into lower cases. As for punctuations, I decided not to deal with punctuation removal, and when I built my words vocabulary, I just kept all potential punctuations within each of my words.

(e) (10pts) Implement the LSTM model for sentiment analysis.

(a) (5pts) Write the initialization function. Your task is to create the model by stacking several necessary layers including an embedding layer, an lstm cell, a linear layer, and a dropout layer. You can call functions from Pytorch's nn library. For example, nn.Embedding, nn.LSTM, nn.Linear.

(b) (5pts) Write the forward function. Decide where to apply dropout. The sequences in the batch have different lengths. Write/call a function to pad the sequences into the same length. Apply a fully-connected (fc) layer to the output of the LSTM layer. Return the output features which is of size [batch size, output dim].

Note for Questions (e-a) and (e-b):

As for the dropout layer, I decided to apply dropout between the LSTM layer and the final fully-connected layer, which means my dropout layer takes the output of the LSTM layer as the input, and the output of my dropout layer will be sent to the final fully-connected layer as input. Although PyTorch LSTM module offers the option to incorporate dropout, I decided not to set any dropout within the LSTM layer, because I am a bit concerned that some important memory cells might be dropped out, and therefore affecting the training, validation and testing accuracies.

To pad the sequences into the same length, I called two functions from the PyTorch package, which are "pack_padded_sequence" and "pad_packed_sequence" respectively. The official website I referenced is shown here: https://pytorch.org/docs/stable/generated/torch.nn.utils.rnn.pad_packed_sequence.html

(f) (5pts) Train the model for 5 epochs. Copy the plotted figures of training/validation loss/accuracy to your self-contained pdf report. What is your testing accuracy? (The provided code contains the plot function and computation of test accuracy. You just need to report the value of testing accuracy.)

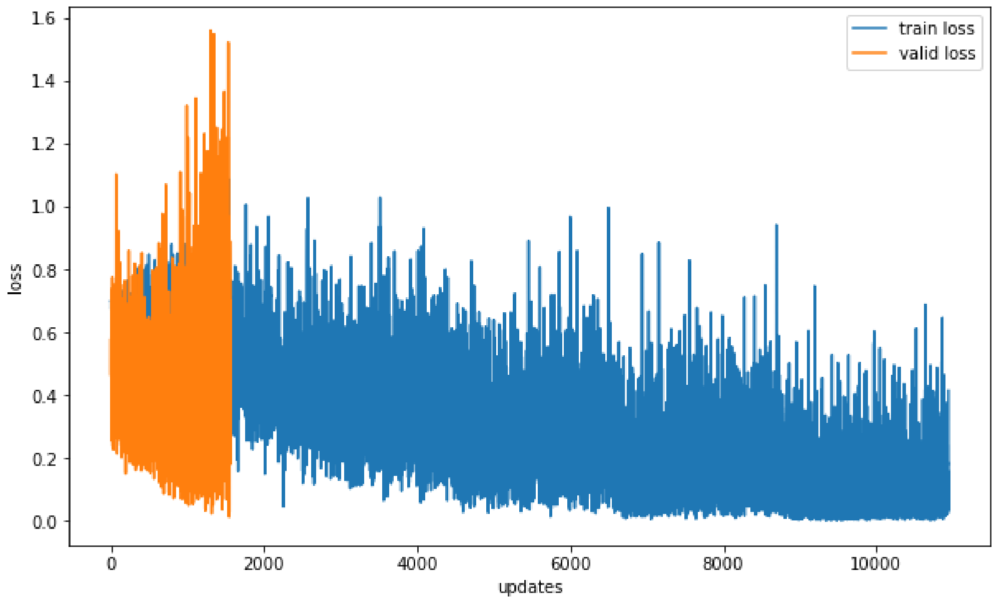
Question (f) Solution:

Length of vocabulary is 56590.

