NLP Assignment 2 Report

20191138 Hyungyu Lee

Code Explanation

Problem 1

Code (Notations Provided Omitted to Keep Report Compact)

```
class IMDbData:
  def __init__(self, path_list):
    self.paths = path_list
    self.tokenizer = Tokenizer()

  def __len__(self) -> int:
    return len(self.paths)

  def __getitem__(self, idx:int) -> Tuple[List[str], int]:
    return self.tokenizer(read_txt(self.paths[idx])), ('pos' in
str(self.paths[idx]))*1
```

Explanation

`len(self)`

`self.paths` is a list containing each `path`, such as: `PosixPath('aclImdb/test/urls_neg.txt'), PosixPath('aclImdb/test/urls_pos.txt'), …` . Thus, length of that list, `len(self.paths)` equals the length of the dataset, which is number of total data samples in dataset.

`getitem(self, idx)`

- This method uses parameter `idx` as index to get tokenized `idx`-th (or technically (`idx-1`)th because python starts counting at zero) string and label whether that string is positive or not.
- Since `self.paths` is a list, we can access `idx`-th element with indexing like `self.path[idx]`. We obtain tokenized element by putting it like this: `self.tokenizer(read_txt(self.paths[idx])`.
- In python, `True * 1 == 1` and `False * 1 == 0`. So, we can use some condition to determine the label for the instance. With HINT provided, we can use 'pos' in the path as the condition. Thus, the second returned value can be written as `('pos' in str(self.paths[idx]))*1`.

The code has passed all test cases.

Problem 2

Code

```
class Str2Idx2Str:
  def __init__(self, vocab: List[str]):

    self.idx2str = []
    self.str2idx = {}

    for idx, word in enumerate(vocab):
      self.idx2str.append(word)
      self.str2idx[word] = idx

    self.unknown_idx = len(self.str2idx)
    self.idx2str.append("UNKNOWN")

  def __call__(self, alist:Union[List[str], List[int], str, int, List[List]]) ->
Union[List[int], List[str], int, str, List[List]]:
    if isinstance(alist, str):
      return self.str2idx.get(alist, self.unknown_idx)
    elif isinstance(alist, int):
      return self.idx2str[alist] if 0 <= alist < len(self.idx2str) else "UNKNOWN"
    elif isinstance(alist, list):
      return [self.__call__(element) for element in alist]
```

Explanation

`init`

In the `Str2Idx2Str` class, `self.idx2str` is initialized as a list and `self.str2idx` as a dictionary. The constructor iterates over the vocabulary using a `for` loop, where enumerate avoids the need for redundant index variables. Each word and its corresponding index are stored in str2idx and idx2str, respectively.

Two Given Lines

self.unknown_idx is set as `len(self.str2idx)`, which will be the index of "UNKNOWN", which is appended later. This is used to prevent error at call(self).

`call`

This function works recursively to handle different types of `alist`. If `alist` is a list, the function recursively works to return a list of converted elements. If the given element is a string, then the function will return the corresponding index. `get(key, value)` method on a dictionary returns value if key does not exist in the dictionary. This is the part where `self.unkown_idx` is utilized. If an int is given, it will first check if the given index is in the range of `idx2str`. If the int is negative or larger than length of `idx2str`, "UNKNOWN" will be returned. It might be said that using `len(self.idx2str)-1` will probably yield "UNKOWN" for all unknown words, too. However, explicitly maintaining "UNKNOWN" in `self.idx2str` ensures consistent results when converting indices back to strings and vice versa repeatedly, enhancing stability and predictability when accessing vocabulary elements.

Before Recursively Writing Code and Utilizing Two Lines

```python
res = []

if isinstance(alist, list):
  for one_element in alist: #an element in this megalist(alist) can be int, str, list[int], list[str]
    if isinstance(one_element, str): #alist is list[str]
      try:
        res.append(self.str2idx[one_element])
      except:
        res.append(self.unknown_idx)
    elif isinstance(one_element, int): #alist is list[int]
      try:
        res.append(self.idx2str[one_element])
      except:
        res.append(self.idx2str['-1'])
    elif isinstance(one_element, list): #alist is list[list]. the list that is an element will be called inner_element
      inner_res = []
      for inner_element in one_element:
        if isinstance(inner_element, str): #inner_element is list[str]
          try:
            inner_res.append(self.str2idx[inner_element])
          except:
            inner_res.append(self.unknown_idx)
        elif isinstance(inner_element, int): #inner_element is list[int]
          try:
            inner_res.append(self.idx2str[inner_element])
          except:
            inner_res.append(self.idx2str['-1'])
      res.append(inner_res)
  else:
    if isinstance(alist, str): #alist is str
      try:
        res.append(self.str2idx[alist])
      except:
        res.append(self.unknown_idx)
    elif isinstance(alist, int): #alist is int
      try:
        res.append(self.idx2str[alist])
      except:
        res.append(self.idx2str['-1'])


return res
```

Before professor's hint that this code can be written recursively, I wrote an extensively long code like this. The examples and conditions provided in question had maximum recursive possibility of 2 (list of list of str or int), so this code could pass test cases, too. Try and except is the code that exists because of lack of utilizing two lines. With the hint and two lines provided, I rewrote the code.

The modified code passed all test cases.

Problem 3

Code

```
class PackCollateWithConverter:
  def __init__(self, converter: Str2Idx2Str):
    self.converter = converter

  def __call__(self, batch: List[Tuple[List[str], int]]):

    word_sentences_in_idxs = [torch.LongTensor(self.converter(one_element[0])) for
one_element in batch]
    label_tensor = torch.FloatTensor([one_element[1] for one_element in batch])

    '''
    Leave the code below as it is
    '''
    assert isinstance(word_sentences_in_idxs, list), f"txts_in_idxs has to be a
list, not {type(word_sentences_in_idxs)}"
    assert isinstance(word_sentences_in_idxs[0], torch.LongTensor), f"An elmenet of
txts_in_idxs has to be a torch.LongTensor, not {type(word_sentences_in_idxs[0])}"
    assert isinstance(label_tensor, torch.FloatTensor), f"labels has to be a
torch.FloatTensor, not {type(label_tensor)}"
    assert label_tensor[-1] == batch[-1][1], "i-th element of labels has to be "

    packed_sequence = pack_sequence(word_sentences_in_idxs, enforce_sorted=False)

    return packed_sequence, label_tensor
```

Explanation

- I made `word_sentences_in_idxs` a list of torch.LongTensor, in which each element is a sequence of integers, which are converted index from vocabulary words.
- I made `label_tensor` a torch.FloatTensor, whose `i`-th element corresponds to label of `word_sentences_in_idx`'s `i`-th data sample in the batch, which is either 0.0 or 1.0

