ECE 661: Homework #3 Understand and Implement Sequence Models

Hai "Helen" Li
ECE Department, Duke University — Octobor 2, 2023

Objectives

Homework #3 covers the contents of Lectures 09∼10. This assignment includes basic knowledge about sequence models (i.e., RNNs and Transformers), detailed instructions on how to implement the core of the sequence models and the training pipeline for training sentiment analysis on the IMDB dataset. In this assignment, you will gain hands-on experience in training an LSTM and a GRU on the IMDB dataset, while also learning techniques for improving the performance of your models.

Some parts of this assignment require advanced features in the latest PyTorch version, which are unfortunately not available on our JupyterLab server. Thus **using Google CoLab is strongly encouraged** for finishing the lab questions. A brief introduction on setting up and using CoLab is provided at the beginning of Lab 1. When conducting the lab projects, actively referring to the **NumPy/PyTorch tutorial** slides on Sakai for the environment setup and instructions for NumPy/PyTorch utilities can be very helpful.

Warning: You are asked to complete the assignment independently.

This lab has 100 points plus 15 bonus points, yet your final score cannot exceed 100 points. The sub- mission deadline will be **11:59pm, Wednesday, October 18**. For each lab, we provide one or multiple notebooks for you to start with. Your task is to fill in the missing modules in these notebooks. No new notebook or scripts need to be created. You will need to submit two independent files including:

1. A self-contained PDF report, which provides answers to all the conceptual questions and clearly demonstrates all your lab results and observations. Remember, **do NOT generate PDF from your Jupiter notebook to serve as the report**, which can increase the TA's burden of grading. Besides, you pdf file should also contain the plotted figures (instead of submit separately). You can upload a saved figure or a screenshot of the figure to the pdf report.

2. LabRNN.ipynb, a single Jupiter notebook that includes your work on Lab 1 and Lab 2.

Note that 20 percent of the grade will be deducted if the submissions doesn't follow the above guidance.

1 True/False Questions (30 pts)

For each question, please provide a short explanation to support your judgment. Note that for questions regarding BERT and GPT, the answers are not fully contained in the lecture. You may need to go thorugh the original paper to find the solutions.

Problem 1.1 (3 pts) In the self-attention layer of Transformer models, we compute three core variables— key, value and query.

True. In Transformer, K, Q, V matrices are calculated in the multi-head self-attention layer.

Problem 1.2 (3 pts) In the self-attention layer of Transformer models, the attention is denoted by the cosine similarity between the key and query.

True. The attention is denoted as the dot product of key and query, followed by softmax function and weighted average. See the function: $s_{ij} = q_i^T k_j, a_{ij} = \frac{exp(s_{ij})}{\sum_{j'} exp(s_{ij'})}, Output_i = \sum_j a_{ij} v_j$. Similarly, cosine similarity is represented using a dot product and magnitude: $S_C(A, B) = \frac{A \cdot B}{\|A\| \|B\|}$. Therefore, we can say that the attention is denoted by the cosine similarity between the key and query.

Problem 1.3 (3 pts) 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).

False. The attention is followed by softmax function and weighted average: $s_{ij} = q_i^T k_j, a_{ij} = \frac{exp(s_{ij})}{\sum_{j'} exp(s_{ij'})}, Output_i = \sum_j a_{ij} v_j$. It is already normalized, so a further normalization layer is not required.

Problem 1.4 (3 pts) The encoder of Transformer learns auto-regressively.

False. The encoder of Transformer is autoencoding while GPT learns auto-regressively.

Problem 1.5 (3 pts) Both BERT's and GPT's architecture are Transformer decoder.

False. BERT is a transformer encoder while GPT is transformer decoder.

Problem 1.6 (3 pts) BERT's pre-training objectives include (a) masked token prediction (masked language modeling) and (b) next sentence prediction.

True. BERT's pretraining objective includes (1): predict the masked words, (2) predict whether two text sequence are contiguous.

Problem 1.7 (3 pts) Both GPT and BERT are zero-shot learner (can be transferred to unseen domain

True. After pre-training, both GPT and BERT learns general knowledge about language, and can be transferred on task-specific data with fine-tuning.

Problem 1.8 (3 pts) Gradient clipping can be used to alleviate gradient vanishing problem.

True. Gradient clipping is a method where the error derivative is clipped to a threshold during backward propagation through the network and using the clipped gradients to update the weights. By defining a minimum clip value and a maximum clip value, if a gradient exceeds some threshold value, we clip that gradient to the threshold.; if the gradient is less than the lower limit then we clip that too, to the lower limit of the threshold. It is often used to solve gradient explosion and gradient vanishing problem.

Problem 1.9 (3 pts) Word embeddings can contain positive or negative values.

True. Word embeddings can contain both positive and negative values. These values represent different dimensions or features of the word's meaning in the embedding space.

Problem 1.10 (3 pts) The memory cell of an LSTM is computed by a weighted average of previous memory state and current memory state where the sum of weights is 1.

False. The memory cell $C_t = f_t \circ C_{t-1} + i_t \circ \overline{C_t}$ where f_t is the forget gate, i_t is the input gate, C_{t-1} is the previous memory state and $\overline{C_t}$ is the candidate memory state. f_t is computed by forget gate: $f_t = \sigma(W_f[h_{t-1}, x_t] + b_f$, i_t is computed by input gate: $i_t = \sigma(W_i[h_{t-1}, x_t] + b_i$. They don't necessarily sum up to 1.

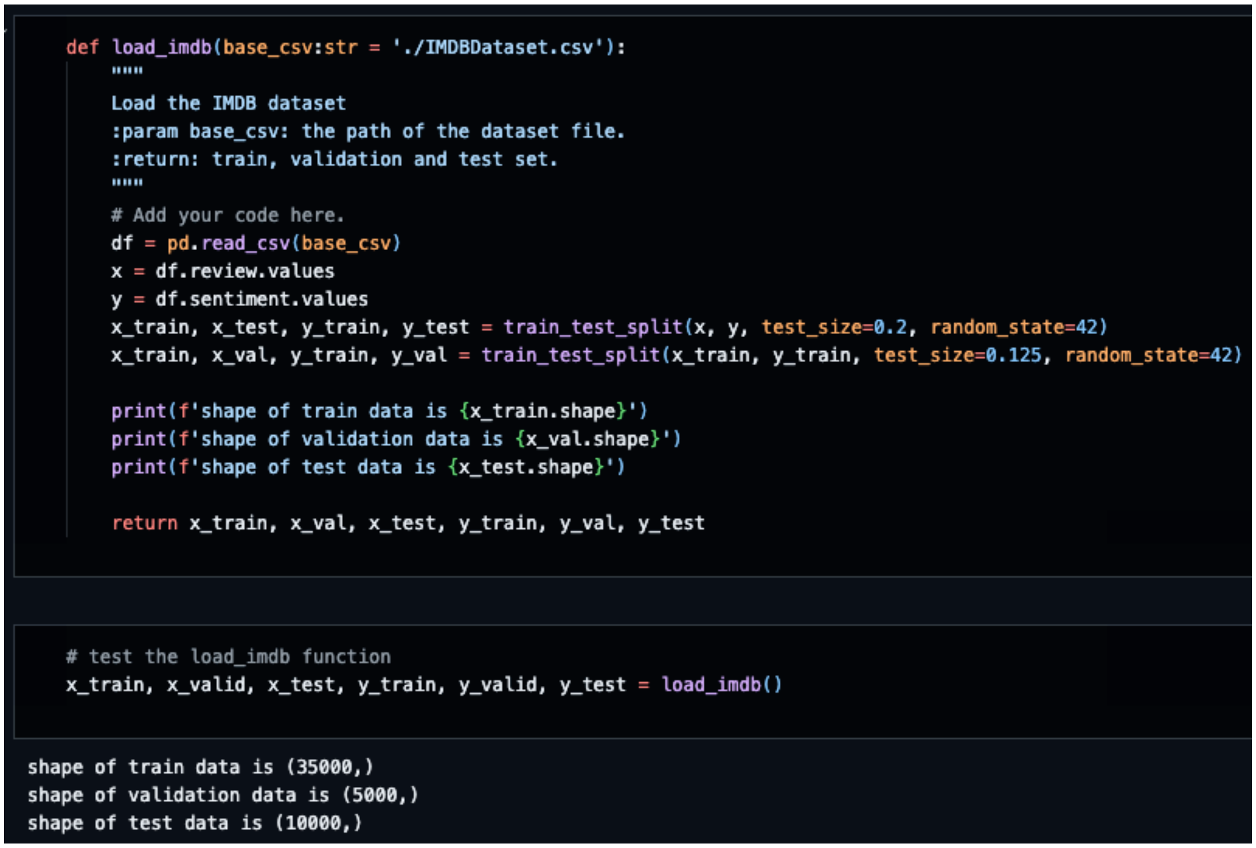
2 Lab 1: Recurrent Neural Network for Sentiment Analysis (45 pts)

