ECE Homework Assignment 3

True/False Questions:

1. **True.** The three core variables in the self-attention layer of Transformer models are key, query, and value, and we compute them from the input sequence. The input sequence is typically represented as a matrix of embeddings, where each embedding represents a word or token in the sequence.
2. **False.** In the self-attention layer of Transformer models, the attention is not denoted by the cosine similarity between the key and query. Attention is calculated using the key, query and value by this formula.

   $k_i = Kx_i,\ q_i = Qx_i,\ v_i = Vx_i$ where $K, Q, V \in \mathbb{R}^{d\times d}$

   score: $s_{ij} = q_i^T k_j$, attention: $a_{ij} = \frac{\exp(s_{ij})}{\sum_{j'} \exp(s_{ij'})}$, output$_i = \sum_j a_{ij} v_j$
3. **True.** In the self-attention layer of Transformer models, after obtaining the attention matrix, we need to further apply a normalization on it (e.g., layer normalization or batch normalization) as it helps the model train faster.
4. **False.** The encoder of Transformer does not learn autoregressively. It is a stack of self-attention layers, which allows it to learn long-range dependencies in the input sequence. However, the self-attention layers do not generate the output sequence one token at a time.
5. **False.** GPT's are pre-trained decoders while BERT's are pre-trained encoders. GPT's are based on transformer architecture but it uses a stack of transformer decoder layers. On the other hand, BERT's are also based on transformer architecture but it uses a stack of transformer encoder layers.
6. **True.** BERT's pre-training objectives include masked language model (MLM) for predicting the masked words and next sentence prediction (NSP) to Predict whether two text sequence are contiguous.
7. **False.** The two language models are few shot learners not zero shot learners, that means that with very few examples they can adapt to new tasks or domains.
8. **False.** Gradient clipping is a technique used to address the exploding gradient problem not the vanishing gradient problem. Gradient clipping is used to scale down the gradients when their magnitudes become too large for training.
9. **True.** Word embeddings can contain both positive as well as negative values. These values encode information about the word, its relationship with other words and context.
10. **False.** The memory cell of the LSTM is not computed as a weighted average. The forget gate is a sigmoid function so the memory cell cannot be a weighted average.

LAB 1: Recurrent Neural Network for Sentiment Analysis

Question (a):

Implementing my own dataloader function. Splitting the dataset into three sets: train, validation and test by 7:1:2 ratio.

```
def load_imdb(base_csv:str = '/content/drive/MyDrive/ECE 661 HW3/IMDBDataset.csv'):
    """
    Load the IMDB dataset
    :param base_csv: the path of the dataset file.
    :return: train, validation and test set.
    """
    # Add your code here.
    df=pd.read_csv(base_csv)


    x_train,x_test,y_train,y_test= train_test_split(df['review'],df['sentiment'], test_size=0.2,random_state=42)
    x_train,x_valid,y_train,y_valid =train_test_split(x_train,y_train,test_size=1/8,random_state=42)

    print(f'shape of train data is {x_train.shape}')
    print(f'shape of test data is {x_test.shape}')
    print(f'shape of valid data is {x_valid.shape}')
    return x_train, x_valid, x_test, y_train, y_valid, y_test
```

```
x_train,x_valid,x_test,y_train,y_valid,y_test=load_imdb()
```

```
shape of train data is (35000,)
shape of test data is (10000,)
shape of valid data is (5000,)
```

Question (b):

The build_vocab function is code that processes the input data, counts word frequencies and filters based on min_freq, and builds a vocabulary where words are assigned unique indices. This code also filters out STOP_WORDS.

```
def build_vocab(x_train:list, min_freq: int=5, hparams=None) -> dict:
    """
    build a vocabulary based on the training corpus.
    :param x_train:  List. The training corpus. Each sample in the list is a string of text.
    :param min_freq: Int. The frequency threshold for selecting words.
    :return: dictionary {word:index}
    """
    # Add your code here. Your code should assign corpus with a list of words.
    corpus = collections.defaultdict(int)
    hparams = HyperParams()
    top_words = hparams.STOP_WORDS


    for x in x_train:
      words =  re.findall('[a-zA-Z]+',re.sub(r'<.*?>', '', x))
      for word in words:
        if word not in top_words:
          corpus[word.lower()]+=1

    # sorting on the basis of most common words
    # corpus_ = sorted(corpus, key=corpus.get, reverse=True)[:1000]
    corpus_ = [word for word, freq in corpus.items() if freq >= min_freq]
    # creating a dict
    vocab = {w:i+2 for i, w in enumerate(corpus_)}
    vocab[hparams.PAD_TOKEN] = hparams.PAD_INDEX
    vocab[hparams.UNK_TOKEN] = hparams.UNK_INDEX
    return vocab
```

Question (c):

The tokenize function transforms a given text example into a list of token indices using a provided vocabulary.

```python
def tokenize(vocab: dict, hparams, example: str) -> list:
    """
    Tokenize the given example string into a list of token indices.
    :param vocab: dict, the vocabulary.
    :param hparams: HyperParams object.
    :param example: a string of text.
    :return: a list of token indices.
    """
    words = re.findall('[a-zA-Z]+', re.sub(r'<.*?>', '', example))
    hparams = HyperParams()

    tokens = []
    for word in words:
      if word.lower() in vocab:
        tokens.append(vocab[word.lower()])
      elif word.lower() in stopwords:
        continue
      else:
        tokens.append(hparams.UNK_INDEX)

    return tokens
```

Question (d):

```python
def __getitem__(self, idx: int):
    """
    Return the tokenized review and label by the given index.
    :param idx: index of the sample.
    :return: a dictionary containing three keys: 'ids', 'length', 'label' which represent the list of token ids, the length of the sequence, the binary label.
    """
    text = self.x.iloc[idx]

    tokens = tokenize(vocab=self.vocab, hparams=hparams,example=text)[:self.max_length]
    label = 1 if self.y.iloc[idx] == 'positive' else 0


    return {
    'ids': tokens,
    'length': len(tokens),
    'label': label
     }
```

Question (e):