I first declared `word_sentenxes_in_idxs` as an empty array, and `label_tensor` as `torch.full((len(batch), ), -1.0)`, with -1.0 as initializing value, then set each variable like below. But used instruction to change two lines to further shorten and improve my code.

```
  word_sentences_in_idxs = [] # you have to change this
  label_tensor = torch.full((len(batch), ), -1.0) # you have to change this,
first declare to 1

  pos = 0
    for one_element in batch:

word_sentences_in_idxs.append(torch.LongTensor(self.converter(one_element[0])))
      label_tensor[pos] = torch.FloatTensor([one_element[1]])  # Fix here
      pos += 1
```

The shell ran successfully, with output like this:

```
A batch looks like this:  (PackedSequence(data=tensor([  493, 24238, 22360,  ...,
15129, 17072,    41]), batch_sizes=tensor([32, 32, 32,  ...,  1,  1,  1]),
sorted_indices=tensor([ 3, 12, 18,  0, 13, 23, 10, 14, 29, 30, 21, 15,  9, 25, 26,
28,  2, 16,
         6,  8, 19, 20,  1,  4, 27, 22, 11, 31,  5, 17, 24,  7]),
unsorted_indices=tensor([ 3, 22, 16,  0, 23, 28, 18, 31, 19, 12,  6, 26,  1,  4,  7,
11, 17, 29,
         2, 20, 21, 10, 25,  5, 30, 13, 14, 24, 15,  8,  9, 27])), tensor([0., 1.,
1., 0., 0., 1., 0., 1., 1., 1., 1., 0., 1., 1., 1., 0., 0., 0.,
        0., 0., 1., 0., 0., 1., 1., 1., 1., 0., 1., 1., 0., 1.]))
```

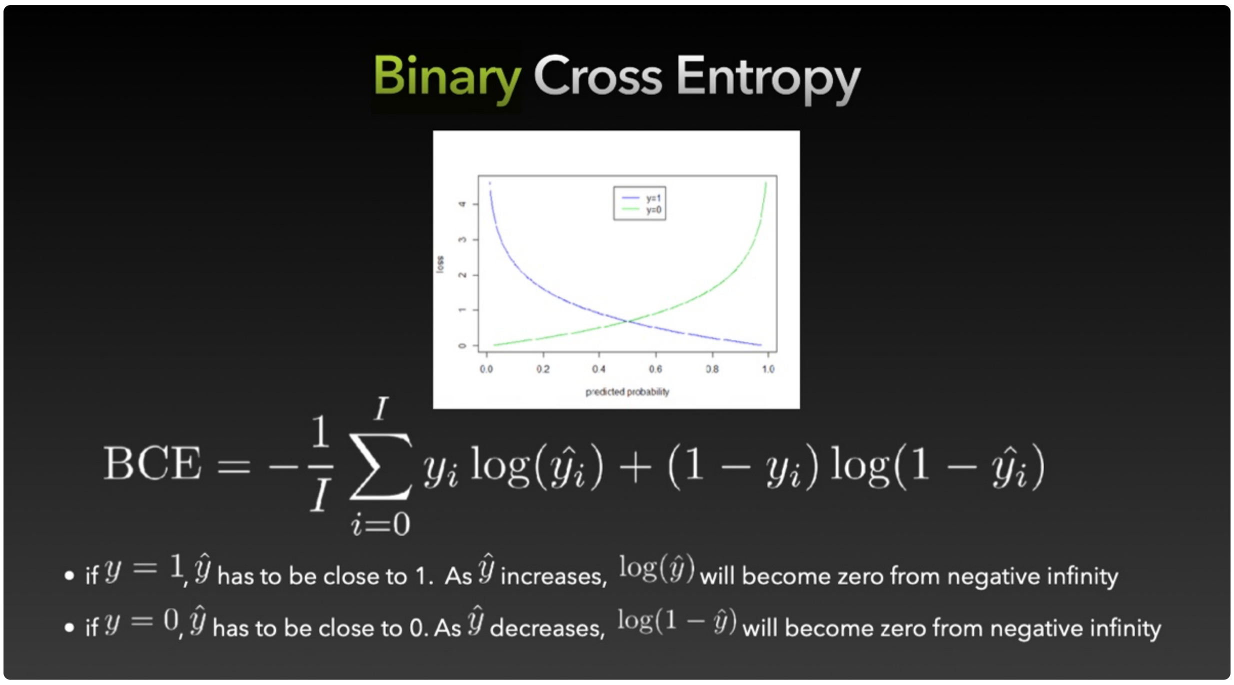
Problem 4

Code

```
def get_binary_cross_entropy_loss(pred:torch.FloatTensor, target:torch.FloatTensor,
eps=1e-8) -> torch.FloatTensor:

  res_tensor = torch.full((pred.size(dim=0), ), .0)
  for i in range(pred.size(dim=0)):
    this_pred = pred[i] + eps
    res_tensor[i] = target[i] * torch.log(this_pred) + (1-target[i]) * torch.log(1-
this_pred)

  return -res_tensor.mean()
```

Explanation

I intuitively wrote the equation for BCE provided in our 4th lecture slide, page 13:


y=1
y=0
loss
predicted probability
0.0
0.2
0.4
0.6
0.8
1.0
0
1
2
3
4

Binary Cross Entropy

$$\text{BCE} = -\frac{1}{I}\sum_{i=0}^{I} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

- if $y = 1$, $\hat{y}$ has to be close to 1. As $\hat{y}$ increases, $\log(\hat{y})$ will become zero from negative infinity
- if $y = 0$, $\hat{y}$ has to be close to 0. As $\hat{y}$ decreases, $\log(1 - \hat{y})$ will become zero from negative infinity

Binary Cross-Entropy (BCE) is negative average of the sum for all instances in dataset, where each term is the sum of the target value * log(predicted probability) plus (1 - target value)* log(1 - predicted probability). To prevent computational errors, especially taking the logarithm of zero, a small constant, `eps` (epsilon), is added to the predictions. This constant ensures numerical stability.

Evaluation

After running the test cell provided, I could see that my get_binary_cross_entropy_loss was written correctly, with following output: `BCE Loss by torch.nn.BCELoss is 0.09634757786989212 and your BCE loss is 0.09634757786989212`.

Problem 5

Code and Explanation

Since this code block is extensively long, I divided the code into subparts.

