ECE 661: Homework #3

Understand and Implement Sequence Models

NetID: wh162

1. True/False Questions:
    a. Problem 1.1: True, as we know from the structure of a Transformer, the basic block is self-attention and feed-forward. Whereas self-attention requires queries, values, and keys.

    b. Problem 1.2: False, in self-attention, the key, value, and queries all came from the same source, where all three of them are linear transformation used for all x_i

    c. Problem 1.3: False, not only after the attention matrix, but we also have to done executing the feed-forward module before we can apply any normalization.

    d. Problem 1.4: False, from Lec 10 P26, we know that GPT is autoregressive while BRET is autoencoding.

    e. Problem 1.5: False, both has encoder and decoder.

    f. Problem 1.6: True, refer to Lec 10 P 27

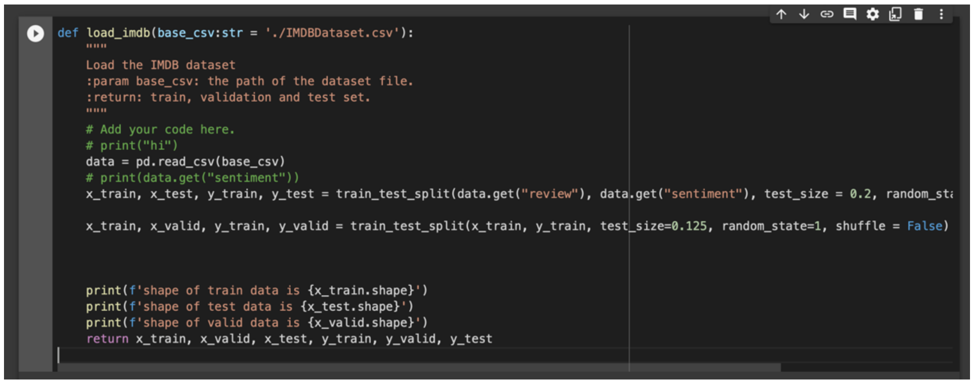
    g. Problem 1.7: False, both GPT and BERT are zero-shot learner

    h. Problem 1.8: True, gradient clipping helps gradient descent to have a reasonable behavior even is the loss landscape in the model is irregular.

i. Problem 1.9: False, word embedding can contain both positive and negative values.

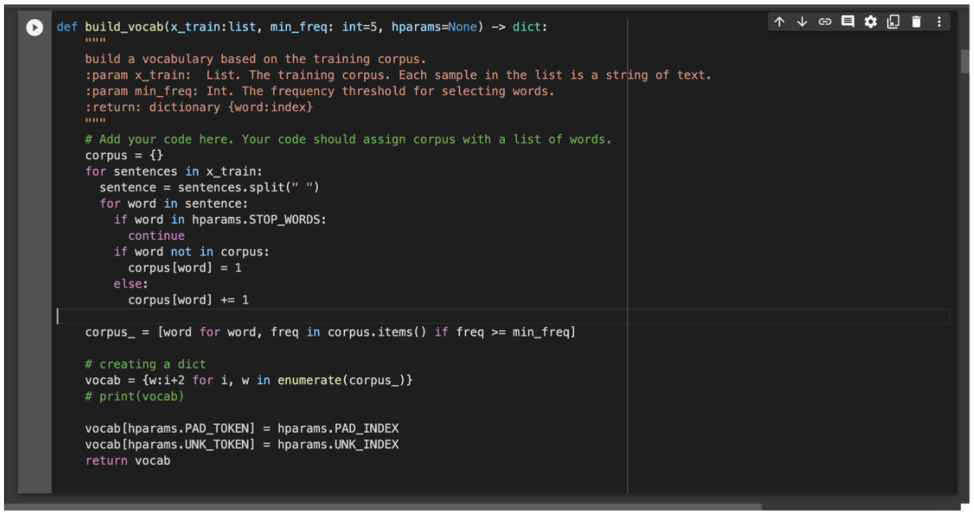
j. Problem 1.10: False, the sum of the weight is not guaranteed to be 1

2. Lab 1: Recurrent Neural Network for sentiment analysis
   a. Problem 2.1:



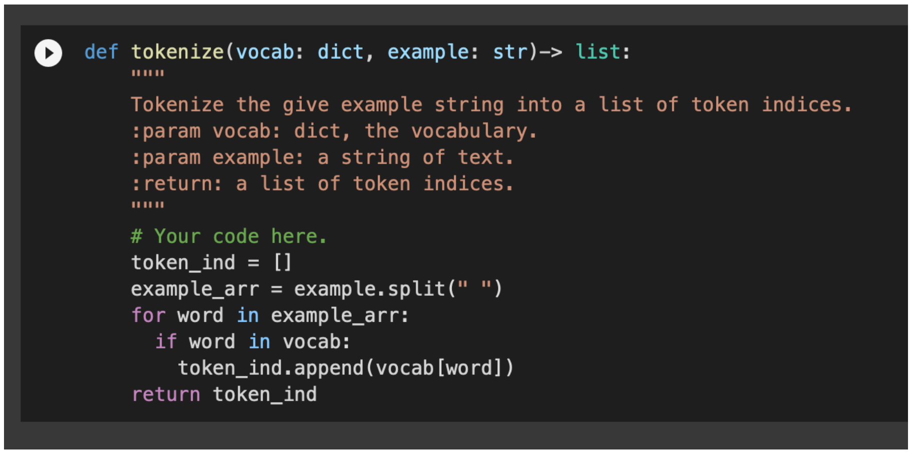
```python
def load_imdb(base_csv:str = './IMDBDataset.csv'):
    """
    Load the IMDB dataset
    :param base_csv: the path of the dataset file.
    :return: train, validation and test set.
    """
    # Add your code here.
    # print("hi")
    data = pd.read_csv(base_csv)
    # print(data.get("sentiment"))
    x_train, x_test, y_train, y_test = train_test_split(data.get("review"), data.get("sentiment"), test_size = 0.2, random_sta

    x_train, x_valid, y_train, y_valid = train_test_split(x_train, y_train, test_size=0.125, random_state=1, shuffle = False)



    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
```

   b. Problem 2.2:



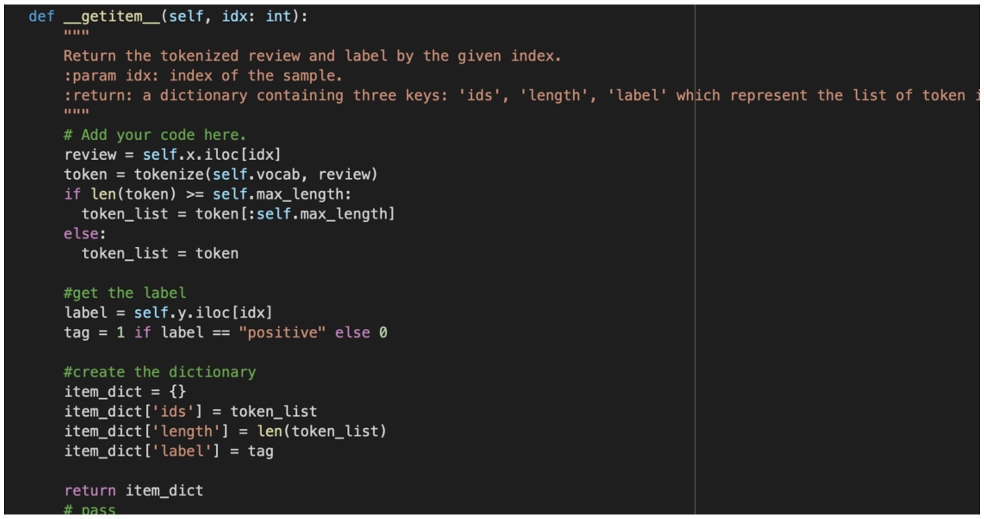
```python
def build_vocab(x_train:list, min_freq: int=5, hparams=None) -> dict:
    """
    build a vocabulary based on the training corpus.
    :param x_train:  List. The training corpus. Each sample in the list is a string of text.
    :param min_freq: Int. The frequency threshold for selecting words.
    :return: dictionary {word:index}
    """
    # Add your code here. Your code should assign corpus with a list of words.
    corpus = {}
    for sentences in x_train:
      sentence = sentences.split(" ")
      for word in sentence:
        if word in hparams.STOP_WORDS:
          continue
        if word not in corpus:
          corpus[word] = 1
        else:
          corpus[word] += 1

    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_)}
    # print(vocab)

    vocab[hparams.PAD_TOKEN] = hparams.PAD_INDEX
    vocab[hparams.UNK_TOKEN] = hparams.UNK_INDEX
    return vocab
```

c. Problem 2.3:



```
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.
    token_ind = []
    example_arr = example.split(" ")
    for word in example_arr:
      if word in vocab:
        token_ind.append(vocab[word])
    return token_ind
```

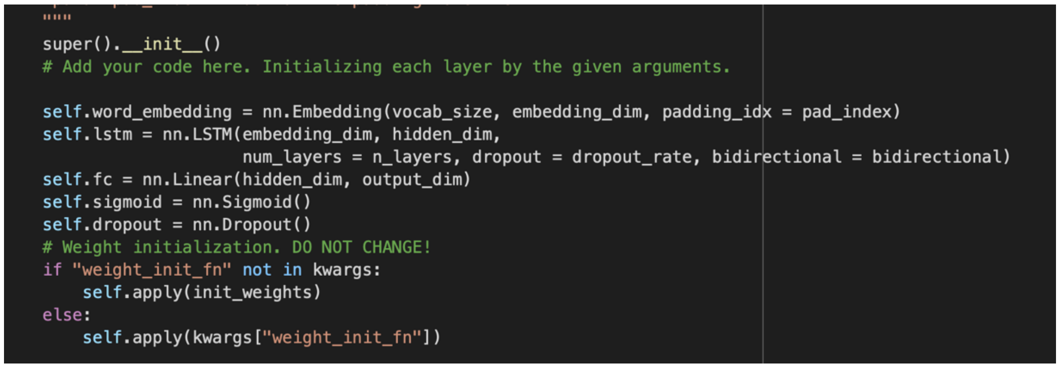
d. Problem 2.4:



```
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 i
    """
    # Add your code here.
    review = self.x.iloc[idx]
    token = tokenize(self.vocab, review)
    if len(token) >= self.max_length:
      token_list = token[:self.max_length]
    else:
      token_list = token

    #get the label
    label = self.y.iloc[idx]
    tag = 1 if label == "positive" else 0

    #create the dictionary
    item_dict = {}
    item_dict['ids'] = token_list
    item_dict['length'] = len(token_list)
    item_dict['label'] = tag

    return item_dict
    # pass
```