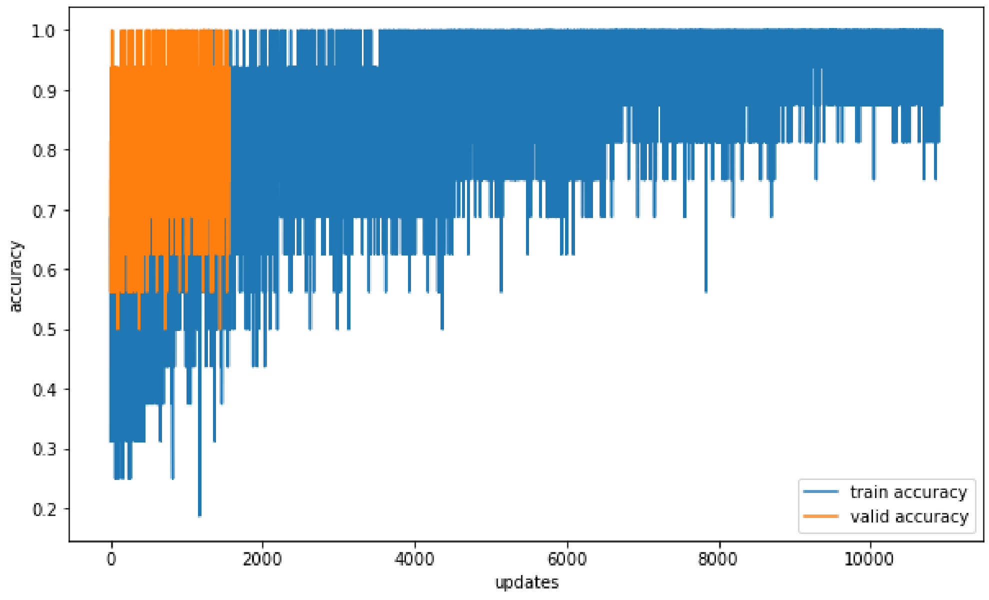
The model has 16, 066, 562 trainable parameters.

The testing accuracy is 82.7%.

The training/validation loss figure is shown below.



The training/validation accuracy figure is shown below.



3 Lab 2: Implement and train a Transformer for sentiment analysis (35 pts)

(b) (5 pts) Train the model for 5 epochs. Copy the plotted figures of training/validation loss/accuracy to your self-contained pdf report. What is your testing accuracy? What is your model size? (The provided code contains the plot function and computation of test accuracy. You just need to report the value of testing accuracy.)

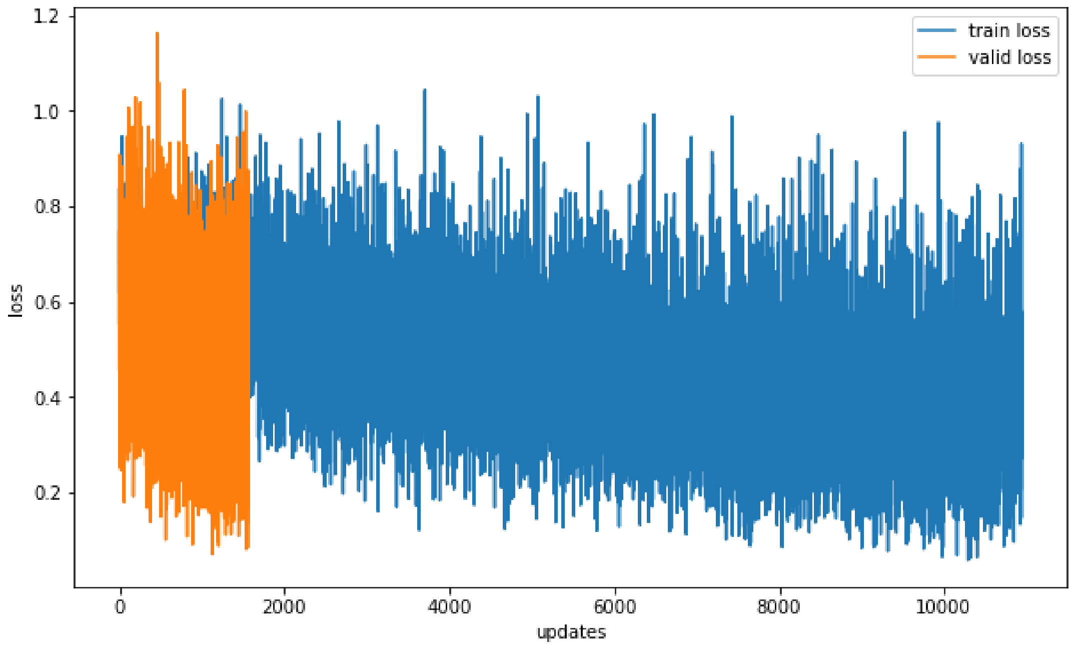
Question (b) Solution:

Length of vocabulary is 56590.