(a) LSTM model

```python
class LSTM(nn.Module):
    def __init__(
        self,
        vocab_size: int,
        embedding_dim: int,
        hidden_dim: int,
        output_dim: int,
        n_layers: int,
        dropout_rate: float,
        pad_index: int,
        bidirectional: bool,
        **kwargs):
        """
        Create a LSTM model for classification.
        :param vocab_size: size of the vocabulary
        :param embedding_dim: dimension of embeddings
        :param hidden_dim: dimension of hidden features
        :param output_dim: dimension of the output layer which equals to the number of labels.
        :param n_layers: number of layers.
        :param dropout_rate: dropout rate.
        :param pad_index: index of the padding token
        """
        super().__init__()
        # Add your code here. Initializing each layer by the given arguments.
        hparams = HyperParams()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_index)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout= dropout_rate, bidirectional=bidirectional)
        self.dropout = nn.Dropout(dropout_rate)
        self.fc = nn.Linear(hidden_dim *hparams.MAX_LENGTH* 2 if bidirectional else hidden_dim * hparams.MAX_LENGTH, output_dim)
```

(b) Forward function

```python
def forward(self, ids:torch.Tensor, length:torch.Tensor):
    """
    Feed the given token ids to the model.
    :param ids: [batch size, seq len] batch of token ids.
    :param length: [batch size] batch of length of the token ids.
    :return: prediction of size [batch size, output dim].
    """
    # Add your code here.
    out = self.embedding(ids)
    out  = nn.utils.rnn.pack_padded_sequence(out, length, batch_first=True, enforce_sorted=False)
    out, _ = self.lstm(out)
    unpacked_out, unpacked_lengths = nn.utils.rnn.pad_packed_sequence(out, batch_first=True)
    out = unpacked_out.view( unpacked_out.shape[0], -1)
    out = self.fc(out)

    return out
```

Question (f):

Test accuracy: 0.514

Test loss:0.692

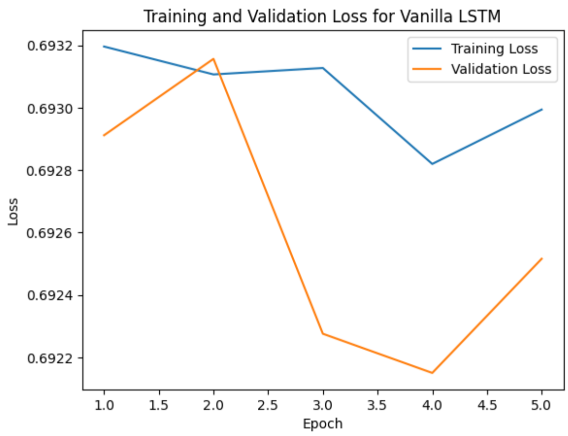
```
org_hyperparams = HyperParams()
_ = train_and_test_model_with_hparams(org_hyperparams, "lstm_1layer_base_sgd_e32_h100")
```

```
shape of train data is (35000,)
shape of test data is (10000,)
shape of valid data is (5000,)
Length of vocabulary is 33573
The model has 125,975 trainable parameters
training...: 100%|██████████| 365/365 [00:25<00:00, 14.22it/s]
evaluating...: 100%|██████████| 52/52 [00:03<00:00, 14.10it/s]
Saving ...
epoch: 1
train_loss: 0.693, train_acc: 0.505
valid_loss: 0.693, valid_acc: 0.509
training...: 100%|██████████| 365/365 [00:25<00:00, 14.06it/s]
evaluating...: 100%|██████████| 52/52 [00:02<00:00, 17.48it/s]
epoch: 2
train_loss: 0.693, train_acc: 0.509
valid_loss: 0.693, valid_acc: 0.509
training...: 100%|██████████| 365/365 [00:26<00:00, 13.66it/s]
evaluating...: 100%|██████████| 52/52 [00:02<00:00, 21.10it/s]
Saving ...
epoch: 3
train_loss: 0.693, train_acc: 0.506
valid_loss: 0.692, valid_acc: 0.512
training...: 100%|██████████| 365/365 [00:26<00:00, 13.66it/s]
evaluating...: 100%|██████████| 52/52 [00:02<00:00, 22.32it/s]
Saving ...
```

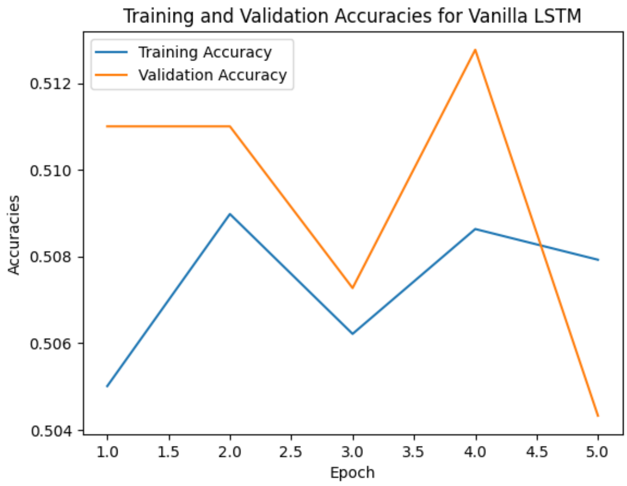
```
epoch: 4
train_loss: 0.693, train_acc: 0.509
valid_loss: 0.692, valid_acc: 0.518
training...: 100%|██████████| 365/365 [00:27<00:00, 13.47it/s]
evaluating...: 100%|██████████| 52/52 [00:02<00:00, 22.63it/s]
epoch: 5
train_loss: 0.693, train_acc: 0.508
valid_loss: 0.692, valid_acc: 0.509
evaluating...: 100%|██████████| 105/105 [00:04<00:00, 21.78it/s]
test_loss: 0.692, test_acc: 0.514
```

<u>Training and Validation Loss for Vanilla LSTM:</u>

The below graph is a plot of the Training and Validation Losses for the Vanilla LSTM model. The training loss is seen to be decreasing after the first epoch, the training loss is the highest for the first epoch and then lowers after that. The validation loss increases after the first epoch but reduces after the second epoch drastically. We can also observe that the validation loss is increases slightly for the last epoch.