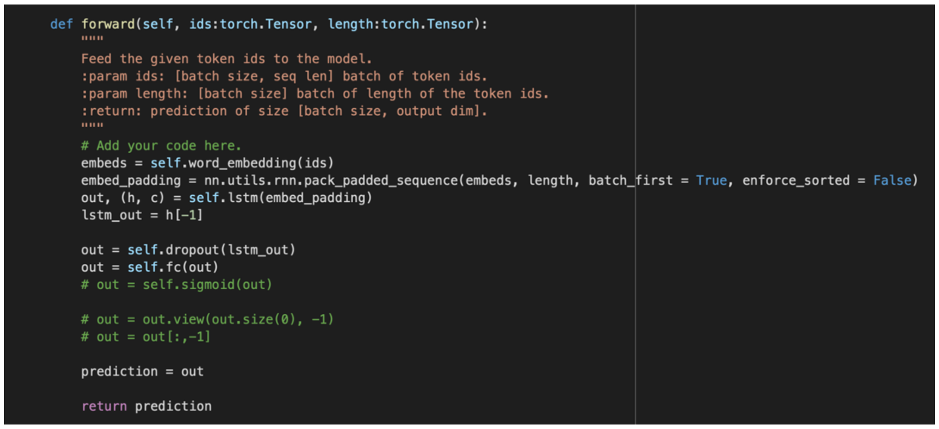
e. Problem 2.5:

i. 2.5.1:



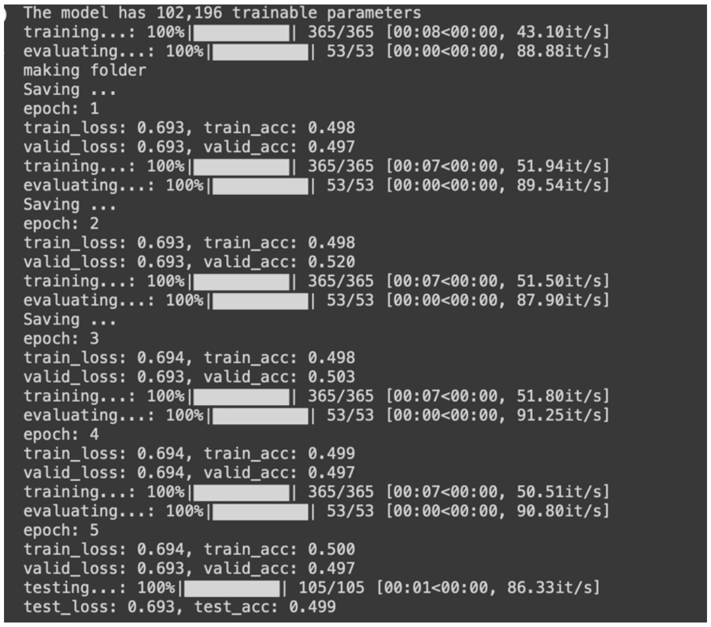
```
        """
        super().__init__()
        # Add your code here. Initializing each layer by the given arguments.

        self.word_embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx = pad_index)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim,
                            num_layers = n_layers, dropout = dropout_rate, bidirectional = bidirectional)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.sigmoid = nn.Sigmoid()
        self.dropout = nn.Dropout()
        # Weight initialization. DO NOT CHANGE!
        if "weight_init_fn" not in kwargs:
            self.apply(init_weights)
        else:
            self.apply(kwargs["weight_init_fn"])
```

ii. 2.5.2:



```
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.word_embedding(ids)
    embed_padding = nn.utils.rnn.pack_padded_sequence(embeds, length, batch_first = True, enforce_sorted = False)
    out, (h, c) = self.lstm(embed_padding)
    lstm_out = h[-1]

    out = self.dropout(lstm_out)
    out = self.fc(out)
    # out = self.sigmoid(out)

    # out = out.view(out.size(0), -1)
    # out = out[:,-1]

    prediction = out

    return prediction
```

f. Problem 2.6:

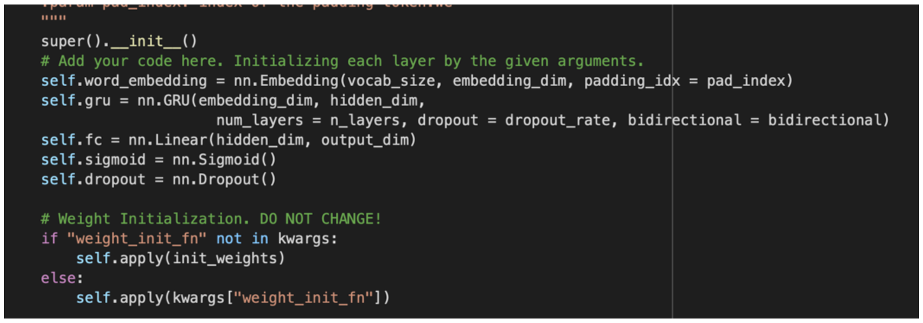


```
The model has 102,196 trainable parameters
training...: 100%|██████████| 365/365 [00:08<00:00, 43.10it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 88.88it/s]
making folder
Saving ...
epoch: 1
train_loss: 0.693, train_acc: 0.498
valid_loss: 0.693, valid_acc: 0.497
training...: 100%|██████████| 365/365 [00:07<00:00, 51.94it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 89.54it/s]
Saving ...
epoch: 2
train_loss: 0.693, train_acc: 0.498
valid_loss: 0.693, valid_acc: 0.520
training...: 100%|██████████| 365/365 [00:07<00:00, 51.50it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 87.90it/s]
Saving ...
epoch: 3
train_loss: 0.694, train_acc: 0.498
valid_loss: 0.693, valid_acc: 0.503
training...: 100%|██████████| 365/365 [00:07<00:00, 51.80it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 91.25it/s]
epoch: 4
train_loss: 0.694, train_acc: 0.499
valid_loss: 0.694, valid_acc: 0.497
training...: 100%|██████████| 365/365 [00:07<00:00, 50.51it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 90.80it/s]
epoch: 5
train_loss: 0.694, train_acc: 0.500
valid_loss: 0.693, valid_acc: 0.497
testing...: 100%|██████████| 105/105 [00:01<00:00, 86.33it/s]
test_loss: 0.693, test_acc: 0.499
```