Initialization

These parts are provided.

```
from tqdm.auto import tqdm

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class Trainer:
  def __init__(self, model, optimizer, loss_fn, train_loader, valid_loader, device):
    self.model = model
    self.optimizer = optimizer
    self.loss_fn = torch.nn.BCELoss()
    self.train_loader = train_loader
    self.valid_loader = valid_loader

    self.model.to(device)

    self.best_valid_accuracy = 0
    self.device = device

    self.training_loss = []
    self.training_acc = []
    self.validation_loss = []
    self.validation_acc = []

  def save_model(self, path='imdb_sentiment_model.pt'):
    torch.save({'model':self.model.state_dict(),
'optim':self.optimizer.state_dict()}, path)

  def train_by_num_epoch(self, num_epochs):
    for epoch in range(num_epochs):
      self.model.train()
      for batch in tqdm(self.train_loader, leave=False):
        loss_value, acc = self._train_by_single_batch(batch)
        self.training_loss.append(loss_value)
        self.training_acc.append(acc)

      self.model.eval()
      validation_loss, validation_acc = self.validate()
      self.validation_loss.append(validation_loss)
      self.validation_acc.append(validation_acc)

      print(f"Epoch {epoch+1}, Training Loss: {loss_value:.4f}, Training Acc:
{acc:.4f}, Validation Loss: {validation_loss:.4f}, Validation Acc:
{validation_acc:.4f}")
      if validation_acc > self.best_valid_accuracy:
        print(f"Saving the model with best validation accuracy: Epoch {epoch+1},
Acc: {validation_acc:.4f} ")
        self.save_model('imdb_sentiment_model_best.pt')
      else:
        self.save_model('imdb_sentiment_model_last.pt')
      self.best_valid_accuracy = max(validation_acc, self.best_valid_accuracy)
```

`_get_accuracy`

```
def _get_accuracy(self, pred, target, threshold=0.5):

  predictions = (pred > threshold).float()
  predictions_equals_count = (predictions == target).float().sum().item()

  return predictions_equals_count / correct_predictions.size(0)
```

I made torch variable `predictions`, which is torch of Boolean tensors, which indicate if the predicted value is larger than threshold, converted to float Tensors. `True` is converted to 1.0, and `False` into 0.0. Then, torch variable `predictions_equals_count` uses another Boolean tensors, predictions == target, then changes it into float. The float value will be 1.0 if the model worked accurately(true positive and true negative). Sum of it, which is float tensor, has its float value taken out with item() command. Thus, `predictions_equals_count` is a float value, which would likely be a number of accurate predictions. That is divided with number of total cases to return accuracy of the model, which is still a float value.

`_get_loss_and_acc_from_single_batch`

```
def _get_loss_and_acc_from_single_batch(self, batch):

  self.device='cuda'

  #Move data to device
  inputs, labels = batch
  inputs, labels = inputs.to(self.device), labels.to(self.device)

  #forward pass
  predictions = self.model(inputs)

  #calculate loss and accuracy
  loss = self.loss_fn(predictions, labels)
  accuracy = self._get_accuracy(predictions, labels)

  return loss, accuracy  # Assuming acc is already a float and loss is a tensor
```

I used `self.device(‘cuda’)` to utilize GPU for this neural network. I extracted inputs and actual labels from the batch, then moved it to device. Without this code, I experienced `RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!` error. I made predictions with self.model(), then calculated loss and accuracy. `loss_fn` returns `-res_tensor.mean()`, which is still a torch.Tensor, not a float.

`_train_by_single_batch`

```
def _train_by_single_batch(self, batch):

  self.model.train()

  inputs, labels = batch
  inputs, labels = inputs.to(self.device), labels.to(self.device)

  self.optimizer.zero_grad()
  loss, acc = self._get_loss_and_acc_from_single_batch((inputs, labels))
  loss.backward()
  self.optimizer.step()

  return loss.item(), acc
```

First, we put model into train mode using `self.model.train()`. We don't put that in `_get_loss_and_acc_from_single_batch` function because it is also used in validation phase, too. We move inputs and labels to the device to prevent `RuntimeError` like before, then set optimizer to zero_grad before calculating loss and accuracy, then make it train/optimize model with the loss obtained. Then the float value of loss and accuracy is returned.

`validate`

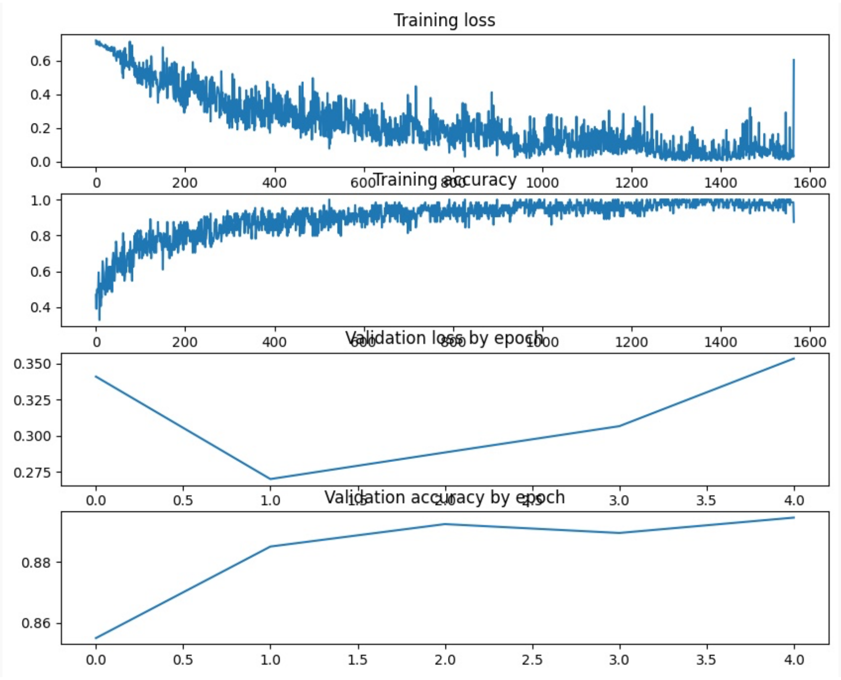
```
def validate(self, external_loader=None):

  ### Don't change this part
  if external_loader and isinstance(external_loader, DataLoader):
    loader = external_loader
    print('An arbitrary loader is used instead of Validation loader')
  else:
    loader = self.valid_loader

  #enter eval 모드
  self.model.eval()

  '''
  Write your code from here, using loader, self.model, self.loss_fn.
  '''

  total_loss = 0.0
  total_accuracy = 0.0
  total_count = 0

  #not calculate gradation
  with torch.no_grad():
    for batch in loader:
        inputs, labels = batch
        inputs, labels = inputs.to(self.device), labels.to(self.device)

        batch_loss, batch_acc = self._get_loss_and_acc_from_single_batch((inputs,
labels))

          total_loss += batch_loss.item()
          total_accuracy += batch_acc
          total_count += 1

    return total_loss/total_count, total_accuracy/total_count
```

This method is called to validate the model. `self.model.eval()` is called to to put model in evaluation(inference)mode. `total_loss`, `total_accuracy`, `total_count` is variables that stores information about results from batch. `torch.no_grad()` disables gradation calculation, thus making the model run faster. Similar to train, loss and accuracy is obtained and stored in corresponding variables per batch. Then, total loss and accuracy is divided with count to get average value.