Training and Validation Accuracies for Vanilla LSTM:



Question (g):

(a) GRU Model

```
class GRU(nn.Module):
    def __init__(
        self,
        vocab_size: int,
        embedding_dim: int,
        hidden_dim: int,
        output_dim: int,
        n_layers: int,
        dropout_rate: float,
        pad_index: int,
        bidirectional: bool,
        **kwargs):
        """
        Create a LSTM model for classification.
        :param vocab_size: size of the vocabulary
        :param embedding_dim: dimension of embeddings
        :param hidden_dim: dimension of hidden features
        :param output_dim: dimension of the output layer which equals to the number of labels.
        :param n_layers: number of layers.
        :param dropout_rate: dropout rate.
        :param pad_index: index of the padding token.we
        """
        super().__init__()
        # Add your code here. Initializing each layer by the given arguments.
        self.max_length = kwargs['max_length']
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_index)
        self.gru = nn.GRU(embedding_dim, hidden_dim, n_layers, dropout= dropout_rate, bidirectional=bidirectional)
        self.fc = nn.Linear(hidden_dim *self.max_length* 2 if bidirectional else hidden_dim * self.max_length, output_dim)
```

(b) Forward Function

```
def forward(self, ids:torch.Tensor, length:torch.Tensor):
    """
    Feed the given token ids to the model.
    :param ids: [batch size, seq len] batch of token ids.
    :param length: [batch size] batch of length of the token ids.
    :return: prediction of size [batch size, output dim].
    """
    # Add your code here.
    out = self.embedding(ids)
    out  = nn.utils.rnn.pack_padded_sequence(out, length, batch_first=True, enforce_sorted=False)
    out, _ = self.gru(out)
    unpacked_out, unpacked_lengths = nn.utils.rnn.pad_packed_sequence(out, batch_first=True, total_length = self.max_length)
    out = unpacked_out.view( unpacked_out.shape[0], -1)
    out = self.fc(out)

    return out
```

Question (h):

Test accuracy: 0.504

Test Loss: 0.693

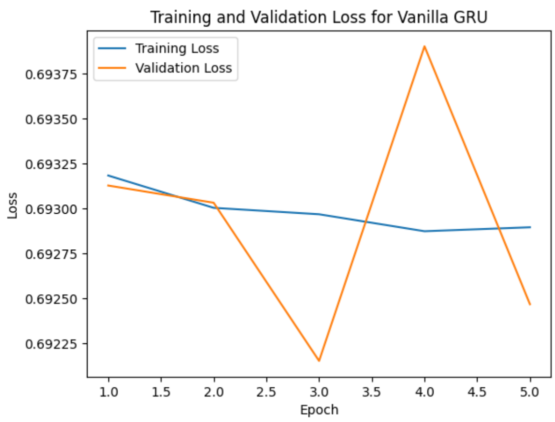
```
org_hyperparams = HyperParams()
_ = train_and_test_model_with_hparams(org_hyperparams, "gru_1layer_base_sgd_e32_h100", override_models_with_gru=True)
```

```
shape of train data is (35000,)
shape of test data is (10000,)
shape of valid data is (5000,)
Length of vocabulary is 33573
The model has 115,675 trainable parameters
training...: 100%|██████████| 365/365 [00:27<00:00, 13.45it/s]
evaluating...: 100%|██████████| 52/52 [00:03<00:00, 16.55it/s]
Saving ...
epoch: 1
train_loss: 0.693, train_acc: 0.505
valid_loss: 0.693, valid_acc: 0.510
training...: 100%|██████████| 365/365 [00:26<00:00, 13.55it/s]
evaluating...: 100%|██████████| 52/52 [00:03<00:00, 15.23it/s]
Saving ...
epoch: 2
train_loss: 0.693, train_acc: 0.507
valid_loss: 0.693, valid_acc: 0.510
training...: 100%|██████████| 365/365 [00:27<00:00, 13.34it/s]
evaluating...: 100%|██████████| 52/52 [00:02<00:00, 21.35it/s]
Saving ...
epoch: 3
train_loss: 0.693, train_acc: 0.510
valid_loss: 0.692, valid_acc: 0.509
training...: 100%|██████████| 365/365 [00:27<00:00, 13.48it/s]
evaluating...: 100%|██████████| 52/52 [00:02<00:00, 22.82it/s]
epoch: 4
```

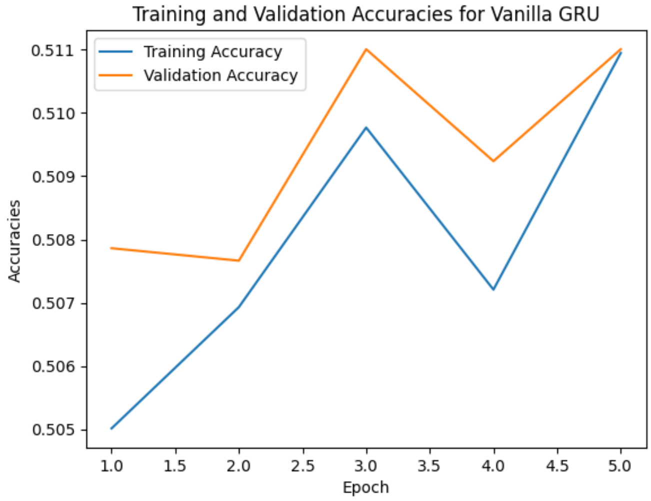
```
epoch: 4
train_loss: 0.693, train_acc: 0.507
valid_loss: 0.694, valid_acc: 0.509
training...: 100%|██████████| 365/365 [00:27<00:00, 13.37it/s]
evaluating...: 100%|██████████| 52/52 [00:02<00:00, 23.21it/s]
epoch: 5
train_loss: 0.693, train_acc: 0.511
valid_loss: 0.693, valid_acc: 0.509
evaluating...: 100%|██████████| 105/105 [00:04<00:00, 22.57it/s]
test_loss: 0.693, test_acc: 0.504
```

<u>Training and Validation Loss for Vanilla GRU:</u>

The below graph is a plot of the training and validation losses for the vanilla GRU, we can observe from the graph that the training loss is slowly decreasing after the first epoch. The validation loss decreases up to the third epoch, increases drastically for the fourth epoch and decreases again.