No, the result doesn't meet my expectation. I thought the training accuracy and validation accuracy would increase as the epoch goes on. However, the accuracy stayed the same even after 5 epochs. My guess is that SGD optimizer got itself into a local min that result in this way.

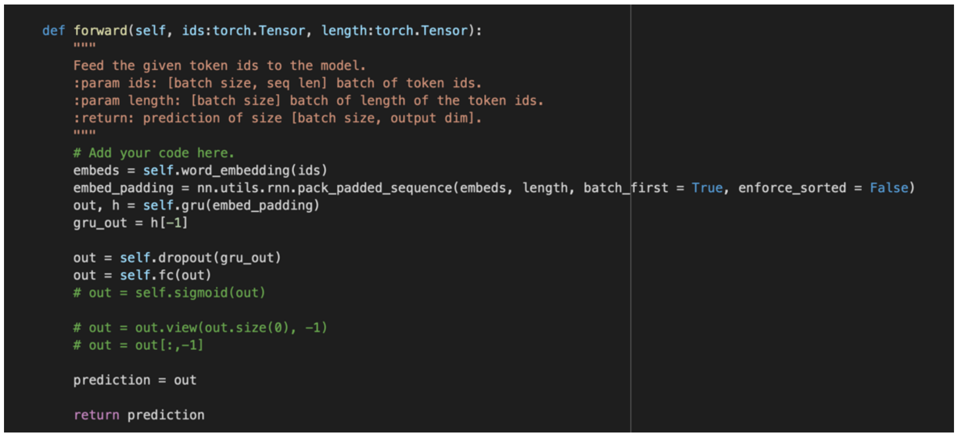
g. Problem 2.7:

i. 2.7.1:



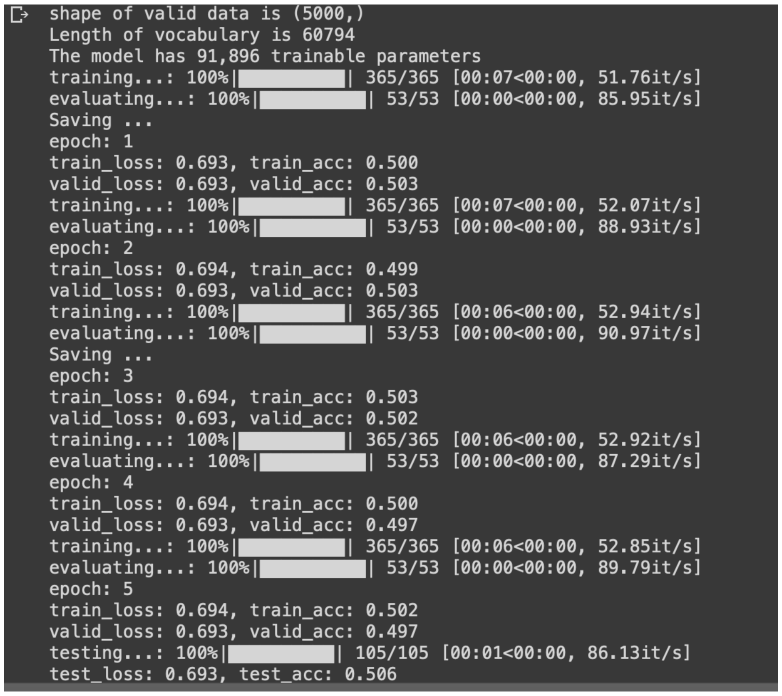
```
        """
        super().__init__()
        # Add your code here. Initializing each layer by the given arguments.
        self.word_embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx = pad_index)
        self.gru = nn.GRU(embedding_dim, hidden_dim,
                          num_layers = n_layers, dropout = dropout_rate, bidirectional = bidirectional)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.sigmoid = nn.Sigmoid()
        self.dropout = nn.Dropout()

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

ii. 2.7.2:



```
    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.word_embedding(ids)
        embed_padding = nn.utils.rnn.pack_padded_sequence(embeds, length, batch_first = True, enforce_sorted = False)
        out, h = self.gru(embed_padding)
        gru_out = h[-1]

        out = self.dropout(gru_out)
        out = self.fc(out)
        # out = self.sigmoid(out)