Training and Plots

Result

```
Epoch 1, Training Loss: 0.3426, Training Acc: 0.8750, Validation Loss: 0.3408,
Validation Acc: 0.8549
Saving the model with best validation accuracy: Epoch 1, Acc: 0.8549
Epoch 2, Training Loss: 0.2821, Training Acc: 0.9062, Validation Loss: 0.2701,
Validation Acc: 0.8852
Saving the model with best validation accuracy: Epoch 2, Acc: 0.8852
Epoch 3, Training Loss: 0.1619, Training Acc: 0.9375, Validation Loss: 0.2885,
Validation Acc: 0.8926
Saving the model with best validation accuracy: Epoch 3, Acc: 0.8926
Epoch 4, Training Loss: 0.0285, Training Acc: 1.0000, Validation Loss: 0.3066,
Validation Acc: 0.8896
Epoch 5, Training Loss: 0.6035, Training Acc: 0.8750, Validation Loss: 0.3533,
Validation Acc: 0.8947
Saving the model with best validation accuracy: Epoch 5, Acc: 0.8947
```



> Test Loss: 0.36901609668014, Test Accuracy: 0.8918568543488825

The training loss starts relatively high and decreases as the epochs progress. This indicates that the model is learning and improving its predictions on the training set. However, there is a noticeable spike in the training loss in the final epoch. This could indicate an issue with that particular batch of training data, or data being erroneous, etc.

The training accuracy shows improvement over time, reaching 100% accuracy by the fourth epoch. But accuracy is not something that is always better when high. This might be a sign of overfitting, especially when the model's accuracy on the training set is perfect, while validation accuracy is not. I think model learns the training data too well, including its noise and details, instead of generalizing to new data.

The validation loss decreases initially but starts to increase again from the third epoch onwards, while the validation accuracy improves slightly with each epoch. The rising validation loss with increasing accuracy could suggest that the model is becoming more confident in its incorrect predictions, or that there's a discrepancy between how loss and accuracy are being calculated or interpreted.

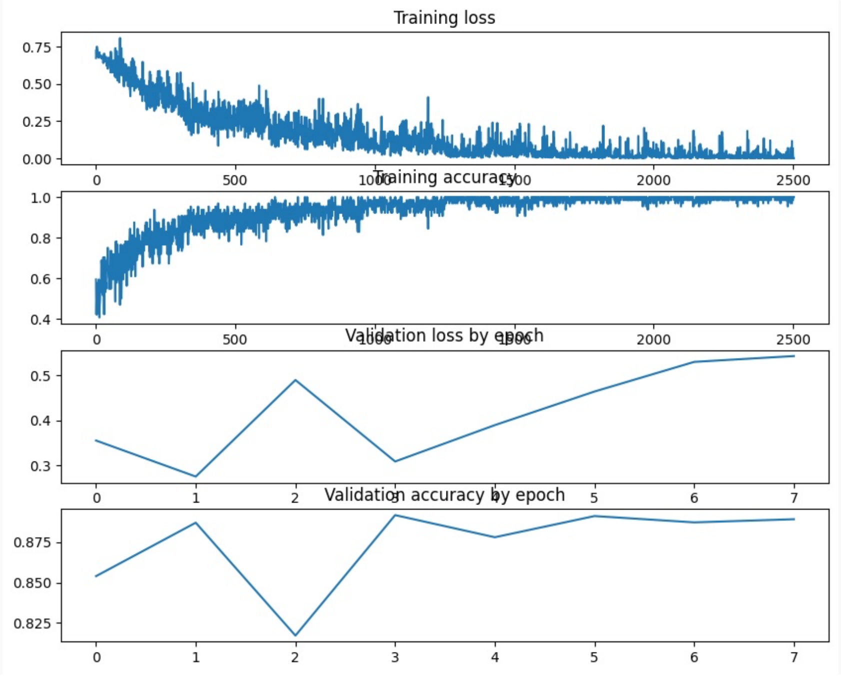
Optimizing

> Optimizing description ends at page 19.

`num_epochs`

- I first tried to optimize model by increasing num_epochs. The result of `trainer.train_by_num_epoch(8)` was as follows:

```
Epoch 1, Training Loss: 0.4430, Training Acc: 0.7812, Validation Loss: 0.3551,
Validation Acc: 0.8539
Saving the model with best validation accuracy: Epoch 1, Acc: 0.8539
Epoch 2, Training Loss: 0.2258, Training Acc: 0.8750, Validation Loss: 0.2748,
Validation Acc: 0.8869
Saving the model with best validation accuracy: Epoch 2, Acc: 0.8869
Epoch 3, Training Loss: 0.3179, Training Acc: 0.8438, Validation Loss: 0.4899,
Validation Acc: 0.8172
Epoch 4, Training Loss: 0.0415, Training Acc: 1.0000, Validation Loss: 0.3084,
Validation Acc: 0.8916
Saving the model with best validation accuracy: Epoch 4, Acc: 0.8916
Epoch 5, Training Loss: 0.0179, Training Acc: 1.0000, Validation Loss: 0.3894,
Validation Acc: 0.8779
Epoch 6, Training Loss: 0.0371, Training Acc: 0.9688, Validation Loss: 0.4643,
Validation Acc: 0.8910
Epoch 7, Training Loss: 0.0517, Training Acc: 0.9688, Validation Loss: 0.5303,
Validation Acc: 0.8871
Epoch 8, Training Loss: 0.0011, Training Acc: 1.0000, Validation Loss: 0.5433,
Validation Acc: 0.8891
```