In this lab, you will learn to implement an LSTM model for sentiment analysis. Sentiment analysis [1, 2] is a classification task to identify the sentiment of language. It usually classifies data into two to three labels: positive, negative or/and neutral. IMDB [3] is one of the most widely used dataset for binary sentiment classification. It uses only positive and negative labels. IMDB contains 50,000 movie review data collected from popular movie rating service IMDB. You may find more details at https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews.

To start this assignment, please open the provided Jupyter Notebook LabRNN.ipynb. We have pro- vided implementation for the training recipe, including optimizers, learning rate schedulers, and weight initialization. You may **NOT** change any of the hyperparameter settings (i.e., the object HyperParams) in this lab. **Grading Instructions.** For this part, we mainly grade on the logic and conciseness of the code. If a mistake is found in this part that lead to erroneous conclusions for questions in the later part (e.g., Lab 2), we will consider this and provide partial/full credit for the later part, to avoid applying the same penalty twice. Note that TA and Professors have the final discretion to adjust grade according to the given submission.

(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.

ANSWER:
I used *train_test_split* in scikit-learn to split the dataset. First I used *test_size=0.2* to separate the test dataset and train+validation dataset, then I used *test_size=0.125* to split training set and validation set. Finally I validate the function by testing on IMDB dataset and printing out the shape of each set, and the train, validation, and test is 7:1:2.



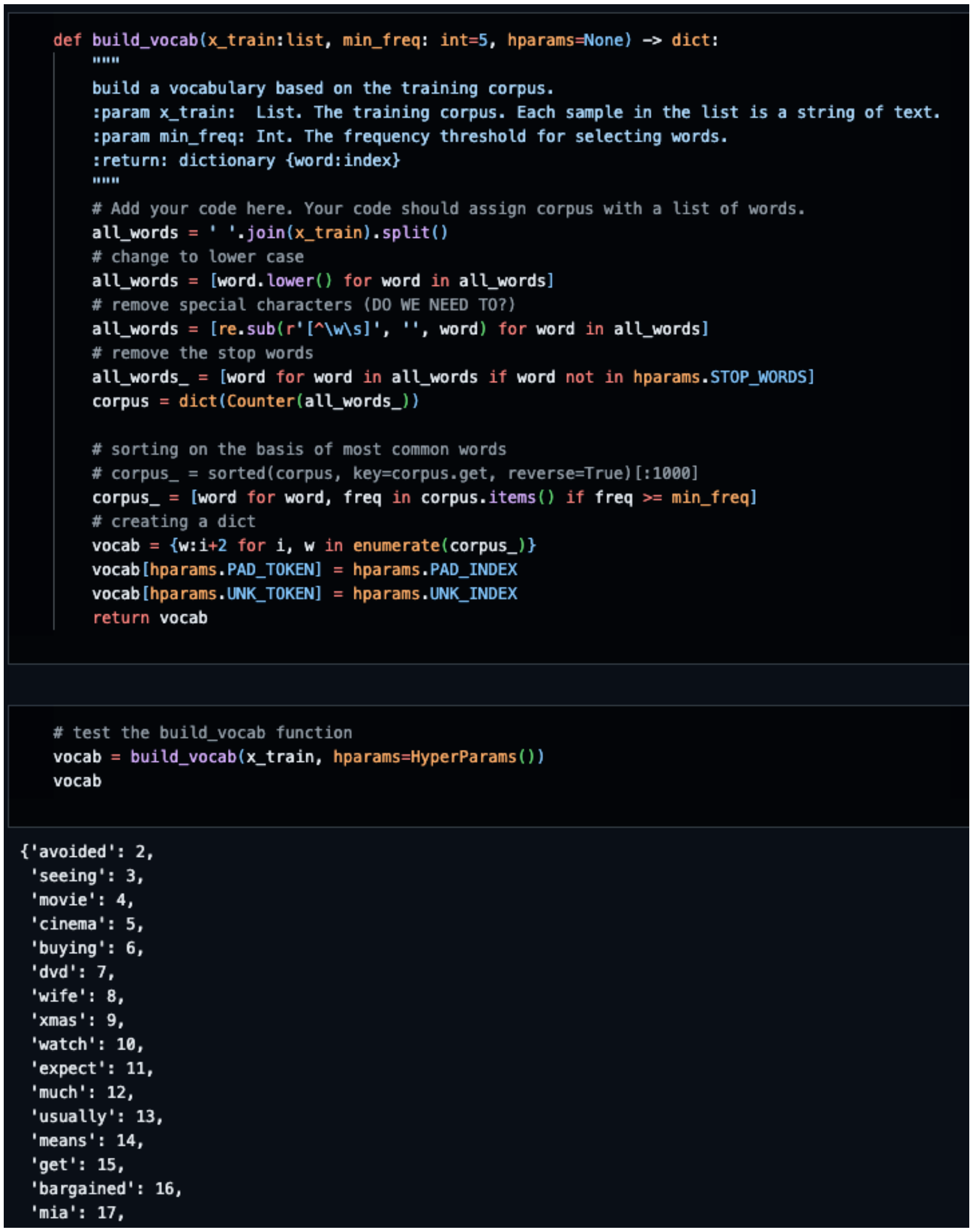
```
def load_imdb(base_csv:str = './IMDBDataset.csv'):
    """
    Load the IMDB dataset
    :param base_csv: the path of the dataset file.
    :return: train, validation and test set.
    """
    # Add your code here.
    df = pd.read_csv(base_csv)
    x = df.review.values
    y = df.sentiment.values
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
    x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.125, random_state=42)

    print(f'shape of train data is {x_train.shape}')
    print(f'shape of validation data is {x_val.shape}')
    print(f'shape of test data is {x_test.shape}')

    return x_train, x_val, x_test, y_train, y_val, y_test
```

```
# test the load_imdb function
x_train, x_valid, x_test, y_train, y_valid, y_test = load_imdb()
```

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

(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 (≥ min_freq) and finally generate a corpus variable that contains a list of words.

ANSWER:
In *build_vocab* function, I first split the input list, change to lower case, remove special characters other than words and digits, and remove the stop words. Then I used Counter function to get the frequency of each word and store them into a dictionary. Finally, I filtered the words that occur higher than minimum frequency threshold and generates a corpus variable dictionary with index.