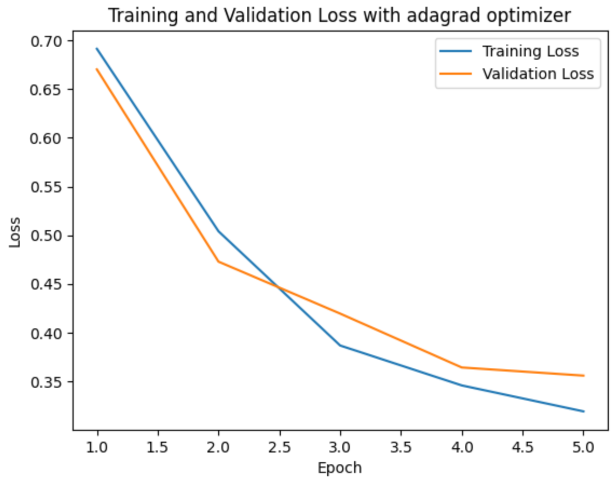
Training and Validation Accuracies for Vanilla GRU:

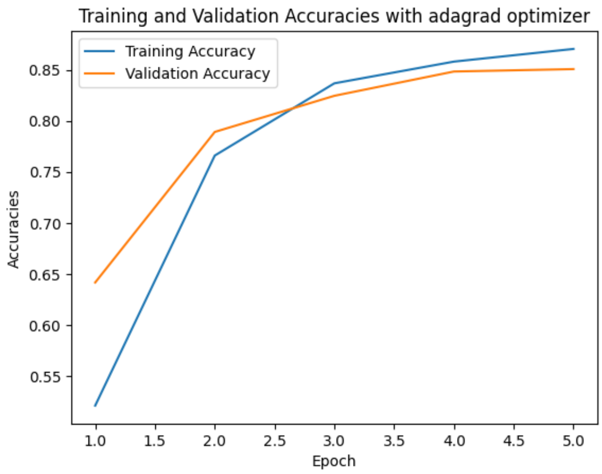


LAB 2: Training and Improving RNN

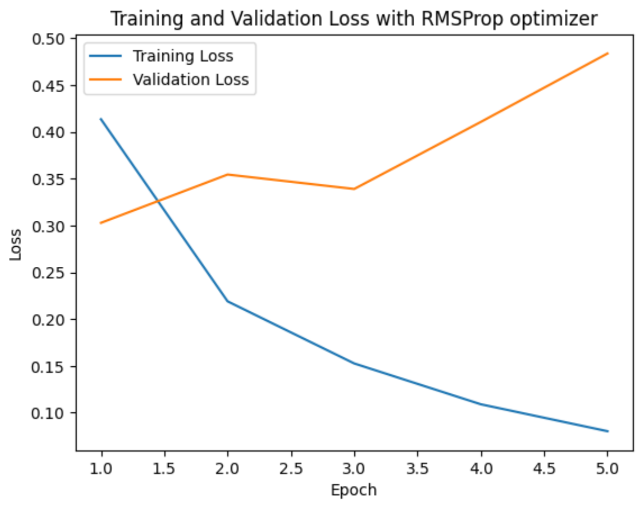
Question (a): LSTM Model

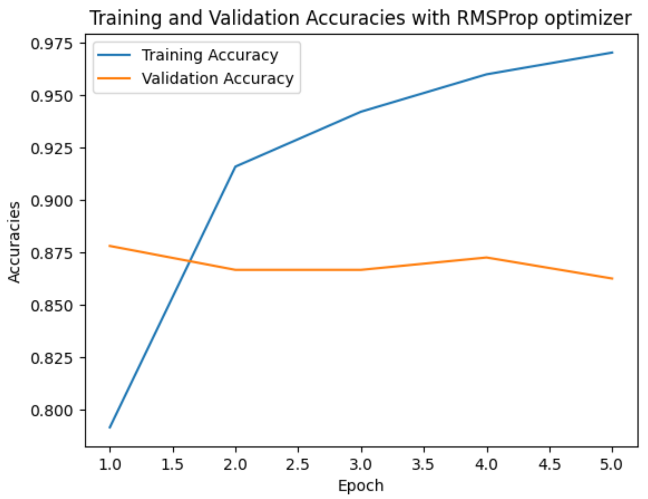
Using Adagrad Optimizer:



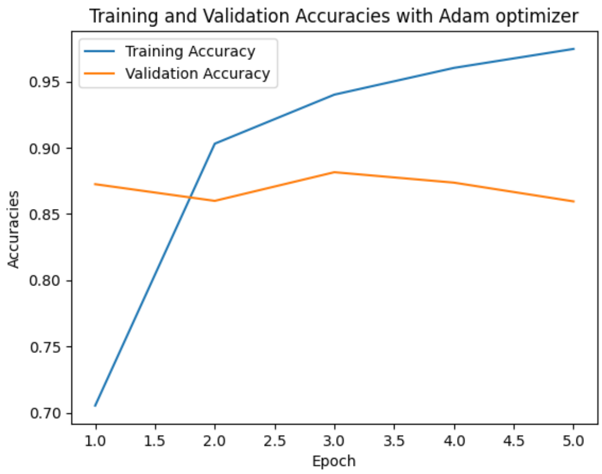


Using RMSProp Optimizer:





Using Adam Optimizer:

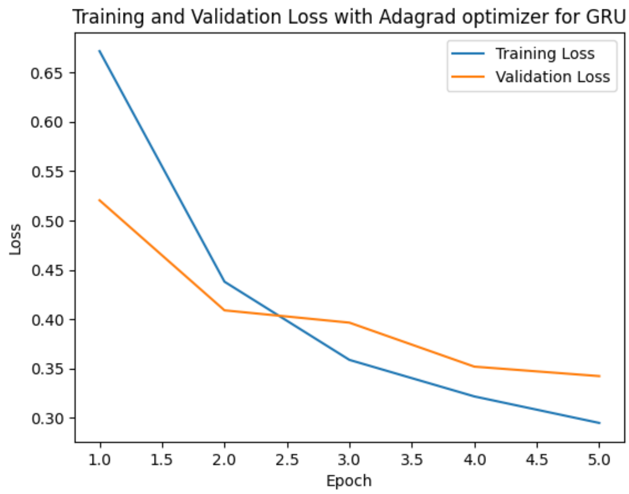


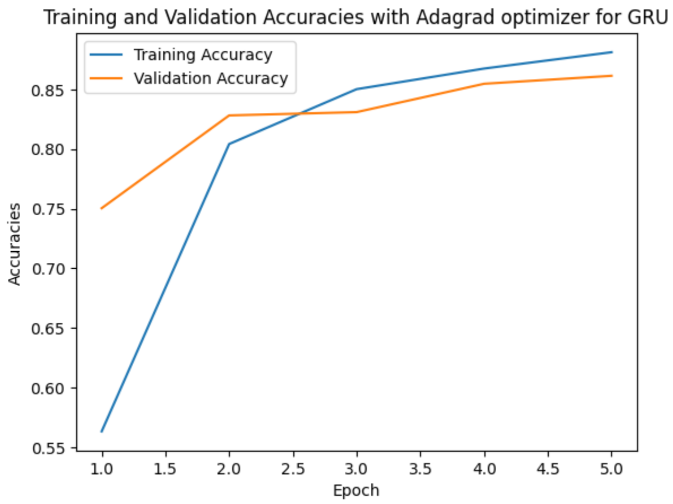
Optimizer	Test Loss	Test Accuracy
SGD Optimizer	0.692	0.514
Adagrad Optimizer	0.354	0.852
RMSProp Optimizer	0.292	0.877
Adam Optimizer	0.294	0.885

Based on the table and the Test Accuracies obtained using the different optimizers for the LSTM model, we can see that the model performs the best with Adam Optimizer. The performance is the lowest with the SGD optimizer.