        # out = out.view(out.size(0), -1)
        # out = out[:,-1]

        prediction = out

        return prediction
```

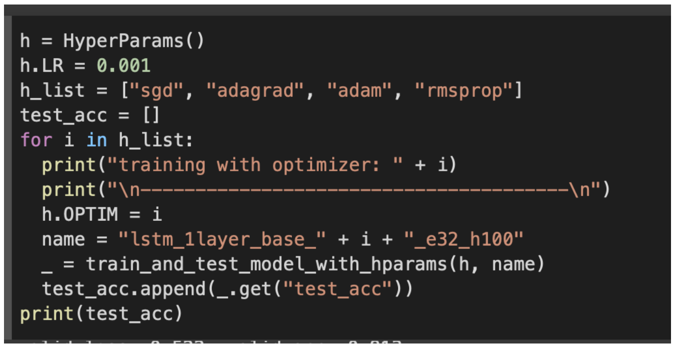
h. Problem 2.8:



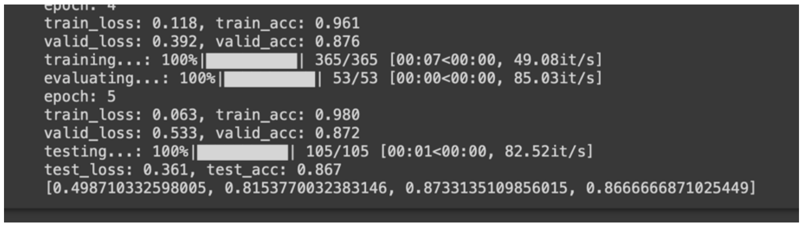
```
shape of valid data is (5000,)
Length of vocabulary is 60794
The model has 91,896 trainable parameters
training...: 100%|██████████| 365/365 [00:07<00:00, 51.76it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 85.95it/s]
Saving ...
epoch: 1
train_loss: 0.693, train_acc: 0.500
valid_loss: 0.693, valid_acc: 0.503
training...: 100%|██████████| 365/365 [00:07<00:00, 52.07it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 88.93it/s]
epoch: 2
train_loss: 0.694, train_acc: 0.499
valid_loss: 0.693, valid_acc: 0.503
training...: 100%|██████████| 365/365 [00:06<00:00, 52.94it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 90.97it/s]
Saving ...
epoch: 3
train_loss: 0.694, train_acc: 0.503
valid_loss: 0.693, valid_acc: 0.502
training...: 100%|██████████| 365/365 [00:06<00:00, 52.92it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 87.29it/s]
epoch: 4
train_loss: 0.694, train_acc: 0.500
valid_loss: 0.693, valid_acc: 0.497
training...: 100%|██████████| 365/365 [00:06<00:00, 52.85it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 89.79it/s]
epoch: 5
train_loss: 0.694, train_acc: 0.502
valid_loss: 0.693, valid_acc: 0.497
testing...: 100%|██████████| 105/105 [00:01<00:00, 86.13it/s]
test_loss: 0.693, test_acc: 0.506
```

We can see that GRU has a slightly higher accuracy than LSTM, but GRU also has the same problem which the accuracy was stuck in a steady value and didn't increase.

3. Lab 2: Training and improving Recurrent Neural Network
   a. Problem 3.1:



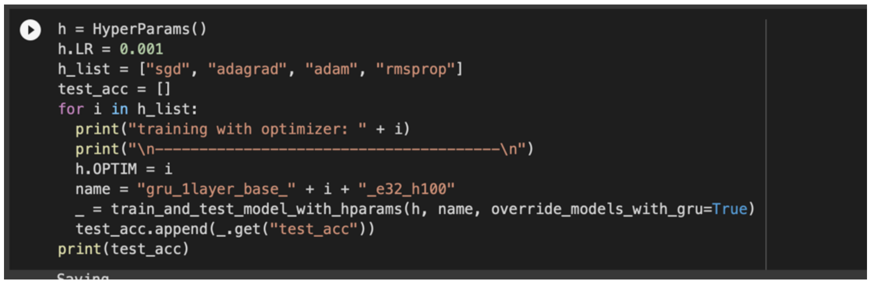
```
h = HyperParams()
h.LR = 0.001
h_list = ["sgd", "adagrad", "adam", "rmsprop"]
test_acc = []
for i in h_list:
  print("training with optimizer: " + i)
  print("\n----------------------------------------\n")
  h.OPTIM = i
  name = "lstm_1layer_base_" + i + "_e32_h100"
  _ = train_and_test_model_with_hparams(h, name)
  test_acc.append(_.get("test_acc"))
print(test_acc)
```



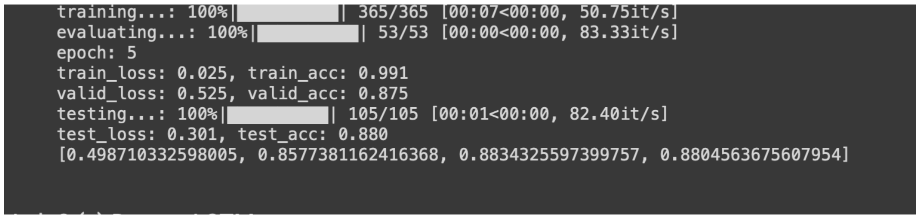
```
epoch: 4
train_loss: 0.118, train_acc: 0.961
valid_loss: 0.392, valid_acc: 0.876
training...: 100%|██████████| 365/365 [00:07<00:00, 49.08it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 85.03it/s]
epoch: 5
train_loss: 0.063, train_acc: 0.980
valid_loss: 0.533, valid_acc: 0.872
testing...: 100%|██████████| 105/105 [00:01<00:00, 82.52it/s]
test_loss: 0.361, test_acc: 0.867
[0.498710332598005, 0.8153770032383146, 0.8733135109856015, 0.8666666871025449]
```

In the experiment we can notice that changing the optimizer result in a very obvious change in the accuracy. The reason why Adam optimizer gives the best accuracy is because Adam adjust its learning rate in corresponding to the training status and is better with dealing with gradient vanish issues.

   b. Problem 3.2:



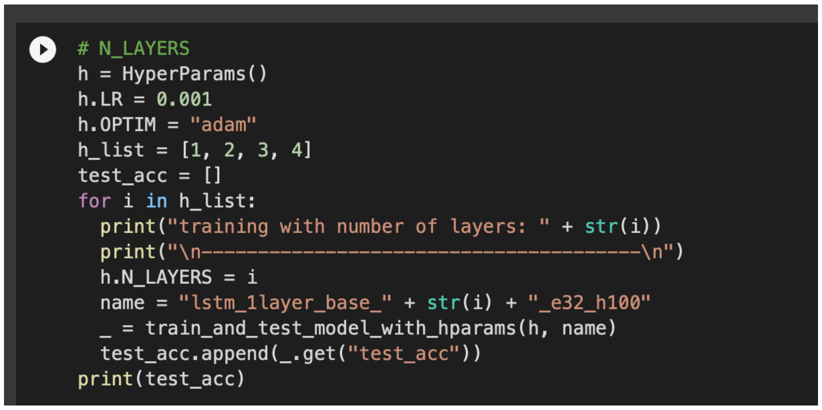
```
h = HyperParams()
h.LR = 0.001
h_list = ["sgd", "adagrad", "adam", "rmsprop"]
test_acc = []
for i in h_list:
  print("training with optimizer: " + i)
  print("\n----------------------------------------\n")
  h.OPTIM = i
  name = "gru_1layer_base_" + i + "_e32_h100"
  _ = train_and_test_model_with_hparams(h, name, override_models_with_gru=True)
  test_acc.append(_.get("test_acc"))
print(test_acc)
```



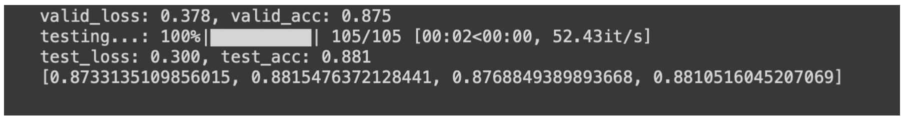
```
training...: 100%|██████████| 365/365 [00:07<00:00, 50.75it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 83.33it/s]