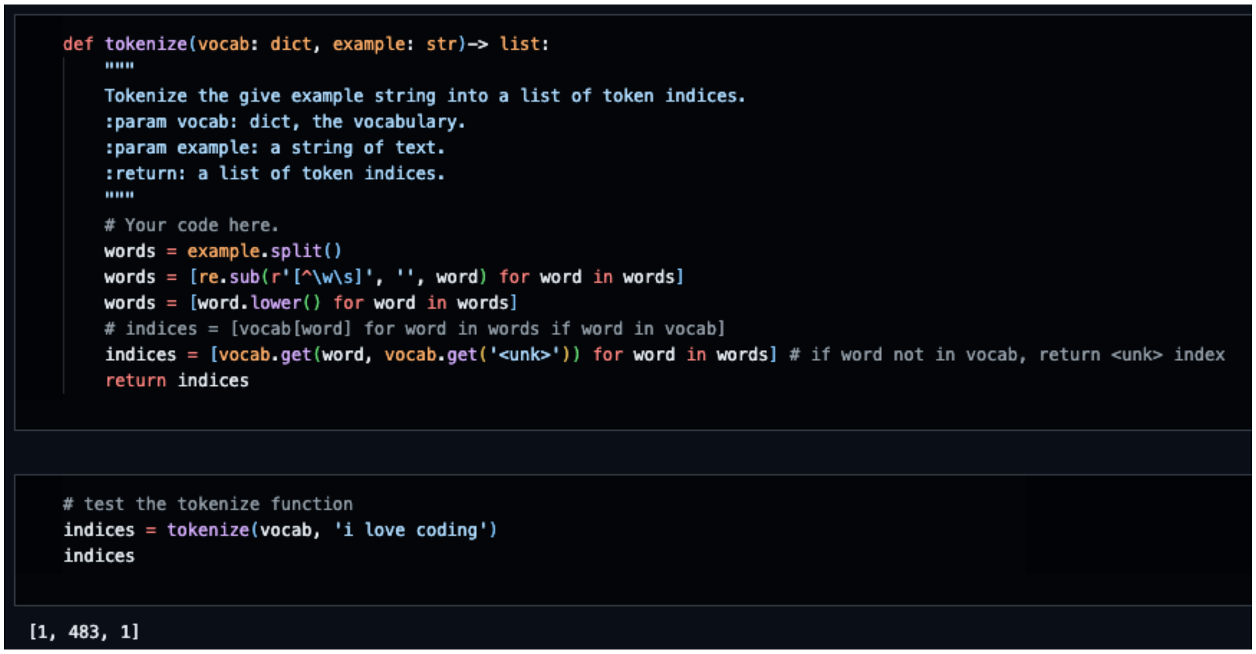
```
def build_vocab(x_train:list, min_freq: int=5, hparams=None) -> dict:
    """
    build a vocabulary based on the training corpus.
    :param x_train:  List. The training corpus. Each sample in the list is a string of text.
    :param min_freq: Int. The frequency threshold for selecting words.
    :return: dictionary {word:index}
    """
    # Add your code here. Your code should assign corpus with a list of words.
    all_words = ' '.join(x_train).split()
    # change to lower case
    all_words = [word.lower() for word in all_words]
    # remove special characters (DO WE NEED TO?)
    all_words = [re.sub(r'[^\w\s]', '', word) for word in all_words]
    # remove the stop words
    all_words_ = [word for word in all_words if word not in hparams.STOP_WORDS]
    corpus = dict(Counter(all_words_))

    # 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
```

```
# test the build_vocab function
vocab = build_vocab(x_train, hparams=HyperParams())
vocab
```

```
{'avoided': 2,
 'seeing': 3,
 'movie': 4,
 'cinema': 5,
 'buying': 6,
 'dvd': 7,
 'wife': 8,
 'xmas': 9,
 'watch': 10,
 'expect': 11,
 'much': 12,
 'usually': 13,
 'means': 14,
 'get': 15,
 'bargained': 16,
 'mia': 17,
```

(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.

ANSWER:
For the input vocab, I split the sentence into words, remove special characters, change to lower case, and use dictionary key to get the index. If the word is not in vocab, it will return the index of *<unk>.*



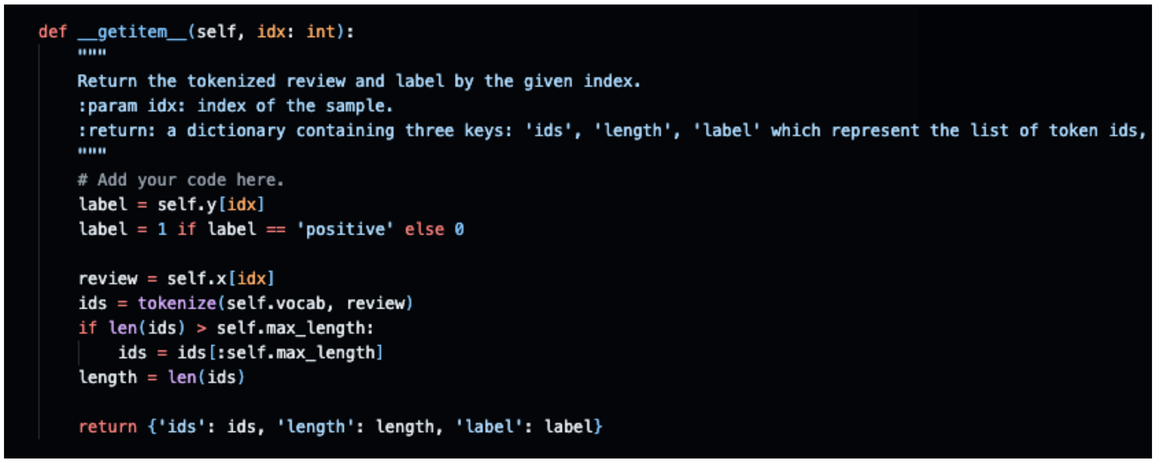
```
def tokenize(vocab: dict, example: str)-> list:
    """
    Tokenize the give example string into a list of token indices.
    :param vocab: dict, the vocabulary.
    :param example: a string of text.
    :return: a list of token indices.
    """
    # Your code here.
    words = example.split()
    words = [re.sub(r'[^\w\s]', '', word) for word in words]
    words = [word.lower() for word in words]
    # indices = [vocab[word] for word in words if word in vocab]
    indices = [vocab.get(word, vocab.get('<unk>')) for word in words] # if word not in vocab, return <unk> index
    return indices
```

```
# test the tokenize function
indices = tokenize(vocab, 'i love coding')
indices
```

```
[1, 483, 1]
```

(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 paramete r 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.

ANSWER:
In *__getitem__* function, I set label to 1 if positive else 0. I also pass the review to *tokenize* function to get indices, and truncate it by *max_length*.

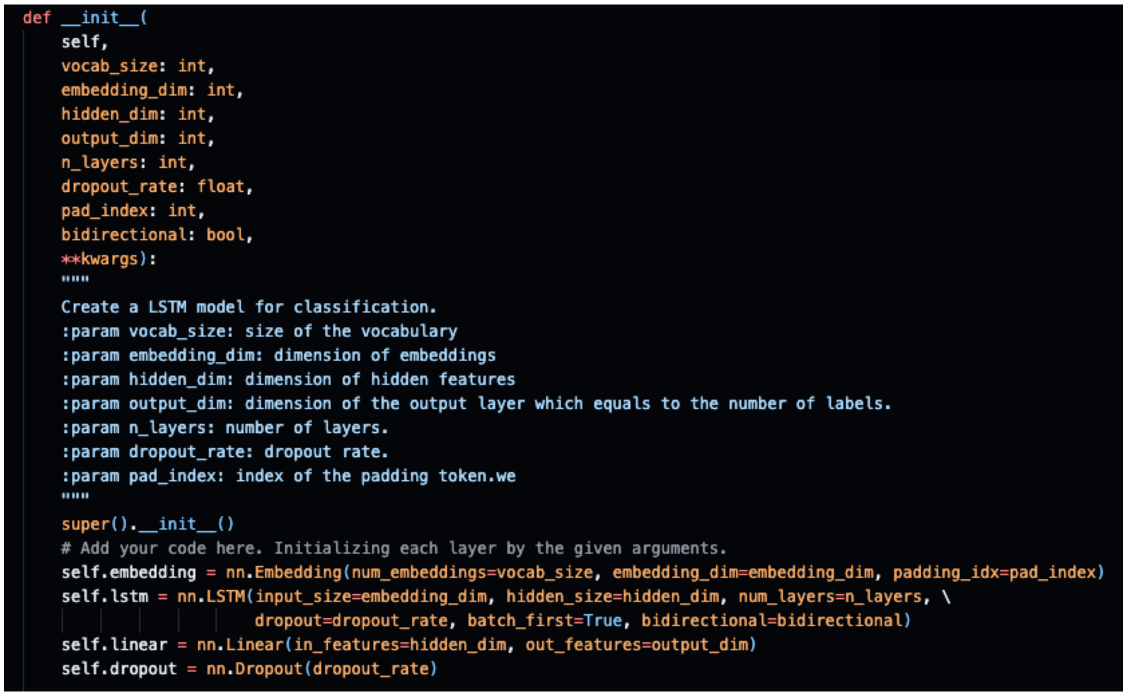


```
def __getitem__(self, idx: int):
    """
    Return the tokenized review and label by the given index.
    :param idx: index of the sample.
    :return: a dictionary containing three keys: 'ids', 'length', 'label' which represent the list of token ids,
    """
    # Add your code here.
    label = self.y[idx]
    label = 1 if label == 'positive' else 0

    review = self.x[idx]
    ids = tokenize(self.vocab, review)
    if len(ids) > self.max_length:
        ids = ids[:self.max_length]
    length = len(ids)

    return {'ids': ids, 'length': length, 'label': label}
```

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

(a) (5 pts) In __init__, a LSTM model contains 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.