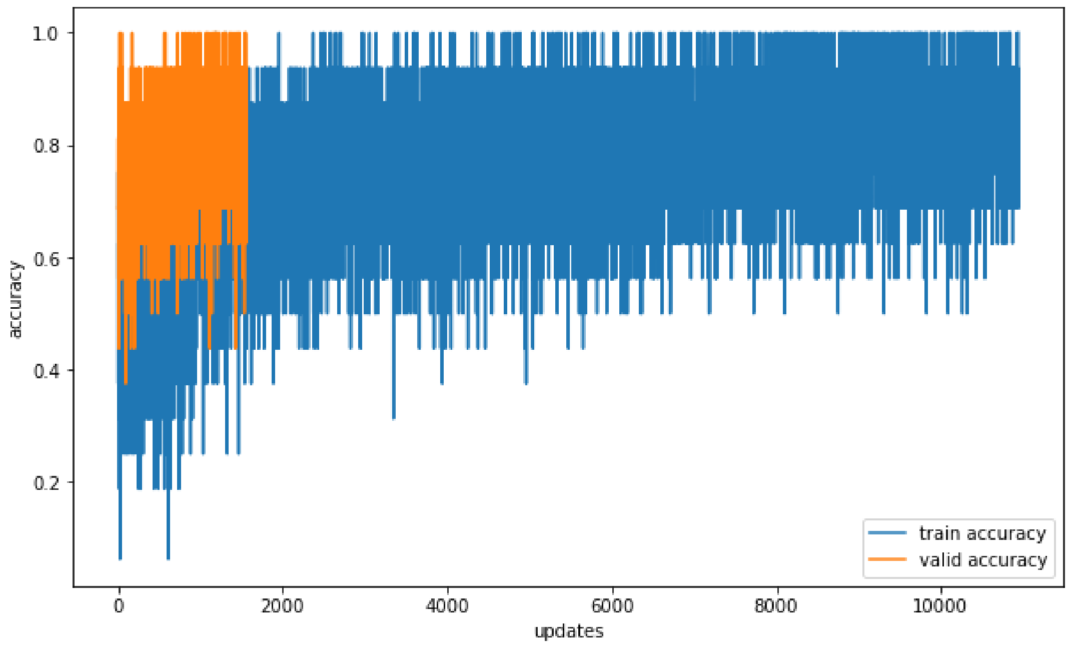
Model Size – The model has 16, 857, 602 trainable parameters.

The testing accuracy is 81.9%.

The training/validation loss figure is shown below.



The training/validation accuracy figure is shown below.



(d) (5 pts) Train the model for 5 epochs. Copy the plotted figures of training/validation loss/accuracy to your self-contained pdf report. What is your testing accuracy? What is your model size? What do you observe when comparing this with the original model's performance? (The provided code contains the plot function and computation of test accuracy. You just need to report the value of testing accuracy.)

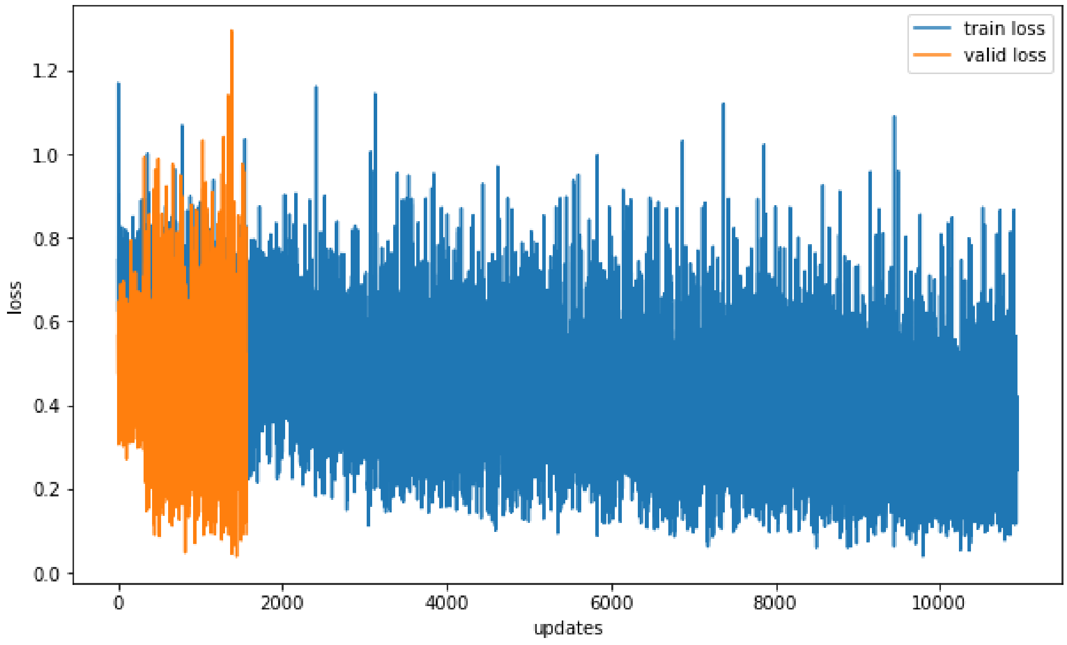
Question (d) Solution:

Length of vocabulary is 30522.

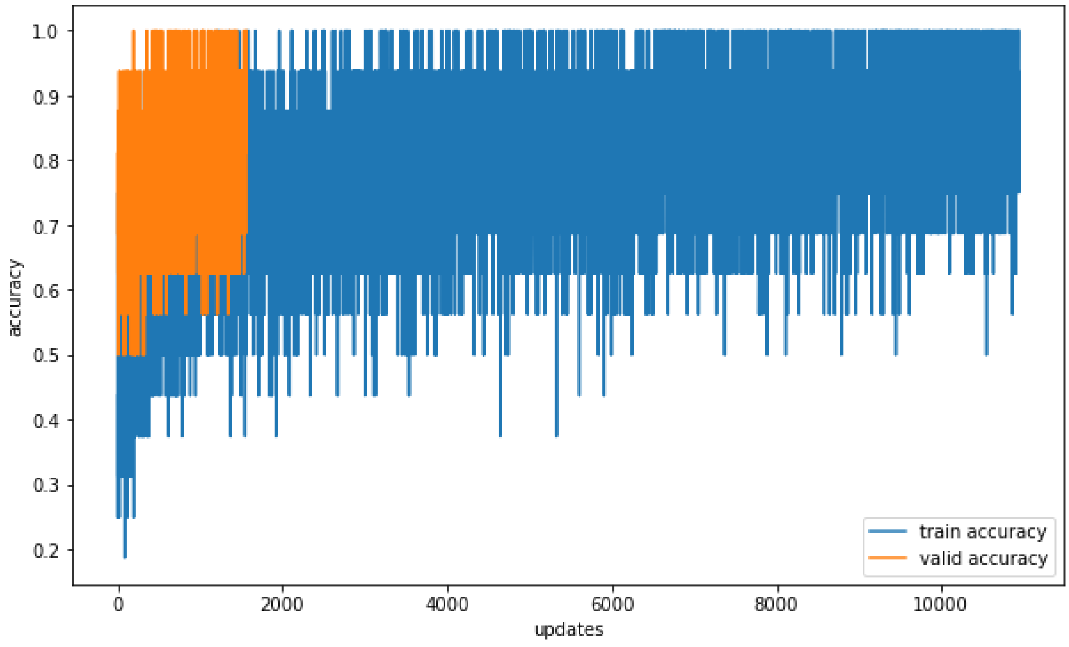
Model Size – The model has 10, 184, 194 trainable parameters.

The testing accuracy is 82.0%.

The training/validation loss figure is shown below.



The training/validation accuracy figure is shown below.



Comparing model's performance in Question (d) with that in Question (b), I observed the following differences. First, model (d) has much less trainable parameters as well as much smaller vocabulary size than model (b), which is the original model.

Second, model (d) achieves almost the same testing accuracy as model (b) after training both models for 5 epochs. However, the validation accuracy of model (b) achieves above 80% after 4 epochs, while model (d) achieves above 80% validation accuracy after only 2 epochs. This means that the model with the pre-trained BERT tokenizer converges faster than the original model, while maintaining relatively good, or even higher validation and testing accuracies.

(e) (5 pts) Replace the current `PositionalEmbedding` class in the `transformer.py` file with a trainable one. The current `PositionalEmbedding` uses the sine and cosine functions to initialize the positional embedding. These functions are deterministic. Could you change the class to use a trainable neural network to initialize the position embedding? For example, you can use the same embedding layer as the token embedding layer.