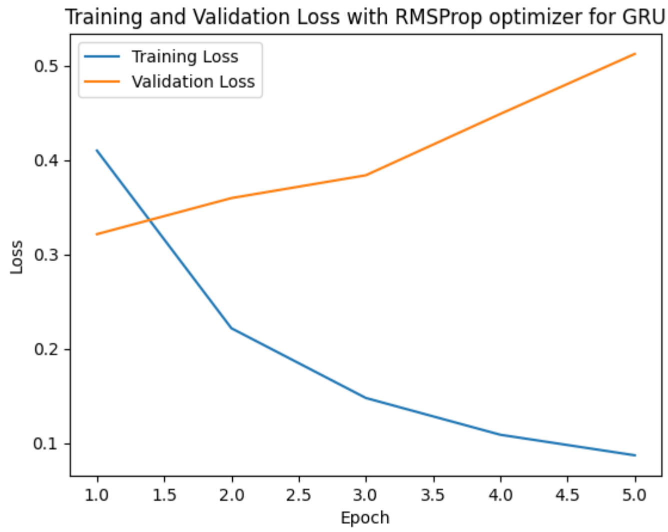
Question (b): GRU Model

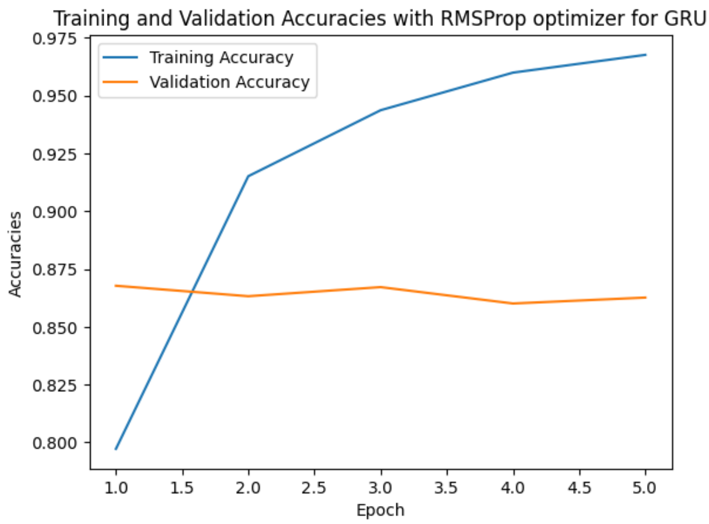
<u>Using Adagrad Optimizer:</u>



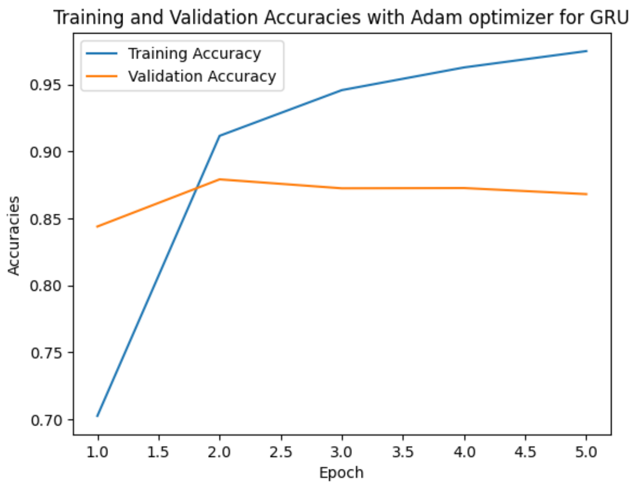


Using RMSProp Optimizer:





<u>Using Adam Optimizer:</u>



Optimizer	Test Loss	Test Accuracy
SGD Optimizer	0.693	0.504
Adagrad Optimizer	0.338	0.860
RMSProp Optimizer	0.318	0.868
Adam Optimizer	0.288	0.883

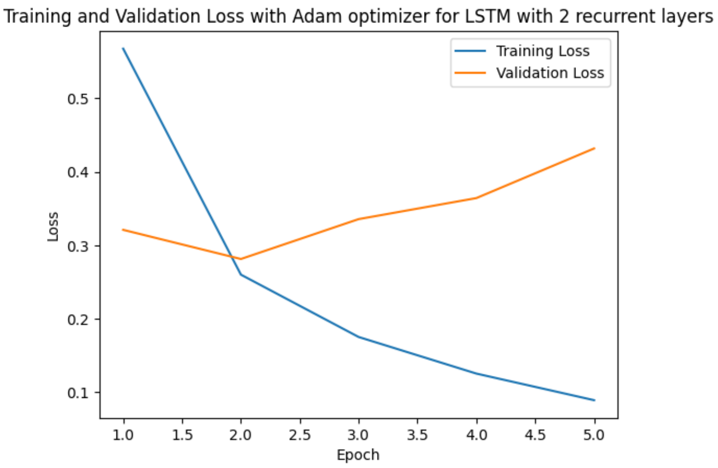
Based on the table and the Test Accuracies obtained using the different optimizers for the GRU model, we can see that the model performs the best with Adam Optimizer. The performance is the lowest with the SGD optimizer.