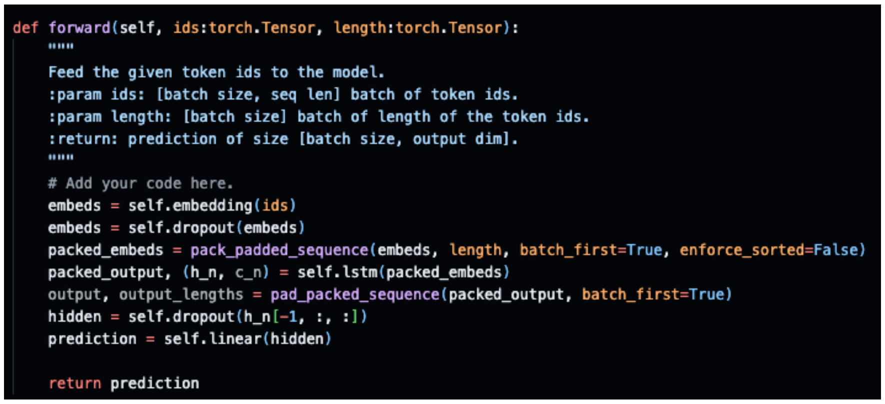
ANSWER:
In *__init__* function, I initialized 4 layers: *self.embedding, self.lstm, self.linear* and *self.dropout* using *nn.Embedding, nn.LSTM, nn.Linear, nn.Dropout*.



```
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.embedding = nn.Embedding(num_embeddings=vocab_size, embedding_dim=embedding_dim, padding_idx=pad_index)
    self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=hidden_dim, num_layers=n_layers, \
                        dropout=dropout_rate, batch_first=True, bidirectional=bidirectional)
    self.linear = nn.Linear(in_features=hidden_dim, out_features=output_dim)
    self.dropout = nn.Dropout(dropout_rate)
```

(b) (5 pts) In forward, 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].

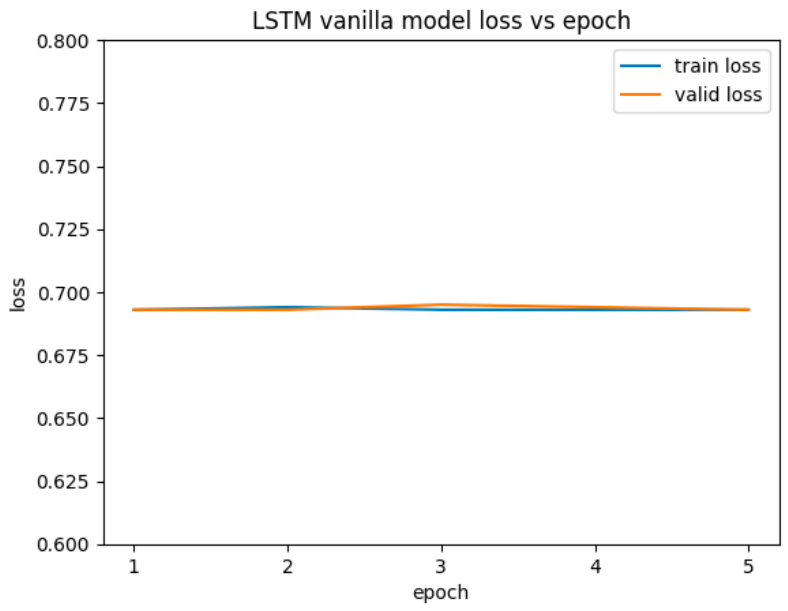
ANSWER:
In *forward* function, I first embed the indices, dropout, then pad the sequences into the same length using *pack_padded_sequence*, feed it to *lstm* layer, dropout the *lstm* output and finally feed the *hidden_n* to fully connected layer.



```
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.
    embeds = self.embedding(ids)
    embeds = self.dropout(embeds)
    packed_embeds = pack_padded_sequence(embeds, length, batch_first=True, enforce_sorted=False)
    packed_output, (h_n, c_n) = self.lstm(packed_embeds)
    output, output_lengths = pad_packed_sequence(packed_output, batch_first=True)
    hidden = self.dropout(h_n[-1, :, :])
    prediction = self.linear(hidden)

    return prediction
```

(f) (5 pts) As a sanity check, train the LSTM model for 5 epochs with **SGD optimizer**. Do you observe a steady and consistent decrease of the loss value as training progresses? Report your observation on the learning dynamics of training loss, and validation loss on the IMDB dataset. Do they meet your expectations and why? (**Hint**: trust what you have observed and report as is.)

ANSWER:



Training LSTM model for 5 epochs with SGD optimizer, the training loss and validation loss are always around 0.695. There is no observable decrease in the loss value. This meets my expectation because RNN is easy to have gradient vanishing/explosion problem, causing no effective learning in the training process. And SGD optimizer is not always the good choice in RNN models.

(g) (10 pts) Gated Recurrent Unit (GRU) is a simplified version of LSTM that performs good on language tasks. Implement the GRU model similar instructions as (e), as follows:

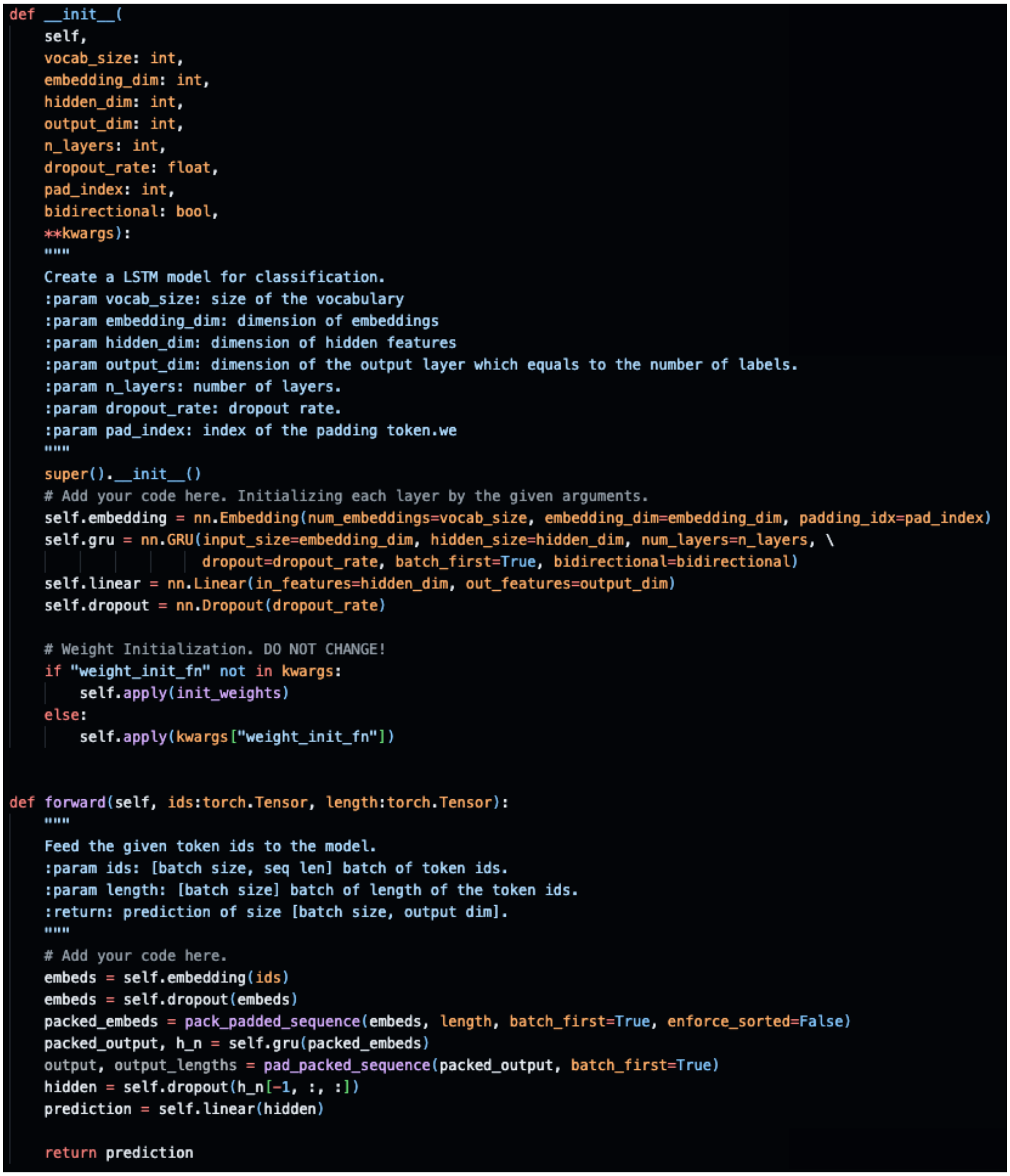
(a) (5 pts) In __init__, a GRU model includes an embedding layer, an gated recurrent unit, a linear layer, and a dropout layer. You can call functions from torch.nn library. For example, nn.Embedding, nn.GRU, nn.Linear.