> Test Loss: 0.5867114421360347, Test Accuracy: 0.8790201348396501

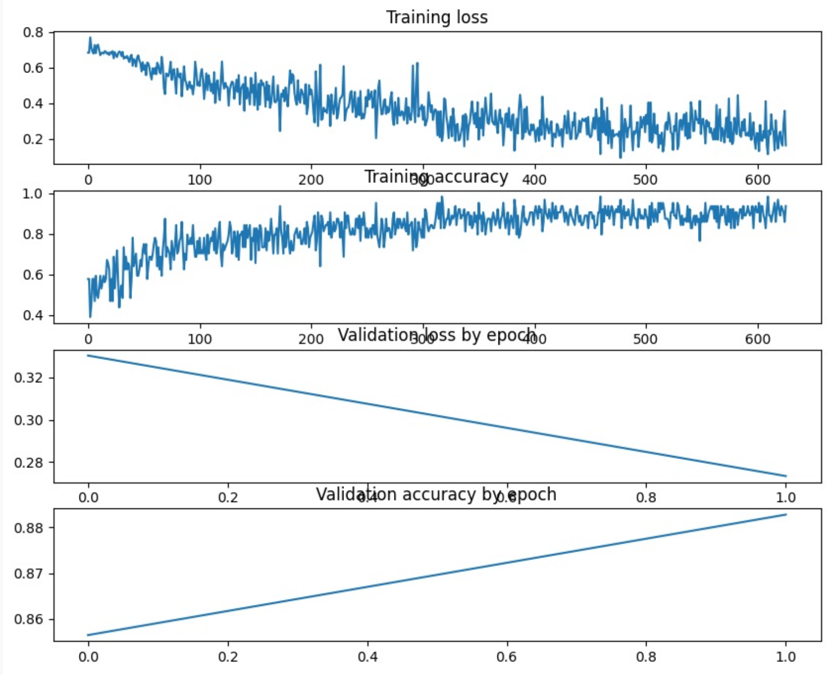
The training loss decreases significantly over the epochs, which is a good sign that the model is learning and adapting to the training data. It seems even better than default situation, in which there were 5 epochs. However, training accuracy hits 1.0 multiple times, which is not necessarily good.

The validation loss initially decreases, which is desirable. However, it starts to increase from the third epoch onward, despite fluctuations in validation accuracy. The increasing loss alongside the fluctuating accuracy could suggest that the model is beginning to overfit to the training data. The highest validation accuracy is achieved in epoch 4, but it does not significantly improve afterwards. This lack of improvement, combined with increasing loss, again suggests possible overfitting past this point.

Overall, improving epoch, when overfitting might have happened before was not desirable. Loss increased and Accuracy decreased.

- Then, I tried to optimize model by decreasing num_epochs. The result of `trainer.train_by_num_epoch(2)` was as follows: It was not as good, because although having too much epoch can be harmful, not having enough training is worse.

```
Epoch 1, Training Loss: 0.3550, Training Acc: 0.9062, Validation Loss: 0.3302,
Validation Acc: 0.8564
Saving the model with best validation accuracy: Epoch 1, Acc: 0.8564
Epoch 2, Training Loss: 0.1622, Training Acc: 0.9375, Validation Loss: 0.2734,
Validation Acc: 0.8828
Saving the model with best validation accuracy: Epoch 2, Acc: 0.8828
```



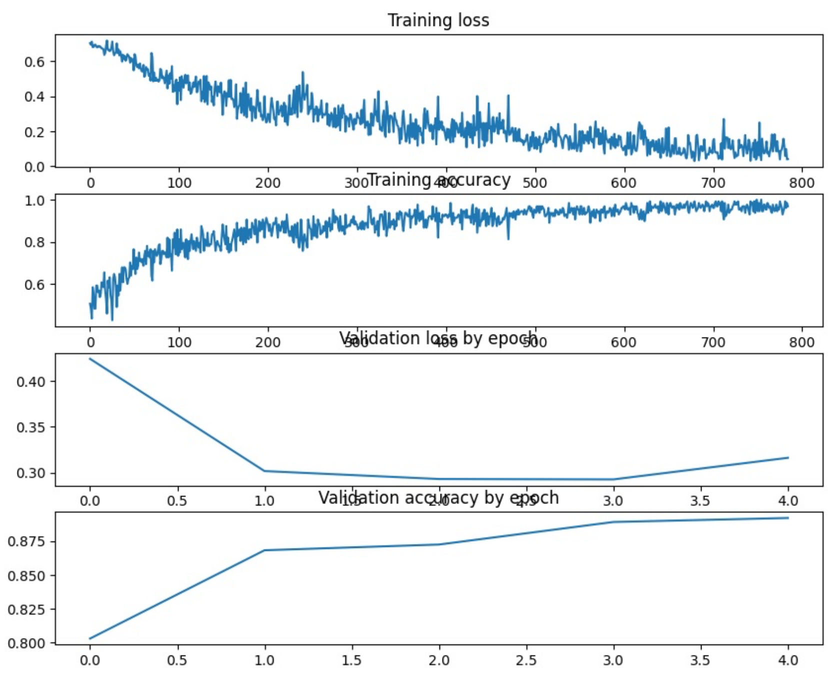
> Test Loss: 0.2821752068643667, Test Accuracy: 0.8796996477162294

`batch_size`

- I wanted to see if increasing batch_size will be helpful.

```
model = SentimentModel(len(converter.idx2str), hidden_size=128, num_layers=3)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
train_loader = DataLoader(trainset, batch_size=128, collate_fn=pack_collate,
shuffle=True)
valid_loader = DataLoader(validset, batch_size=256, collate_fn=pack_collate,
shuffle=False)
test_loader = DataLoader(testset, batch_size=256, collate_fn=pack_collate,
shuffle=False)

trainer =  Trainer(model, optimizer, get_binary_cross_entropy_loss, train_loader,
valid_loader, device='cuda')
```

```
Epoch 1, Training Loss: 0.4923, Training Acc: 0.7500, Validation Loss: 0.4242,
Validation Acc: 0.8031
Saving the model with best validation accuracy: Epoch 1, Acc: 0.8031
Epoch 2, Training Loss: 0.1795, Training Acc: 0.9688, Validation Loss: 0.3017,
Validation Acc: 0.8683
Saving the model with best validation accuracy: Epoch 2, Acc: 0.8683
Epoch 3, Training Loss: 0.4057, Training Acc: 0.8125, Validation Loss: 0.2930,
Validation Acc: 0.8726
Saving the model with best validation accuracy: Epoch 3, Acc: 0.8726
Epoch 4, Training Loss: 0.1518, Training Acc: 0.9375, Validation Loss: 0.2926,
Validation Acc: 0.8892
Saving the model with best validation accuracy: Epoch 4, Acc: 0.8892
Epoch 5, Training Loss: 0.0419, Training Acc: 0.9688, Validation Loss: 0.3162,
Validation Acc: 0.8922
Saving the model with best validation accuracy: Epoch 5, Acc: 0.8922
```



Test Loss: 0.32208898891599813, Test Accuracy: 0.8853574242196879

The training loss shows a generally decreasing trend which is a positive indicator that the model is learning from the training data. There's some fluctuation, which is normal in training processes, especially when dealing with stochastic gradient descent and its variants.