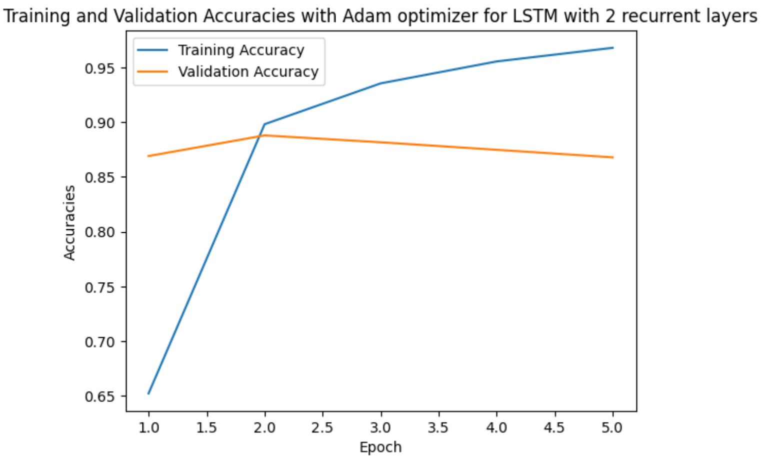
Comparing LSTM and GRU:

When comparing the results for both the models, GRU shows better results as compared to LSTM when using the adagrad optimizer. With SGD, Adam and RMSProp we can observe that LSTM performs better with better efficiency.

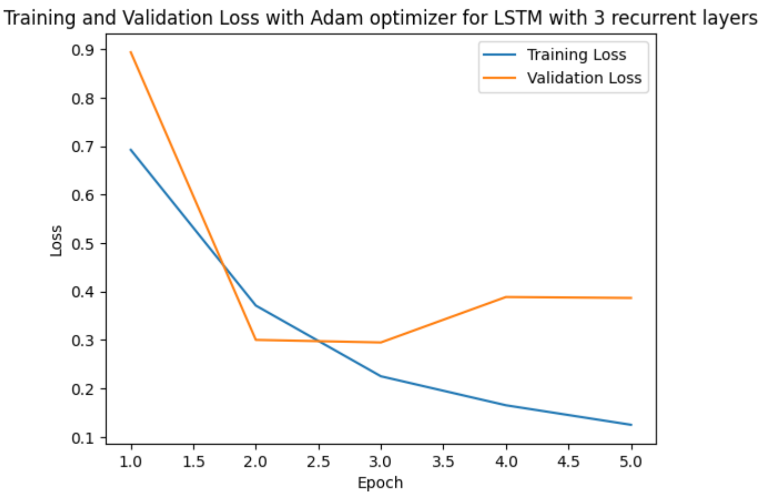
Question (c):

<u>Using LSTM with the Adam Optimizer and 2 recurrent layers:</u>

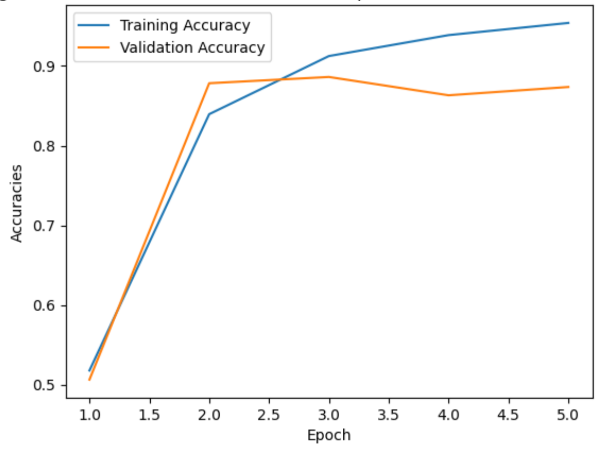




<u>Using LSTM with the Adam Optimizer and 3 recurrent layers:</u>



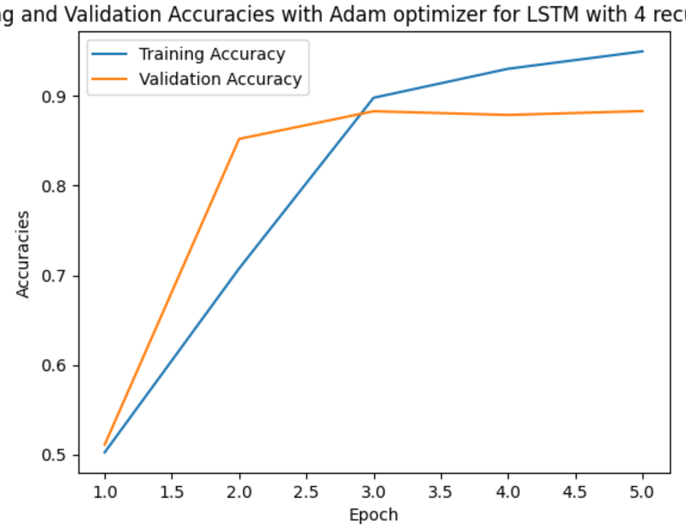
Training and Validation Accuracies with Adam optimizer for LSTM with 3 recurrent layers



Using LSTM with the Adam Optimizer and 4 recurrent layers:

Training and Validation Loss with Adam optimizer for LSTM with 4 recurrent layers

Training and Validation Accuracies with Adam optimizer for LSTM with 4 recurrent layers