ANSWER:
In *__init__* function, I initialized 4 layers: *self.embedding, self.gru, self.linear* and *self.dropout* using *nn.Embedding, nn.GRU, nn.Linear, nn.Dropout*.

(b) (5 pts) In forward, 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 GRU layer. Return the output features which is of size [batch size, output dim].

ANSWER:
In *forward* function, I first embed the indices, dropout, then pad the sequences into the same length

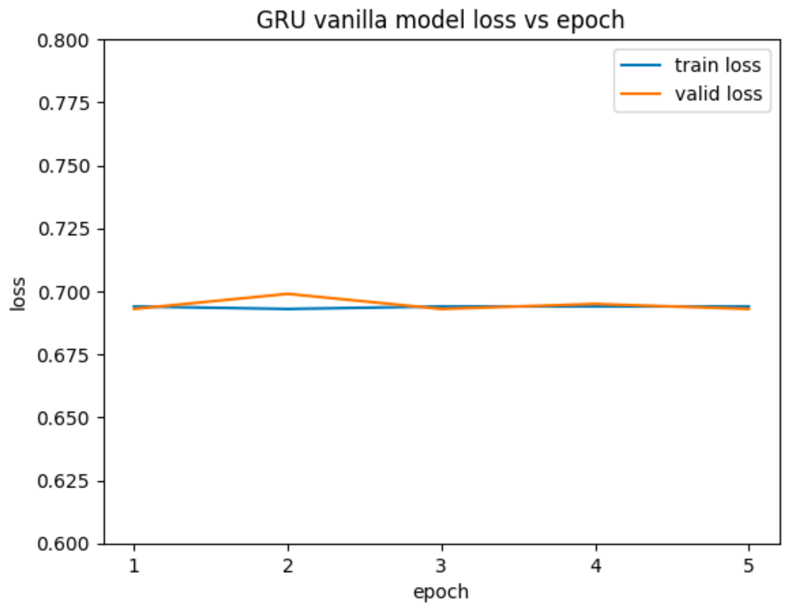
using *pack_padded_sequence*, feed it to *gru* layer, dropout the *lstm* output and finally feed the *hidden_n* to fully connected layer.



```
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.embedding = nn.Embedding(num_embeddings=vocab_size, embedding_dim=embedding_dim, padding_idx=pad_index)
    self.gru = nn.GRU(input_size=embedding_dim, hidden_size=hidden_dim, num_layers=n_layers, \
                      dropout=dropout_rate, batch_first=True, bidirectional=bidirectional)
    self.linear = nn.Linear(in_features=hidden_dim, out_features=output_dim)
    self.dropout = nn.Dropout(dropout_rate)

    # Weight Initialization. DO NOT CHANGE!
    if "weight_init_fn" not in kwargs:
        self.apply(init_weights)
    else:
        self.apply(kwargs["weight_init_fn"])


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.
    embeds = self.embedding(ids)
    embeds = self.dropout(embeds)
    packed_embeds = pack_padded_sequence(embeds, length, batch_first=True, enforce_sorted=False)
    packed_output, h_n = self.gru(packed_embeds)
    output, output_lengths = pad_packed_sequence(packed_output, batch_first=True)
    hidden = self.dropout(h_n[-1, :, :])
    prediction = self.linear(hidden)

    return prediction
```

(h) (**Bonus**, 5 pts) Repeat (f) for GRU model and report your observations on the training curve. What do you hypothesize is the issue that results in the current training curve? (**Hint:** Check slides of Lecture 9 for the recommendations on training RNNs.)

ANSWER:



The training process of GRU model with SGD optimizer is the similar to LSTM model. Train loss and validation loss stuck around 0.695. This is probably due to gradient vanishing/explosion problem when using SGD optimizer. According to Lecture9 Page30, RMSprop optimizer can be used instead to address the problem.

Please do **NOT** run more than 5 epochs as several epochs suffices to achieve the expected results. You are required to carry the completed part to Lab 2.

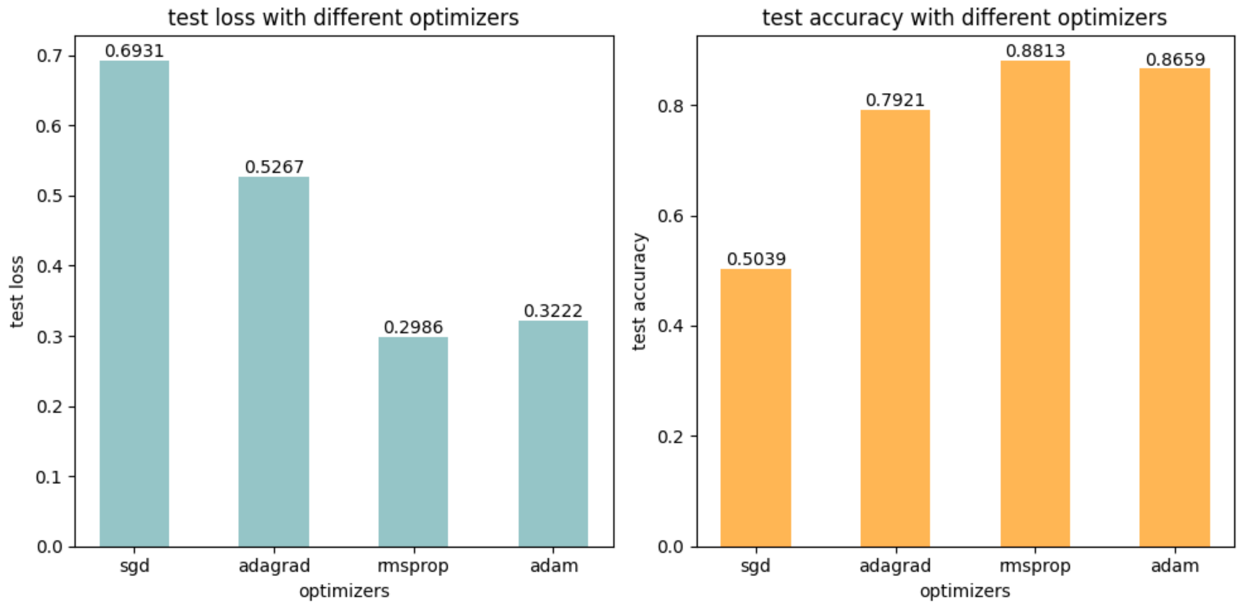
3 Lab 2: Training and Improving Recurrent Neural Network(25 pts)

In this lab, you will explore the training of RNN models on IMDB dataset, and propose some design improvements over the existing implementation. you should continue to work on LabRNN.ipynb with your implementation of the IMDB dataloaders, transformations, RNNs (LSTM and GRU). You will start playing with the hyperparameter settings to improve the training on IMDB sentiment analysis tasks, and study better design choices that crafts a better recurrent neural network.