The training accuracy improves over time. There's a dip in the third epoch, which could be due to the variance in the difficulty of batches or could be an indication of the learning rate being too high, causing the model to miss the minimum of the loss function.

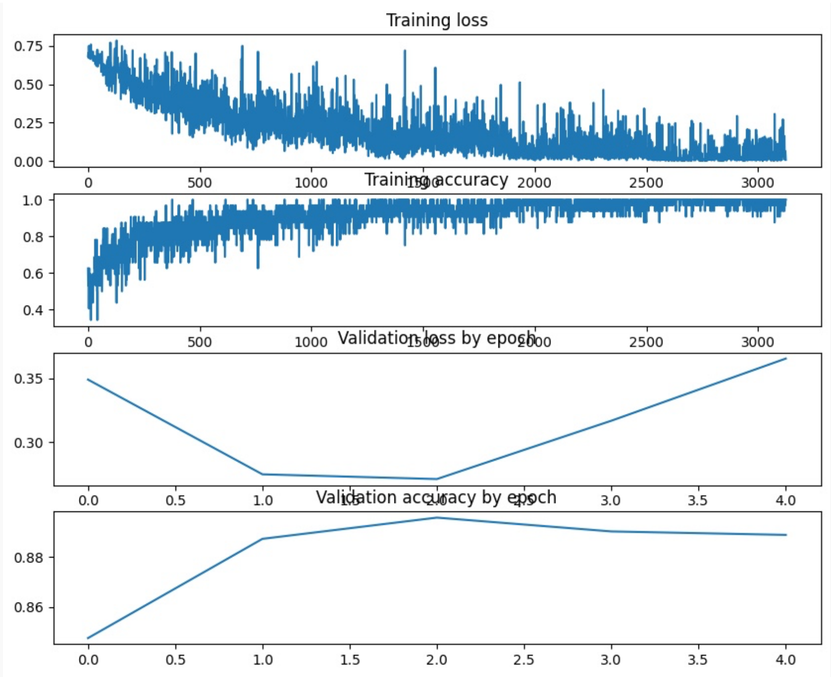
The validation loss decreases until the fourth epoch and then starts to increase slightly. The initial decrease is good as it indicates that the model is generalizing well to unseen data; however, the subsequent increase might suggest the onset of overfitting.

The validation accuracy improves with each epoch, suggesting that, overall, the model's generalization is improving. The best validation accuracy is at the fifth epoch (89.22%).

- Decreasing batch_size using code below returned:

```
model = SentimentModel(len(converter.idx2str), hidden_size=128, num_layers=3)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
train_loader = DataLoader(trainset, batch_size=32, collate_fn=pack_collate,
shuffle=True)
valid_loader = DataLoader(validset, batch_size=64, collate_fn=pack_collate,
shuffle=False)
test_loader = DataLoader(testset, batch_size=64, collate_fn=pack_collate,
shuffle=False)

trainer =  Trainer(model, optimizer, get_binary_cross_entropy_loss, train_loader,
valid_loader, device='cuda')
trainer.train_by_num_epoch(5)
```

```
Epoch 1, Training Loss: 0.1933, Training Acc: 0.9062, Validation Loss: 0.3490,
Validation Acc: 0.8475
Saving the model with best validation accuracy: Epoch 1, Acc: 0.8475
Epoch 2, Training Loss: 0.2756, Training Acc: 0.8750, Validation Loss: 0.2750,
Validation Acc: 0.8873
Saving the model with best validation accuracy: Epoch 2, Acc: 0.8873
Epoch 3, Training Loss: 0.2819, Training Acc: 0.9062, Validation Loss: 0.2713,
Validation Acc: 0.8958
Saving the model with best validation accuracy: Epoch 3, Acc: 0.8958
Epoch 4, Training Loss: 0.2313, Training Acc: 0.8750, Validation Loss: 0.3169,
Validation Acc: 0.8902
Epoch 5, Training Loss: 0.0076, Training Acc: 1.0000, Validation Loss: 0.3655,
Validation Acc: 0.8888
```



> Test Loss 0.3902655012543549, Test Accuracy: 0.8809695073681647

The training loss decreases over time, which is an indication of learning. However, the variability in the loss indicates that the model might be reacting to the noise in the training data or that certain batches are significantly harder to learn from than others.

The training accuracy improves to perfect accuracy by the fifth epoch. This may suggest that the model has effectively learned the training data. However, achieving 100% accuracy is often a red flag for overfitting, especially when it isn't mirrored by a comparable validation accuracy.

I think only increasing batch size was helpful in terms of optimizing the model. Then I tried two different optimizer, then decreased model size, and chose the best working model.

`Optimizer`

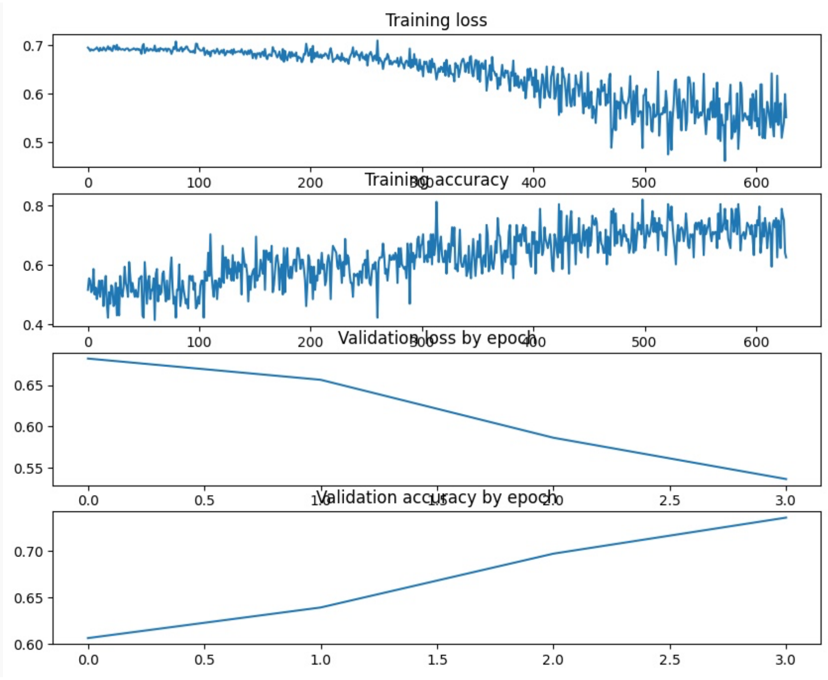
- Using Different Optimizer - SGD

I have tried implementing SGD instead of Adam to this model.

```
model = SentimentModel(len(converter.idx2str), hidden_size=128, num_layers=3)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

...

trainer.train_by_num_epoch(4)
```