(f) (5 pts) Train the model with the updated `PositionalEmbedding` for 5 epochs. Copy the plotted figures of training/validation loss/accuracy to your self-contained pdf report. What is your testing accuracy? What is your model size? (The provided code contains the plot function and computation of test accuracy. You just need to report the value of testing accuracy.)

Note for Question (e): Just as "For example, you can use the same embedding layer as the token embedding layer" mentioned in Question (e), I replaced the original positional embedding layer with a trainable embedding layer, and this trainable embedding layer is the same as the token embedding layer.

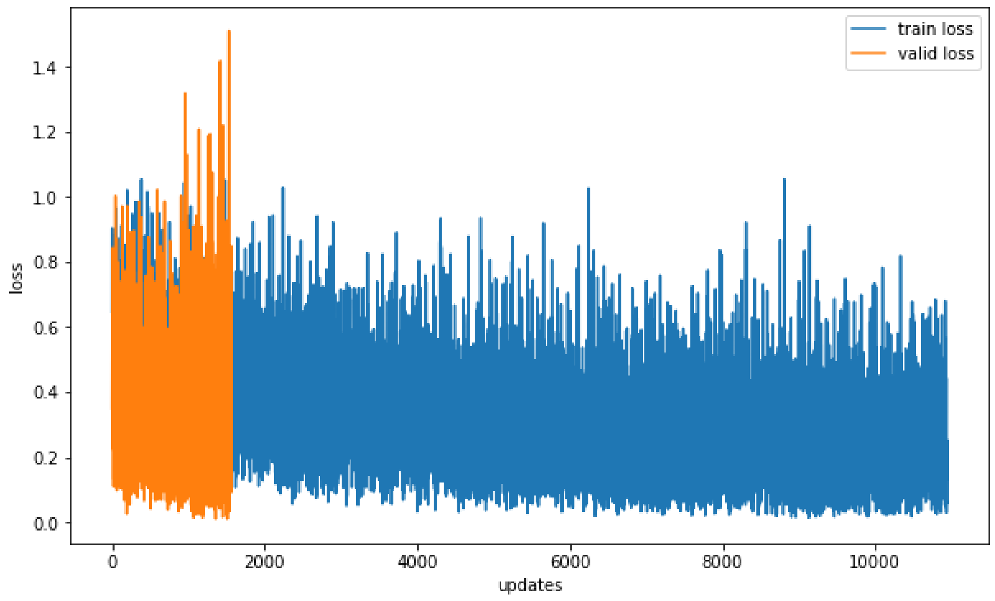
Question (f) Solution:

Length of vocabulary is 30522.

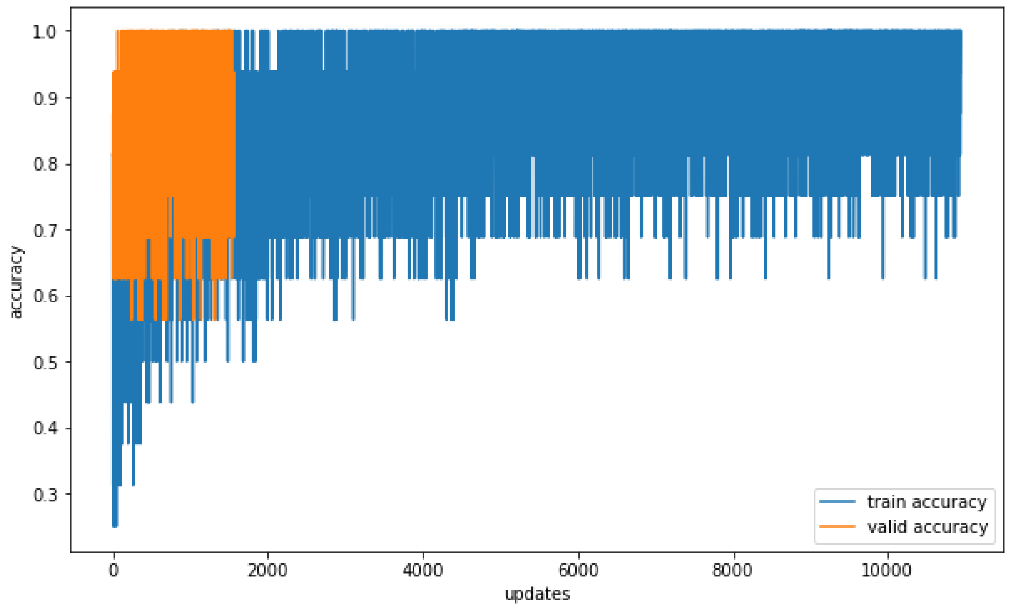
Model Size – The model has 17, 997, 826 trainable parameters.