For question (a) and (b), you may follow the hint to properly set the learning rate for your optimiz- ers. Despite that, you may **NOT** change any hyperparameters defined in LabRNN.ipynb, including the HyperParams object, RNN model configurations, and optimizer configurations.

(a) (5 pts) We start to look at the optimizers in training RNN models. As SGD might not be good enough, we switch to advanced optimizers (i.e., Adagrad, RMSprop, and Adam) with adaptive learning rate schedule. We start with exploring these optimizers on training a LSTM. You are asked to employ each of the optimizer on LSTM and report your observations. Which optimizer gives the best training results, and why?

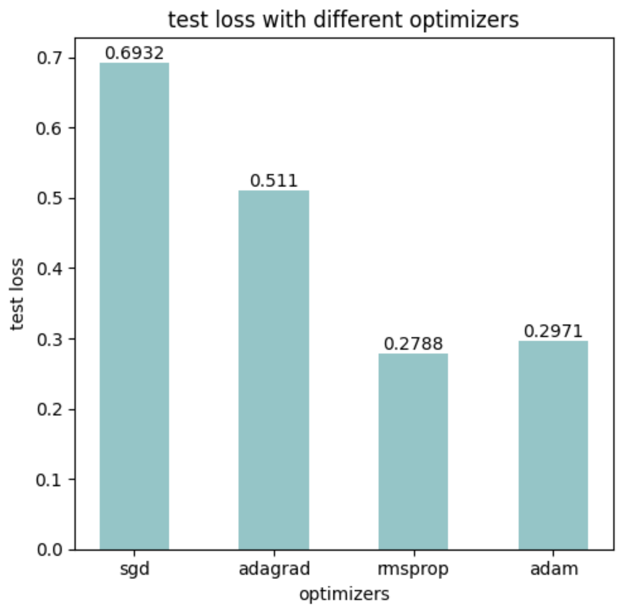
ANSWER:

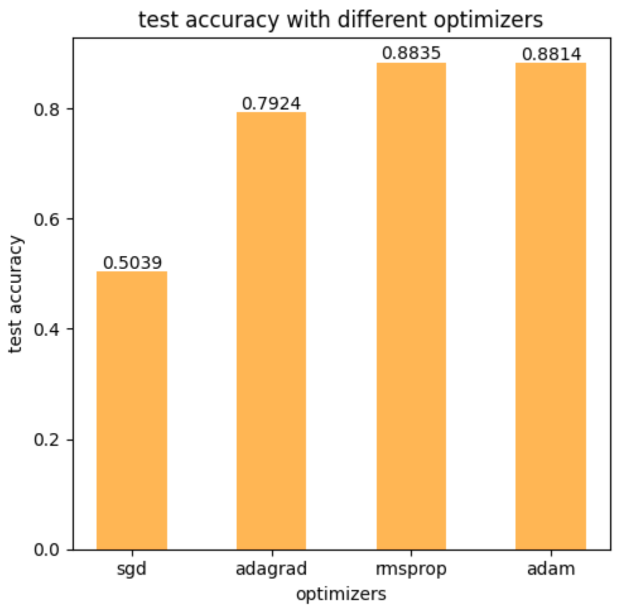


Comparing four optimizers while keeping other hyperparameters as default, RMSprop has the highest test accuracy of 0.8813, followed by adam, adagrad and sgd. This is because RMSprop uses a moving average of incoming gradients as the denominator. It helps prevent learning rate shrinkage and find optimal minima in the loss landscape. Adam is also an adaptive learning rate method, which achieved good results in the training process. Adagrad, another adaptive learning rate method, performs worse than the first two due to its aggressive decrease in learning rate, which can slow down learning in the later stages of training. On the other hand, SGD performs the worst among the four, suggesting it's not a good choice in RNN models.

(b) (5pts) Repeat (a) on training your GRU model, and provide ananalysis on the performance of different optimizers on a GRU model. How does a GRU compare with a LSTM, in terms of model performance and model efficiency?

ANSWER:



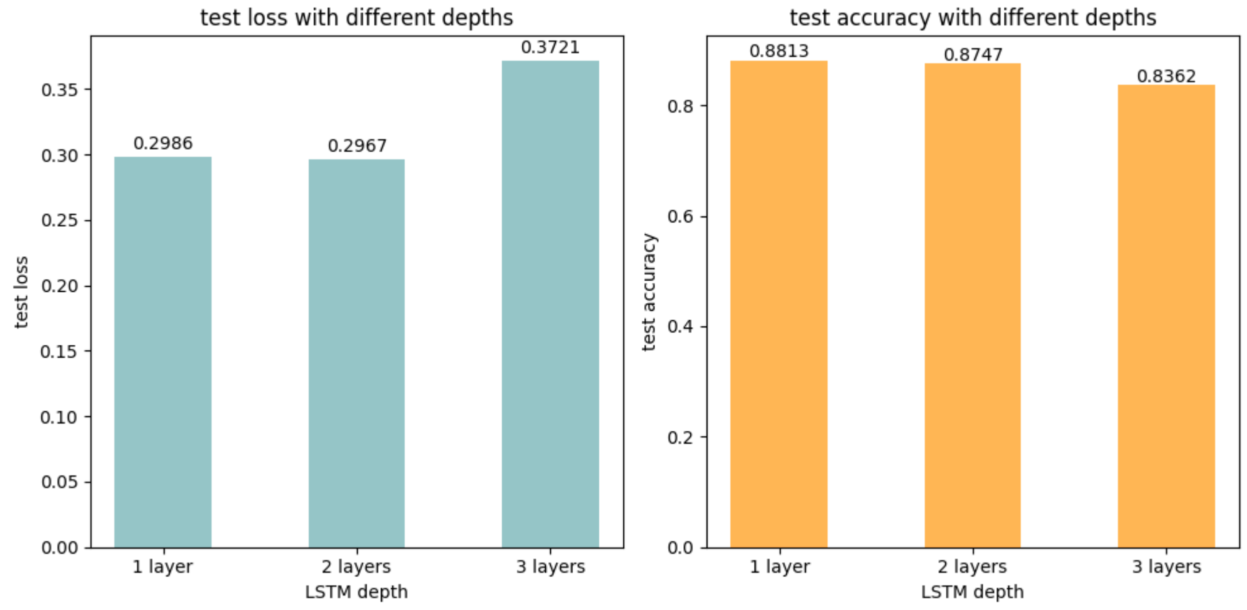


When training GRU model with different optimizers while keeping other hyperparameters as default, the trend is the same: RMSprop has the highest test accuracy, followed by adam, adagrad and sdg. However, the accuracy of GRU model for each optimizer is slightly higher than its corresponding LSTM model, indicating GRU has slightly better performance than LSTM. In terms of number of parameters, LSTM has 79398 parameters while GRU has 69098, which indicates that GRU is more lightweight than LSTM.

Up to this stage, you should expect **at least 84%** accuracy on IMDB dataset. You may find that optimizers solve most of the existing dilemmas in RNN training. Next, we will start playing with the structures of RNN models and see if we can design a better RNN model. Starting here, you may change any of the hyperparameters **except random seed**. Yet, our recommendation of hyperparameter change is limited to learning rate, number of layers in RNN models, and the dimension of embedding/hidden units.

(c) (5 pts) Try to make your RNN model deeper by changing the number of layers. Is your RNN model achieving a better accuracy on IDMB classification? You may use LSTM as an example. (**Hint:** you do not need to explore more than 4 recurrent layers in a RNN model.).