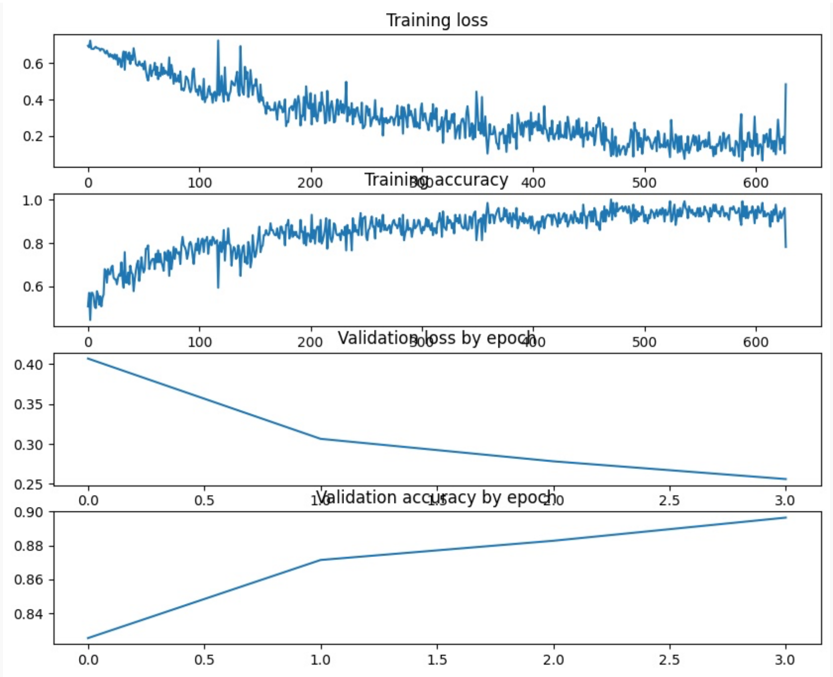
With altered values, our model, with Adam optimizer returned:

```
Epoch 1, Training Loss: 0.6841, Training Acc: 0.5938, Validation Loss: 0.6822,
Validation Acc: 0.6062
Saving the model with best validation accuracy: Epoch 1, Acc: 0.6062
Epoch 2, Training Loss: 0.6293, Training Acc: 0.8125, Validation Loss: 0.6565,
Validation Acc: 0.6393
Saving the model with best validation accuracy: Epoch 2, Acc: 0.6393
Epoch 3, Training Loss: 0.4882, Training Acc: 0.7812, Validation Loss: 0.5863,
Validation Acc: 0.6973
Saving the model with best validation accuracy: Epoch 3, Acc: 0.6973
Epoch 4, Training Loss: 0.5512, Training Acc: 0.6250, Validation Loss: 0.5363,
Validation Acc: 0.7361
Saving the model with best validation accuracy: Epoch 4, Acc: 0.7361
```



> Test Loss: 0.5399094150990856, Test Accuracy: 0.7282670880852341

The results were disappointing. I chose to use Adam, epoch-size: 5, batch size: twice from provided code. Then did final optimization, which is trying to decreasing the model size. I think with less parametrization towards sample data, maybe issue of overfitting might be resolved. This model, which is candidate for well-optimized model returned test result as follows:



> Test Loss: 0.279983499372492, Test Accuracy: 0.8843609318727491

For comparison, pre-optimized model’s test result was:

> Test Loss: 0.3594953169171907, Test Accuracy: 0.8908983236151603

Test accuracy dropped by 0.7%, while loss decreased about 22.1%. Thus, it could be said that this optimization was helpful.

Model Size

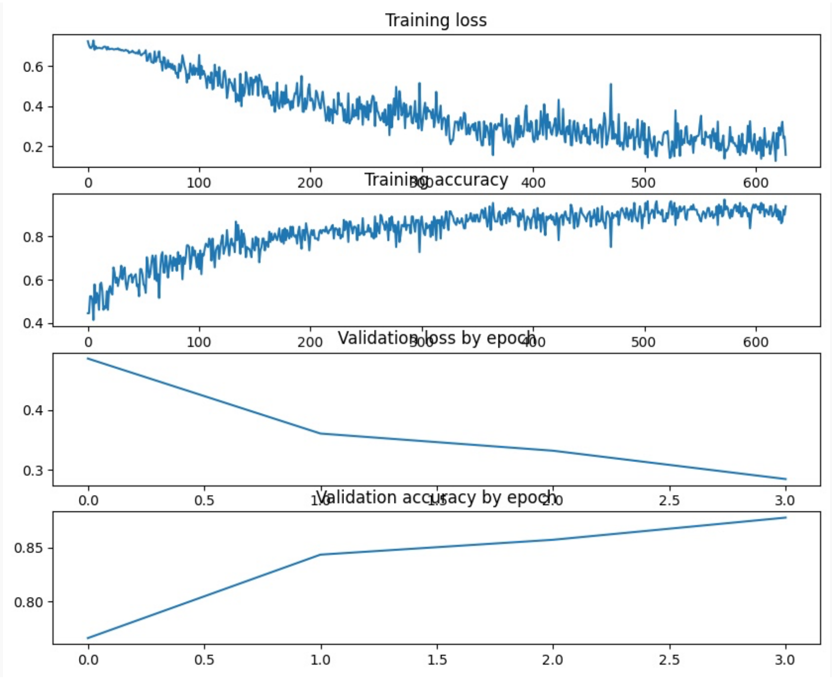
- Decreasing Model size

I wanted to see if decreasing the model size from the best model we obtained above will be helpful. I just halved the model size from hidden_size 128 to 64. I intuitively thought that since what I had to deal with the most in optimization process was overfitting, increasing model size may be unhelpful.

```
model = SentimentModel(len(converter.idx2str), hidden_size=64, num_layers=3)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

...

trainer.train_by_num_epoch(4)
```

And the result was as follows:

```
Epoch 1, Training Loss: 0.5247, Training Acc: 0.7500, Validation Loss: 0.4851,
Validation Acc: 0.7656
Saving the model with best validation accuracy: Epoch 1, Acc: 0.7656
Epoch 2, Training Loss: 0.4719, Training Acc: 0.7812, Validation Loss: 0.3606,
Validation Acc: 0.8436
Saving the model with best validation accuracy: Epoch 2, Acc: 0.8436
Epoch 3, Training Loss: 0.5111, Training Acc: 0.7500, Validation Loss: 0.3321,
Validation Acc: 0.8574
Saving the model with best validation accuracy: Epoch 3, Acc: 0.8574
Epoch 4, Training Loss: 0.1584, Training Acc: 0.9375, Validation Loss: 0.2851,
Validation Acc: 0.8781
Saving the model with best validation accuracy: Epoch 4, Acc: 0.8781
```



> Test Loss: 0.30322808255346456, Test Accuracy: 0.8709080507202881

Both in terms of test loss and test accuracy, the model was not improved. That is, decreasing the model size was not helpful.