epoch: 5
train_loss: 0.025, train_acc: 0.991
valid_loss: 0.525, valid_acc: 0.875
testing...: 100%|██████████| 105/105 [00:01<00:00, 82.40it/s]
test_loss: 0.301, test_acc: 0.880
[0.498710332598005, 0.8577381162416368, 0.8834325597399757, 0.8804563675607954]
```

The experiment in GRU results the same comparing to LSTM. We can see that Adam has the highest accuracy and in GRU, the result is slightly higher than LSTM.

c. Problem 3.3:



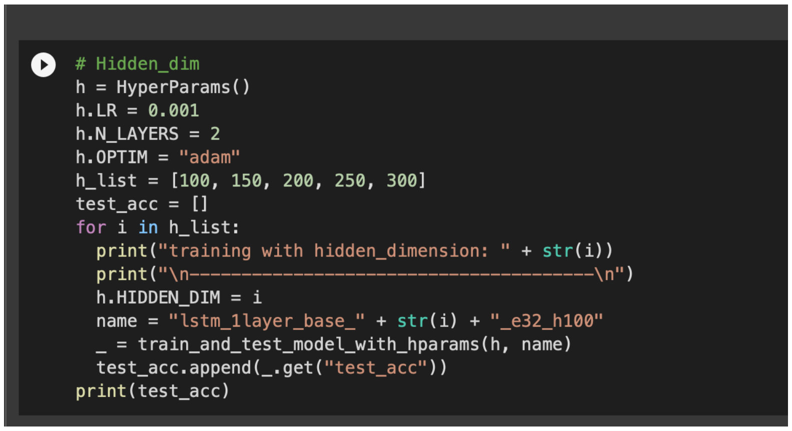
```
# N_LAYERS
h = HyperParams()
h.LR = 0.001
h.OPTIM = "adam"
h_list = [1, 2, 3, 4]
test_acc = []
for i in h_list:
  print("training with number of layers: " + str(i))
  print("\n-----------------------------------------\n")
  h.N_LAYERS = i
  name = "lstm_1layer_base_" + str(i) + "_e32_h100"
  _ = train_and_test_model_with_hparams(h, name)
  test_acc.append(_.get("test_acc"))
print(test_acc)
```



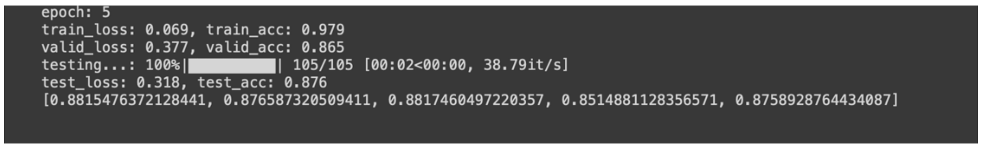
```
valid_loss: 0.378, valid_acc: 0.875
testing...: 100%|██████████| 105/105 [00:02<00:00, 52.43it/s]
test_loss: 0.300, test_acc: 0.881
[0.8733135109856015, 0.8815476372128441, 0.8768849389893668, 0.8810516045207069]
```

We can see that when the number of layer is 2, the accuracy is the highest.

d. Problem 3.4:



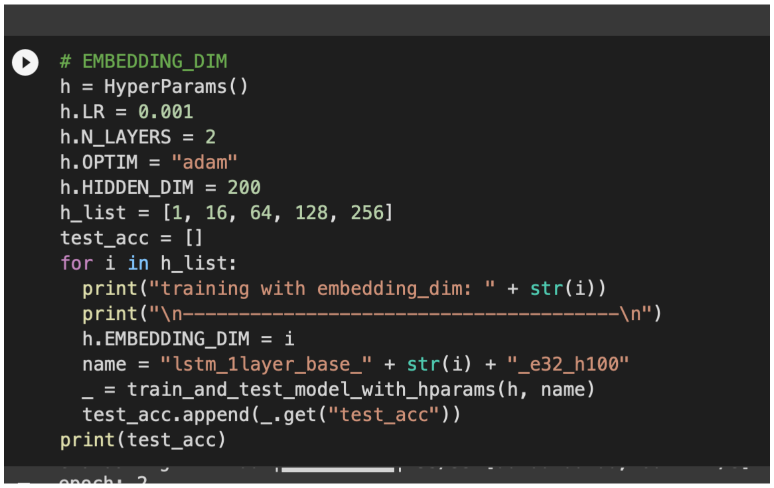
```
# Hidden_dim
h = HyperParams()
h.LR = 0.001
h.N_LAYERS = 2
h.OPTIM = "adam"
h_list = [100, 150, 200, 250, 300]
test_acc = []
for i in h_list:
  print("training with hidden_dimension: " + str(i))
  print("\n-----------------------------------------\n")
  h.HIDDEN_DIM = i
  name = "lstm_1layer_base_" + str(i) + "_e32_h100"
  _ = train_and_test_model_with_hparams(h, name)
  test_acc.append(_.get("test_acc"))
print(test_acc)
```



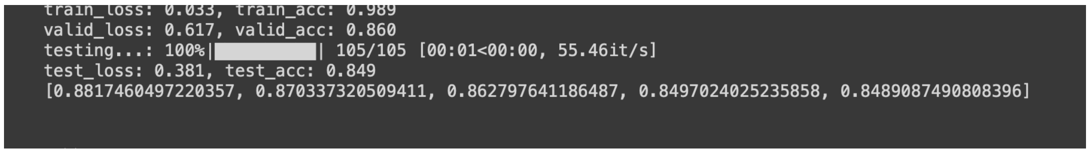
```
epoch: 5
train_loss: 0.069, train_acc: 0.979
valid_loss: 0.377, valid_acc: 0.865
testing...: 100%|██████████| 105/105 [00:02<00:00, 38.79it/s]
test_loss: 0.318, test_acc: 0.876
[0.8815476372128441, 0.876587320509411, 0.8817460497220357, 0.8514881128356571, 0.8758928764434087]
```

We can see that when the hidden units is 200, the accuracy is the highest.

e. Problem 3.5:



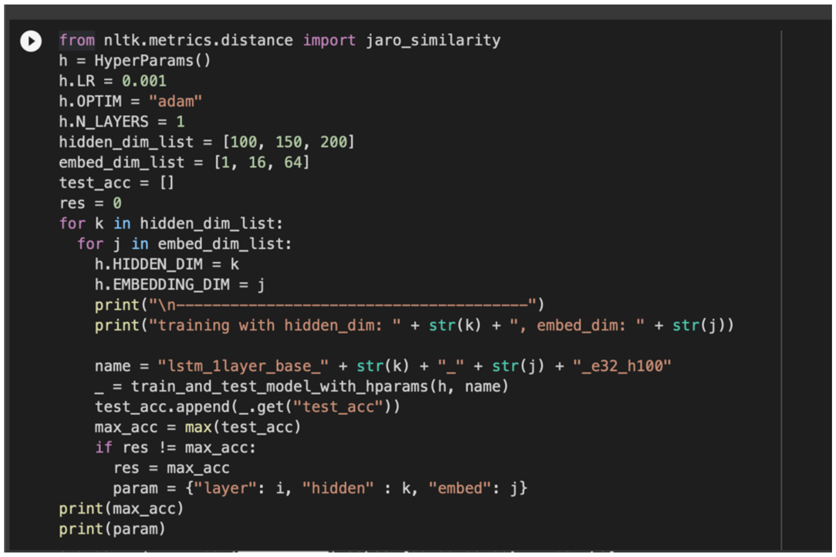
```python
# EMBEDDING_DIM
h = HyperParams()
h.LR = 0.001
h.N_LAYERS = 2
h.OPTIM = "adam"
h.HIDDEN_DIM = 200
h_list = [1, 16, 64, 128, 256]
test_acc = []
for i in h_list:
  print("training with embedding_dim: " + str(i))
  print("\n-----------------------------------------\n")
  h.EMBEDDING_DIM = i
  name = "lstm_1layer_base_" + str(i) + "_e32_h100"
  _ = train_and_test_model_with_hparams(h, name)
  test_acc.append(_.get("test_acc"))
print(test_acc)
```



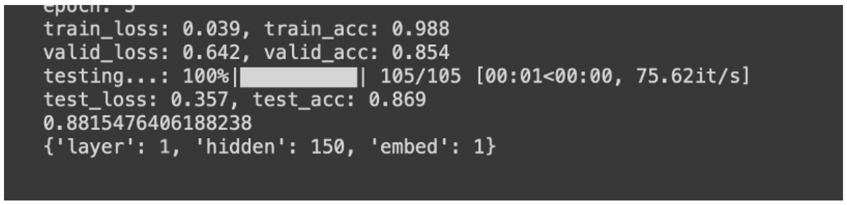
```
train_loss: 0.033, train_acc: 0.989
valid_loss: 0.617, valid_acc: 0.860
testing...: 100%|██████████| 105/105 [00:01<00:00, 55.46it/s]
test_loss: 0.381, test_acc: 0.849
[0.8817460497220357, 0.870337320509411, 0.862797641186487, 0.8497024025235858, 0.8489087490808396]
```

We can see that when the embedded dimension is 1, the accuracy is the highest.

f. Problem 3.6:



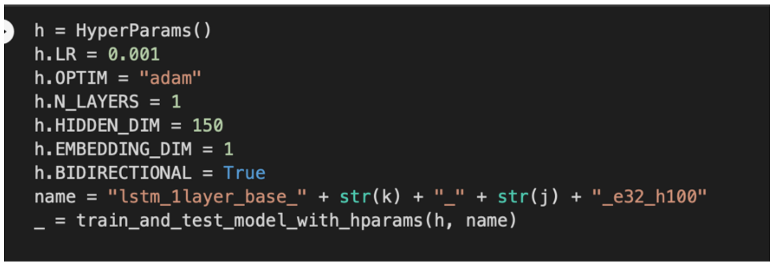
```python
from nltk.metrics.distance import jaro_similarity
h = HyperParams()
h.LR = 0.001
h.OPTIM = "adam"
h.N_LAYERS = 1
hidden_dim_list = [100, 150, 200]
embed_dim_list = [1, 16, 64]
test_acc = []
res = 0
for k in hidden_dim_list:
  for j in embed_dim_list:
    h.HIDDEN_DIM = k
    h.EMBEDDING_DIM = j
    print("\n---------------------------------------")
    print("training with hidden_dim: " + str(k) + ", embed_dim: " + str(j))