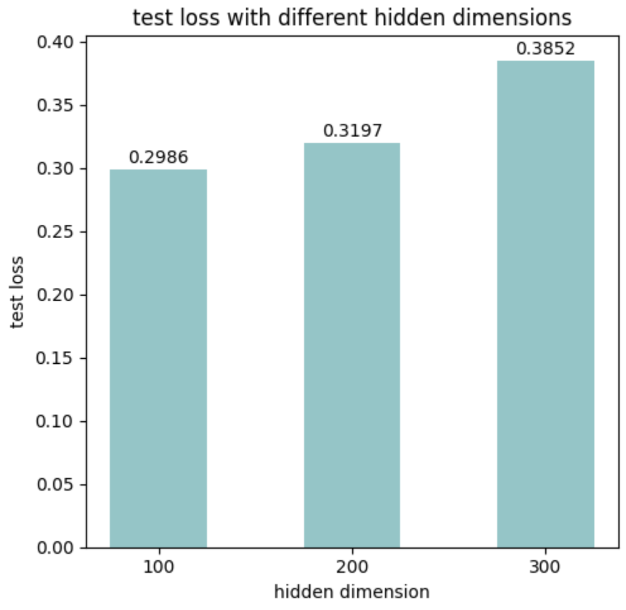
ANSWER:

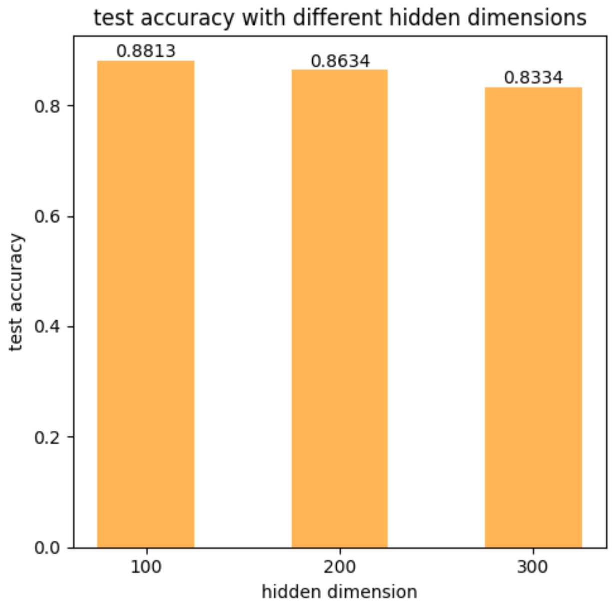


Increasing from 1 LSTM layer to 3 LSTM layers while keeping other hyperparameters as default, the test accuracy keeps decreasing. While deeper networks have a greater capacity to learn intricate patterns, they also come with challenges that might affect performance, such as overfitting, over-complex loss landscapes with local minima, saddle points and plateaus, insufficient training data, etc. Therefore, a deeper network is not always the best choice.

(d) (5 pts) Try to make your RNN model wider by changing the number of hidden units. Is your RNN model achieving a better accuracy on IDMB classification? You may use LSTM as an example. (**Hint:** you do not need to explore a hidden dimension of more than 320 on IMDB).

ANSWER:

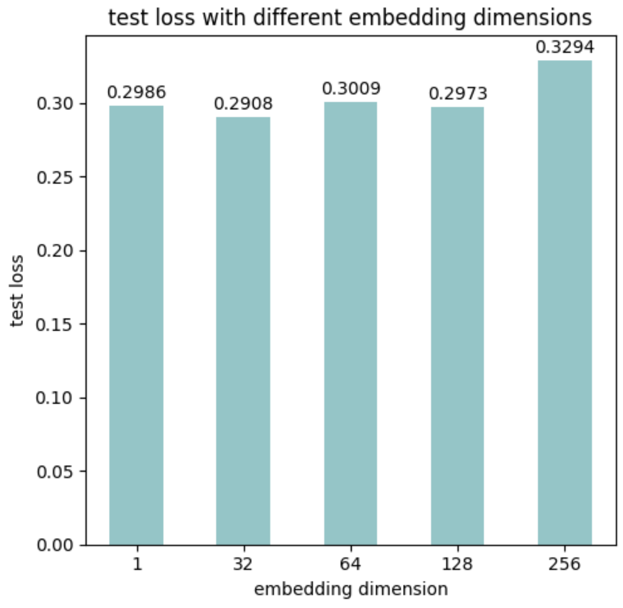


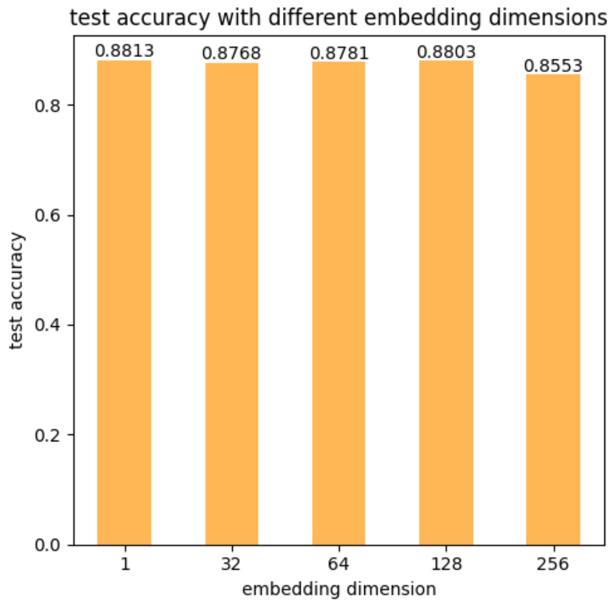


In LSTM model, increasing hidden dimensions from 100 to 300 while keeping other hyperparameters as default led to decreased test accuracy from 0.8813 to 0.8334. Similar to the last question, increasing model complexity can bring challenges like overfitting, over-complex loss, insufficient training data, etc.

(e) (5 pts) Embedding tables contain rich information of the input words and help build a more powerful representation with word vectors. Try to increase the dimension of embeddings. Is your RNN model achieving a better accuracy on IMDB classification? You may use LSTM as an example. (**Hint:** you do not need to explore an embedding dimension of larger than 256).

ANSWER:

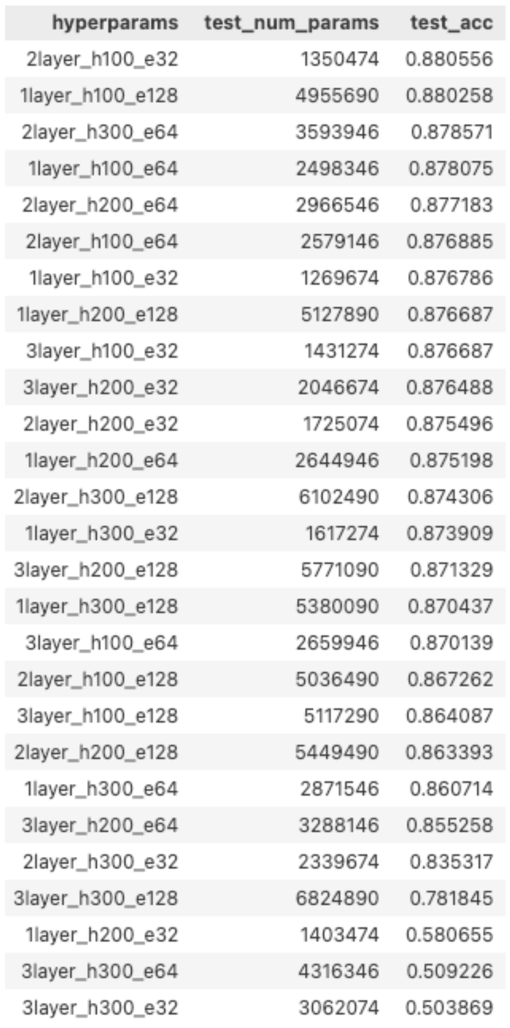




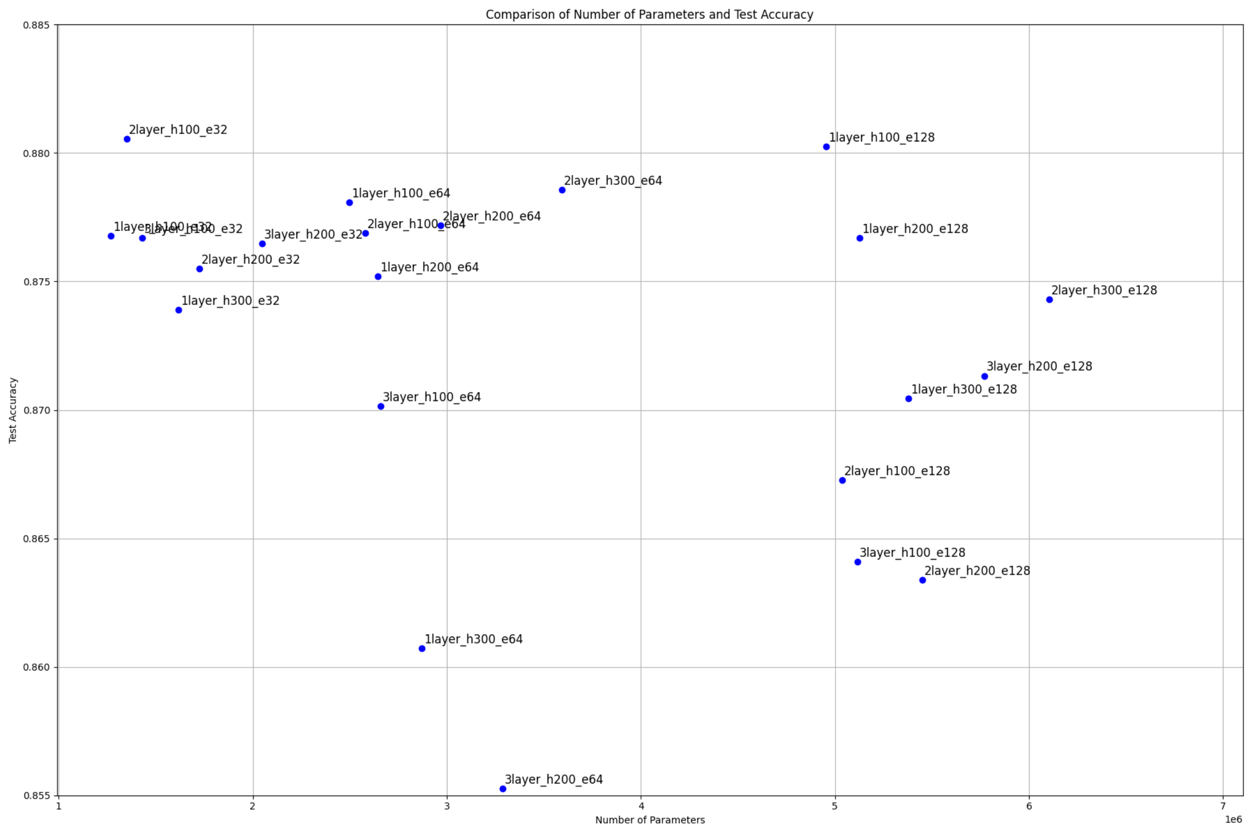
In the LSTM model, the test accuracies for embedding dimensions of 1, 32, 64, and 128 are 0.8813, 0.8768, 0.8781, and 0.8803, respectively while keeping other hyperparameters as default. Model performances are relatively consistent across these dimensions. However, when the embedding dimension is increased to 256, there's a noticeable drop in performance, with the accuracy falling to 0.8553.

(f) (**Bonus**, 5 pts) A better way to scale up RNN models is simultaneously scaling up the number of hidden units, number of layers, and embedding dimension. This is called **compound scaling**, which is widely adopted in emerging ML research. You are asked to make **no more than 50** trials to perform a compound scaling on your implemented RNN models. Is the model crafted via compound scaling performing better than the models you obtain in (d), (e), and (f)? You may use LSTM as an example. (**Hint:** You may use the accuracy-parameter trade-off as a criterion.)

ANSWER:



hyperparams	test_num_params	test_acc
2layer_h100_e32	1350474	0.880556
1layer_h100_e128	4955690	0.880258
2layer_h300_e64	3593946	0.878571
1layer_h100_e64	2498346	0.878075
2layer_h200_e64	2966546	0.877183
2layer_h100_e64	2579146	0.876885
1layer_h100_e32	1269674	0.876786
1layer_h200_e128	5127890	0.876687
3layer_h100_e32	1431274	0.876687
3layer_h200_e32	2046674	0.876488
2layer_h200_e32	1725074	0.875496
1layer_h200_e64	2644946	0.875198
2layer_h300_e128	6102490	0.874306
1layer_h300_e32	1617274	0.873909
3layer_h200_e128	5771090	0.871329
1layer_h300_e128	5380090	0.870437
3layer_h100_e64	2659946	0.870139
2layer_h100_e128	5036490	0.867262
3layer_h100_e128	5117290	0.864087
2layer_h200_e128	5449490	0.863393
1layer_h300_e64	2871546	0.860714
3layer_h200_e64	3288146	0.855258
2layer_h300_e32	2339674	0.835317
3layer_h300_e128	6824890	0.781845
1layer_h200_e32	1403474	0.580655
3layer_h300_e64	4316346	0.509226
3layer_h300_e32	3062074	0.503869



I applied compound scaling to simultaneously scale up the number of layers of 1, 2, 3, the number of hidden units of 100, 200, 300, and embedding dimension of 32, 64, 128, which is 27 trials in total. After comparing the trade-off between accuracy and model size, 2 layers with 100 hidden units and 32 embedding dimension performs the best. It reaches the highest test accuracy of 88.06%, and the number of parameters is the second smallest. It is not better than the best model achieved in (d)(e)(f), since the 1 layer with 100 hidden units and 1 embedding dimension achieves 0.07% better accuracy in previous questions. This is the hyperparameter combination that is not covered in compound scaling.

(g) (**Bonus**, 5 pts) RNNs can be bidirectional. Use the best model discovered in (f) and make it bidirectional. Do you observe better accuracy on IMDB dataset and why?

ANSWER:

	LSTM	Bidirectional LSTM
Number of parameters	1,350,474	1,564,874
Test accuracy	0.881	0.877

When using the best model discovered in the last question, which is 2 layer, 100 hidden dimensions, 32 embeddings, the bidirectional model doesn't perform better than the vanilla LSTM. The accuracy is lower (0.877 vs 0.881) but the number of parameters is larger. This might be due to the fact that bidirectional LSTM is more complicated and the increased model complexity can lead to overfitting, especially when the dataset is not large enough or the model is not regularized properly.

Please do **NOT** run more than 5 epochs as several epochs suffices to achieve the expected results. Com- pleting (a) ~ (e) and (g) should give an accuracy of 84~86%, and completing (f) should give an accuracy of 86~88%. Yet, you are only required to achieve **84%** to successfully complete Lab 2. You are required to submit the completed version of LabRNN.ipynb for Lab 2.

Info: Additional requirements:

- **DO NOT** copy code directly online or from other classmates. We will check it! The result can be severe if your codes fail to pass our check. As this assignment requires much computing power of GPUs, we suggest:
- Plan your work in advance and start early. We will **NOT** extend the deadline because of the unavailability of computing resources.
- Be considerate and kill Jupyter Notebook instances when you do not need them.
- **DO NOT** run your program forever. Please follow the recommended/maximum training budget in each lab.
- Please do not train any models for more than 5 epochs, especially when you are doing (f) and (g) in Lab 2.

References

[1] P.D.Turney,"Thumbs up or thumbs down? Semantic orientation app lied to unsupervised classification of reviews," *arXiv preprint cs/0212032*, 2002.

[2] B. Pang and L. Lee, "A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts," *arXiv preprint cs/0409058*, 2004.

[3] S. Dooms, T. De Pessemier, and L. Martens, "Movietweetings: a movie rating dataset collected from twitter," in *Workshop on Crowdsourcing and human computation for recommender systems, CrowdRec at RecSys*, vol. 2013, p. 43, 2013.