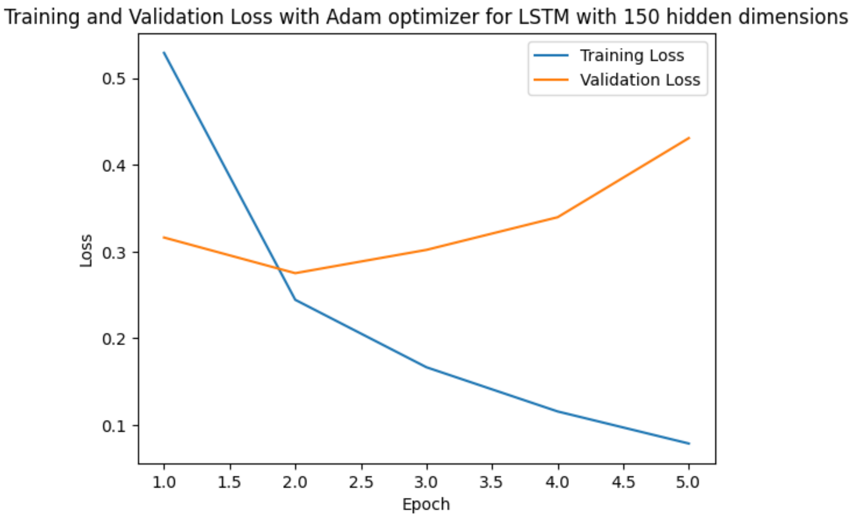


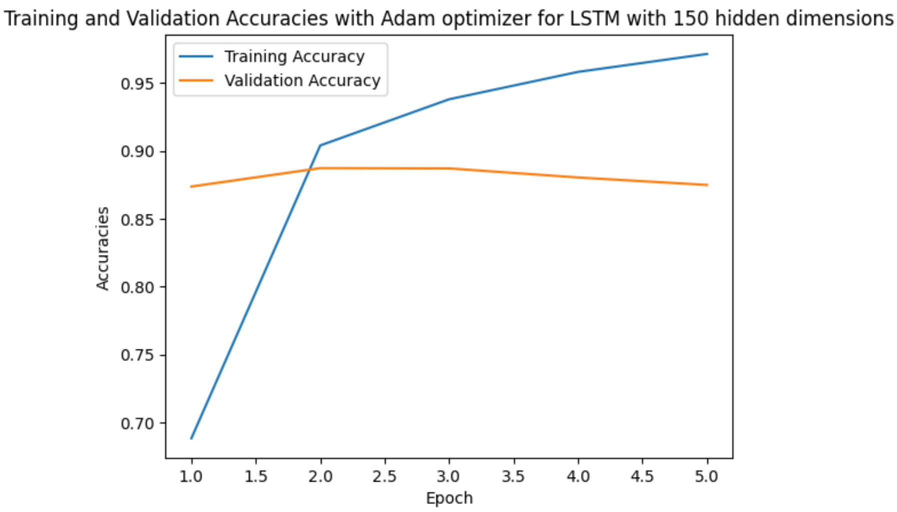
N Layers	Test Loss	Test Accuracy
1	0.294	0.885
2	0.275	0.887
3	0.284	0.885
4	0.298	0.880

Based on the table and the graphs plotted above, we can see that when we use LSTM with the adam optimizer, the model performs the best with 2 recurrent layers, the performance of the model reduces as you increase the number of recurrent layers.

Question (d):

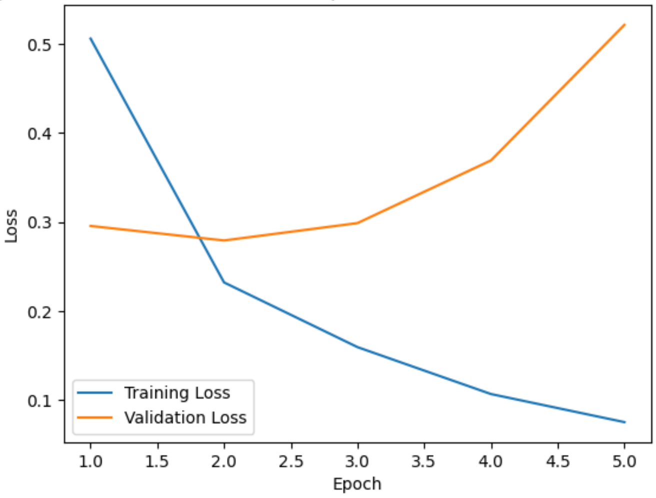
<u>Using LSTM with the Adam Optimizer with 150 hidden dimensions:</u>



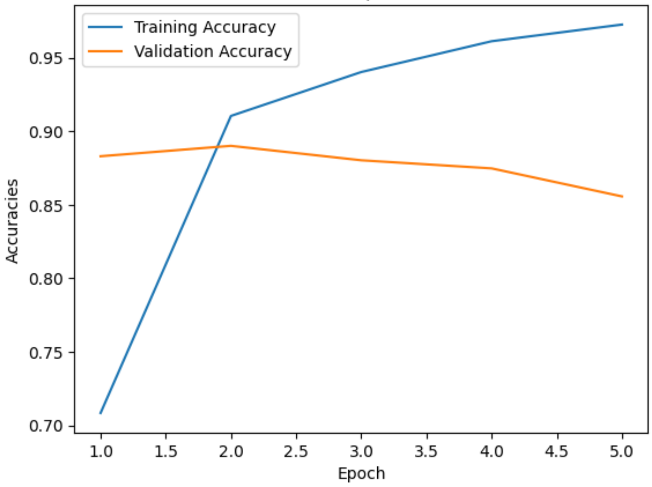


Using LSTM with the Adam Optimizer with 220 hidden dimensions:

Training and Validation Loss with Adam optimizer for LSTM with 220 hidden dimensions

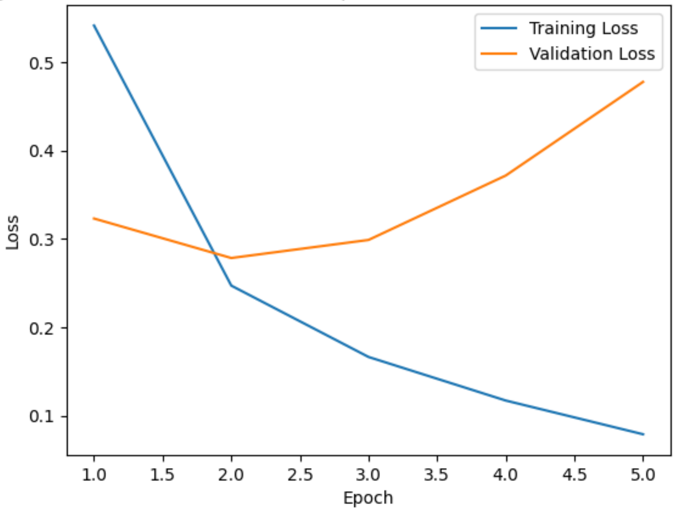


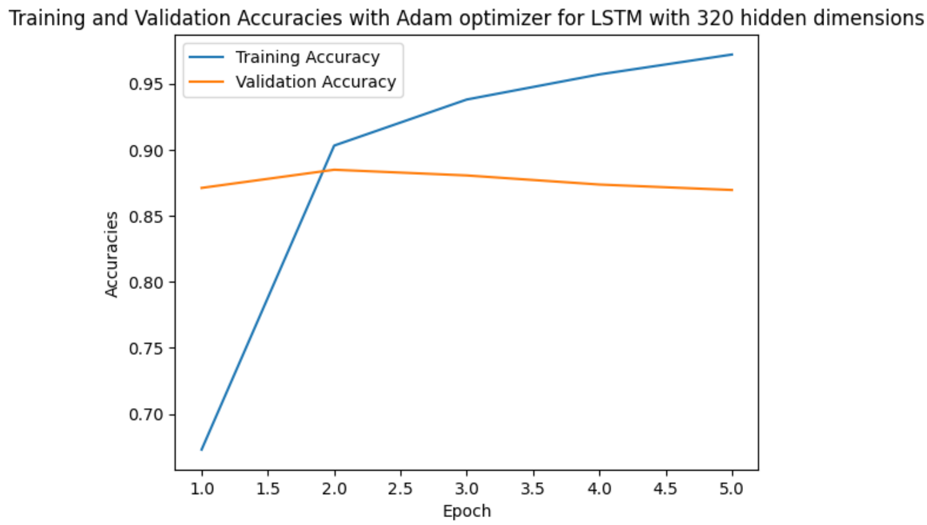
Training and Validation Accuracies with Adam optimizer for LSTM with 220 hidden dimensions