    name = "lstm_1layer_base_" + str(k) + "_" + str(j) + "_e32_h100"
    _ = train_and_test_model_with_hparams(h, name)
    test_acc.append(_.get("test_acc"))
    max_acc = max(test_acc)
    if res != max_acc:
      res = max_acc
      param = {"layer": i, "hidden" : k, "embed": j}
print(max_acc)
print(param)
```



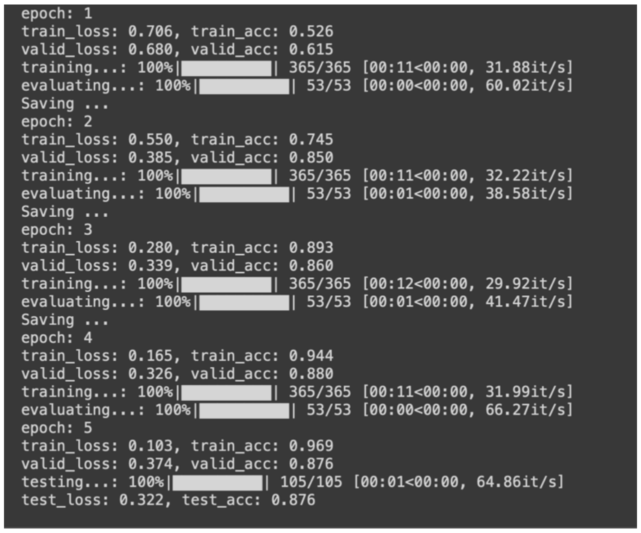
```
train_loss: 0.039, train_acc: 0.988
valid_loss: 0.642, valid_acc: 0.854
testing...: 100%|██████████| 105/105 [00:01<00:00, 75.62it/s]
test_loss: 0.357, test_acc: 0.869
0.8815476406188238
{'layer': 1, 'hidden': 150, 'embed': 1}
```

We can see that when the number of layers is 1, the hidden units is 150, and embedded dimension is 1, the accuracy is the highest.

g. Problem 3.7:



```
h = HyperParams()
h.LR = 0.001
h.OPTIM = "adam"
h.N_LAYERS = 1
h.HIDDEN_DIM = 150
h.EMBEDDING_DIM = 1
h.BIDIRECTIONAL = True
name = "lstm_1layer_base_" + str(k) + "_" + str(j) + "_e32_h100"
_ = train_and_test_model_with_hparams(h, name)
```



```
epoch: 1
train_loss: 0.706, train_acc: 0.526
valid_loss: 0.680, valid_acc: 0.615
training...: 100%|██████████| 365/365 [00:11<00:00, 31.88it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 60.02it/s]
Saving ...
epoch: 2
train_loss: 0.550, train_acc: 0.745
valid_loss: 0.385, valid_acc: 0.850
training...: 100%|██████████| 365/365 [00:11<00:00, 32.22it/s]
evaluating...: 100%|██████████| 53/53 [00:01<00:00, 38.58it/s]
Saving ...
epoch: 3
train_loss: 0.280, train_acc: 0.893
valid_loss: 0.339, valid_acc: 0.860
training...: 100%|██████████| 365/365 [00:12<00:00, 29.92it/s]
evaluating...: 100%|██████████| 53/53 [00:01<00:00, 41.47it/s]
Saving ...
epoch: 4
train_loss: 0.165, train_acc: 0.944
valid_loss: 0.326, valid_acc: 0.880
training...: 100%|██████████| 365/365 [00:11<00:00, 31.99it/s]
evaluating...: 100%|██████████| 53/53 [00:00<00:00, 66.27it/s]
epoch: 5
train_loss: 0.103, train_acc: 0.969
valid_loss: 0.374, valid_acc: 0.876
testing...: 100%|██████████| 105/105 [00:01<00:00, 64.86it/s]
test_loss: 0.322, test_acc: 0.876
```

We can see that although we enabled the bidirectional parameter, the accuracy didn’t went up, it even decreases a bit.