The testing accuracy is 86.0%.

The training/validation loss figure is shown below.



The training/validation accuracy figure is shown below.



(g) (5 pts) Warm start learning rate schedule is a useful technique for training Transformers. The learning rate starts from a very small value at the beginning of the training. In the first few epochs, it is gradually increased to a peak value. Afterwards, the learning rate gradually decreases throughout the remaining training. Please implement the warm start learning rate schedule and run your code with it. You can choose your own learning rate values and schedule parameters such as the rising and falling speed (slope).

Question (g) Solution:

To implement warm start learning rate schedule, in the Lab2 Jupyter Notebook, I wrote my own function named "warm_start_lr". This function takes the value of current epoch as input and returns the updated learning rate correspondingly. The structure of this function looks a bit like switch-case in C programming. Since I do not plan to use many epochs for training, I think maybe this is a relatively straightforward way to show that the requirement of assigning different learning rates for different training epochs has been accomplished.

In specific, I decide to set the epoch number as 10, and set the peak value of learning rate as 3e-4 (0.0003), which is the recommended learning rate value. I have also tried 3e-3, but my model directly failed training even given only one epoch in which learning rate is 3e-3.

To warm up, my learning rate starts from a very small value, 6e-6, and gradually increases during the first 3 epochs. After reaching the peak value (3e-4) at epoch 4, my learning rate gradually decreases for the rest of epochs. The learning rate decreasing speed is a bit slower than the increasing speed.

Training, validation and testing results as well as a brief analysis is shown on the following pages.

Length of vocabulary is 30522.

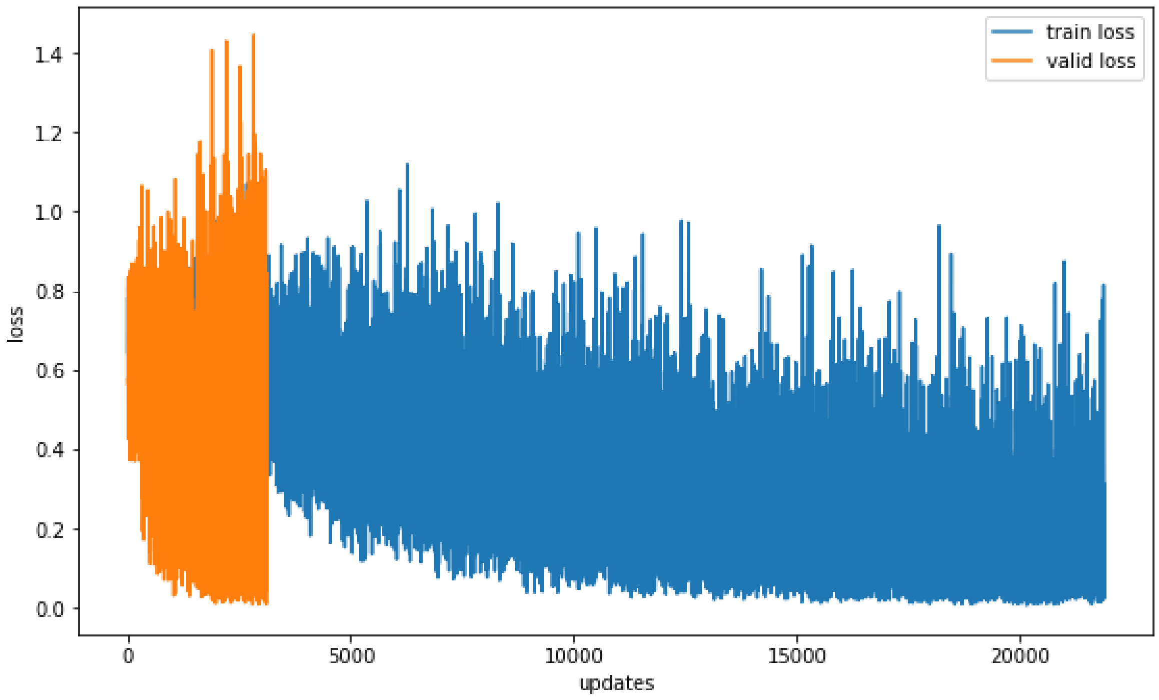
Model Size – The model has 17, 997, 826 trainable parameters.