Using LSTM with the Adam Optimizer with 320 hidden dimensions:

Training and Validation Loss with Adam optimizer for LSTM with 320 hidden dimensions





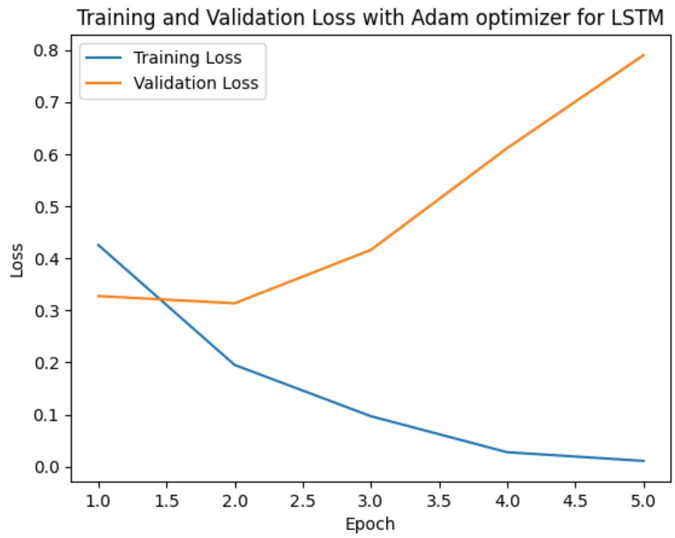
No. of hidden dimensions	Test Loss	Test Accuracy
100	0.294	0.885
150	0.271	0.890
220	0.273	0.888
320	0.271	0.888

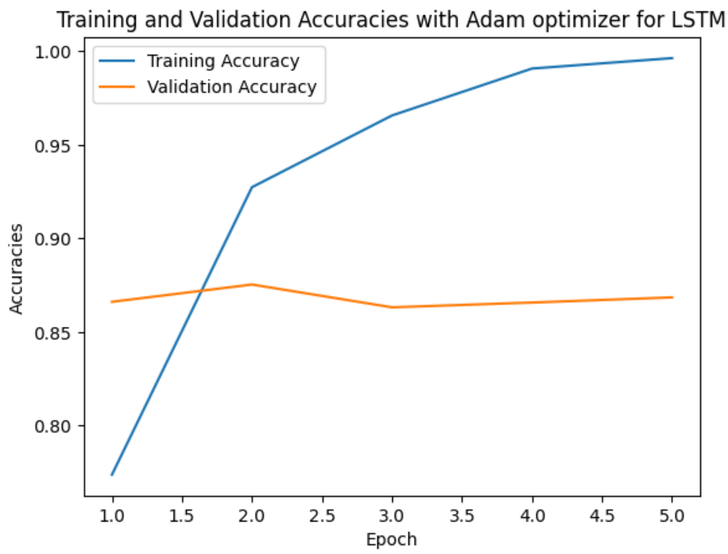
Based on the table and the graphs plotted above, we can see that when we use LSTM with the adam optimizer, the model performs the best with 150 hidden dimensions. The performance decreases as the number of dimensions increases.

Question (e):

Test loss: 0.307,

Test Accuracy: 0.878





Based on the plots and the accuracy, we can observe that the LSTM model performs more poorly when you increase the number of embedding dimensions as compared to when the dimensions are lesser.

Question (f):

Compound scaling:

Learning rate=0.01

Optimizer= “Adam”

No. of hidden dimensions=150

No. of N_Layers=2

No. of Embedded Dimension=150

```
adam_hyperparams = HyperParams()
adam_hyperparams.LR = 0.001
adam_hyperparams.OPTIM = "adam"
adam_hyperparams.HIDDEN_DIM = 150
adam_hyperparams.N_LAYERS=2
adam_hyperparams.EMBEDDING_DIM = 150
adam_optim_comp= train_and_test_model_with_hparams(adam_hyperparams, "lstm_1layer_base_adam_comp")
```

```
shape of train data is (35000,)
shape of test data is (10000,)
shape of valid data is (5000,)
Length of vocabulary is 33573
The model has 5,475,152 trainable parameters
training...: 100%|██████████| 365/365 [00:35<00:00, 10.23it/s]
evaluating...: 100%|██████████| 53/53 [00:04<00:00, 13.18it/s]
Saving ...
epoch: 1
train_loss: 0.424, train_acc: 0.781
valid_loss: 0.303, valid_acc: 0.876
training...: 100%|██████████| 365/365 [00:35<00:00, 10.20it/s]
evaluating...: 100%|██████████| 53/53 [00:02<00:00, 20.05it/s]
epoch: 2
train_loss: 0.202, train_acc: 0.923
valid_loss: 0.311, valid_acc: 0.875
training...: 100%|██████████| 365/365 [00:36<00:00, 10.12it/s]
evaluating...: 100%|██████████| 53/53 [00:02<00:00, 18.97it/s]
epoch: 3
train_loss: 0.122, train_acc: 0.956
valid_loss: 0.392, valid_acc: 0.871
training...: 100%|██████████| 365/365 [00:35<00:00, 10.34it/s]
evaluating...: 100%|██████████| 53/53 [00:03<00:00, 14.17it/s]
```

```
epoch: 4
train_loss: 0.080, train_acc: 0.971
valid_loss: 0.420, valid_acc: 0.872
training...: 100%|██████████| 365/365 [00:35<00:00, 10.37it/s]
evaluating...: 100%|██████████| 53/53 [00:02<00:00, 19.19it/s]
epoch: 5
train_loss: 0.051, train_acc: 0.981
valid_loss: 0.625, valid_acc: 0.873
evaluating...: 100%|██████████| 105/105 [00:07<00:00, 14.12it/s]
test_loss: 0.303, test_acc: 0.875
```

For compound scaling I have selected the above values to train my LSTM model. These values were selected based on the results I achieved throughout this homework assignment.