The testing accuracy is 85.4%.

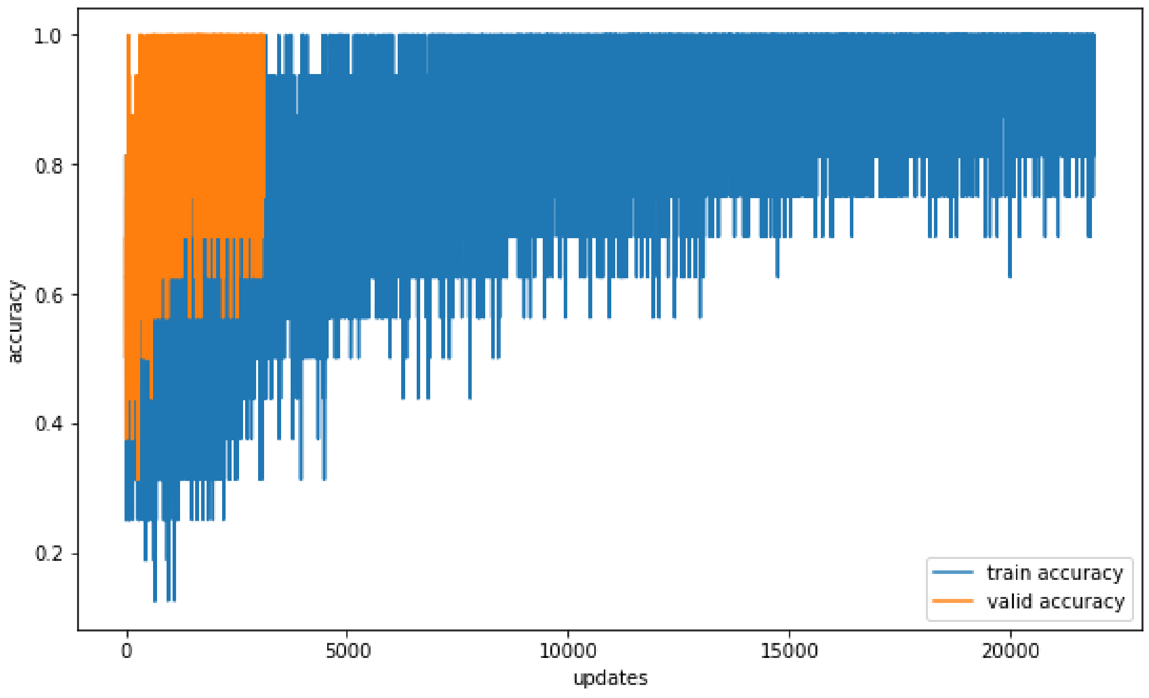
The training/validation accuracies and learning rate for each epoch is shown in the table below.

Epoch Number	Current Learning Rate	Training Accuracy	Validation Accuracy
1	6e-6 = 0.000006	55.0%	66.2%
2	6e-5 = 0.00006	70.8%	76.9%
3	1e-4 = 0.0001	79.6%	82.0%
4	3e-4 = 0.0003	83.2%	83.4%
5	2.5e-4 = 0.0025	87.1%	85.2%
6	2e-4 = 0.0002	89.3%	86.1%
7	1.5e-4 = 0.0015	91.0%	**86.7%**
8	1e-4 = 0.0001	92.4%	86.1%
9	0.5e-4 = 0.00005	93.5%	86.1%
10	6e-5 = 0.00006	93.8%	85.9%

The training/validation loss figure is shown below.



The training/validation accuracy figure is shown below.



Brief Analysis and Potential Insights

For the first 3 epochs, both training and validation accuracies increase slowly, which is expected because the learning rate is very small. It should be noticed that, however, at epoch 7, my model achieves 86.7% validation accuracy, which has never been reached for all my previous models. Perhaps this could give a few insights to demonstrate that warm start learning rate schedule could help models converge to better sub-optimal.