Conclusion

For problem, 6, I decided to increase the batch size, and set the num_epoch to 4. I used the model as follows:

```
model = SentimentModel(len(converter.idx2str), hidden_size=128, num_layers=3)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
train_loader = DataLoader(trainset, batch_size=128, collate_fn=pack_collate,
shuffle=True)
valid_loader = DataLoader(validset, batch_size=128, collate_fn=pack_collate,
shuffle=False)
test_loader = DataLoader(testset, batch_size=256, collate_fn=pack_collate,
shuffle=False)

trainer =  Trainer(model, optimizer, get_binary_cross_entropy_loss, train_loader,
valid_loader, device='cuda')
trainer.train_by_num_epoch(4)
```

Running Code

After using `!python3 NLP_Assignment_2.py`, the result was like this:

```
Your BCE result: 0.09634759277105331
Loss value for 10 repetition for the same training batch is  ['0.6839', '0.6763',
'0.6693', '0.6626', '0.6562', '0.6500', '0.6441', '0.6383', '0.6325', '0.6268']
An arbitrary loader is used instead of Validation loader
Valid loss: 0.7093917727470398, Accuracy: 0.5
Epoch 1, Training Loss: 0.6803, Training Acc: 0.5625, Validation Loss: 0.7060,
Validation Acc: 0.5000
Saving the model with best validation accuracy: Epoch 1, Acc: 0.5000
Last 5 Training loss: [0.6652844548225403, 0.674573540687561, 0.7526171207427979,
0.6637386679649353, 0.6803191900253296]
```

Loss value for 10 repetition for the same training batch was decreasing over time, so I could conclude that the model was built well. The training performance in terms of loss was not as good as the provided example, but I think that can be just affected by random values.

Problem 6

Choosing Models

I chose to use the best model. Since our last optimization trial was using size 64 instead of 128, I had to run optimized model declaration and train mode again.

Explanation

Most Incorrect Predictions

Interestingly, when I manually processed the most incorrect prediction sets, the label seemed to be wrong for the most incorrect predictions. For example, my model made prediction of 0.0007 for sample index 23331, which is like this: `David Morse and Andre Braugher are very talented actors, which is why I'm trying so hard to support this program. Unfortunately, an irrational plot, and very poor writing is making it difficult for me. (`…`).` This is a negative review, but I think the labels are made just based on the first sentence without considering the whole review. On the other hand, the model considered whole review, found out that there were more negative sentences, and I think that's how it led to the conclusion that this review is negative.

This issue is persistent, but the common thing for such prediction is that the predicted value is either less than 0.001 or more than 0.999. Maybe this is the reason why model should not be overfitted, or we should refrain from believing model's drastic choices. The problematic 'labeled-as-incorrect' prediction value was as follows: `0.0007(1−0.9993), 0.0010(1−0.9990), 0.9989, 0.9989, 0.9988, 0.9987`.

The interesting incorrect predictions, which is sample index 17174 and 22698. 17174 is praising the film by making negative remarks on other film, with the latter scathing review taking up the majority of the text. This got predicted value of 0.0014, while correct label is 1. I think this shows that the model determines semantics of the text by considering how much of the content is positive, and how much is negative. But the first sample tells otherwise. Maybe 'very talented actors' was strong enough to consider this review as positive.

22698 is praising the movie being such a good B-movie, which is all about being weird and bad (on purpose.) This also got predicted value of 0.0014, while correct label is 1. I think the model is not that good at finding out sarcasm or euphemism. Thus, I could conclude that high-context language, such as Korean and Japanese might be much difficult to be modeled compared to low-context language, like Dutch.

Most Correct

Words like downright, slap in the face, bad, terrible, 1, worst, lamest, crappiest were included in predictions that were mostly correctly as 0. One thing to note was that all the top 10 correctly predicted words were all negative reviews. I think the fact that people tend to write negative sentences more harshly can also be connected to the fact that the model is better at detecting negative vibes than positive ones. And those well-recognized negative reviews are pretty long and hash.

Unknowns

Most proper nouns were converted as UNKNOWN. I think this is because proper nouns are not used frequently as other words. However, they were mostly just name of character/location that did not impose an affect in the overall semantic of the review. This model handled UNKNOWN well.

I was wondering how sentence with only UNKNOWN will be predicted. Desired result is 0.5, since it should be considered neutral. With only one ''UNKNOWN', `using your_text = 'asdfasdf ' * 1`, I got desired `tensor(0.499, ..)`. Then I tested a text with 100 unknown words. I used gibberish words like `your_text = 'jasfdnjknjvk ' * 100` and `your_text = 'asdfasddabkjn ' * 100`. Both values should be the same, since they should all be. UNKNOWN. Results were as expected, with `tensor(0.0797, grad_fn=<SqueezeBackward0>)`. I was surprised that that value was not neutral. Sentence became more negative if they were just long unknown words. Maybe some swear words were converted as unknown, or were masked during the formation of data.

I wanted to see if the model had a vast vocabulary, good enough to not have that much UNKOWNs when it processed other movie reviews. I used frequently used words data from Kaggle (https://www.kaggle.com/datasets/rtatman/english-word-frequency) and checked how many of them were unknown. I used first 20,000 most commonly used words, since an average english vocabulary for a person is about that much words.

This dataset is sorted according to frequency of usage, first word, 'the' being the most frequently used. I simply converted all 20,000 words into a list, converted it using str2idx, then converted it back with idx2string, and counted 'UNKNOWN', and divided it by length of the list. With code below:

```
import csv

def get_first_column_elements(file_path):
    with open(file_path, newline='', encoding='utf-8') as csvfile:
        reader = csv.reader(csvfile)
        # Skip the header row
        next(reader)
        first_column_elements = []
        for i, row in enumerate(reader):
            if i >= 20000:  # Stop after reading the first 20,000 rows
                break
            first_column_elements.append(row[0])  # Assumes the element you need is
in the first column
    return first_column_elements

# Example usage
file_path = 'unigram_freq.csv'
elements = get_first_column_elements(file_path)

converted_elements = converter(converter(elements))

print(converted_elements.count('UNKNOWN'))
print(converted_elements.count('UNKNOWN')/len(elements))

res = []
for i in range(len(elements)):
  if converted_elements[i] == 'UNKNOWN':
    res.append(elements[i])
print(res)
print(res[::-1])
```

6,973 out of 20,000 words were UNKNOWN. That is, 34.865%, or more than a third of the words were unknown. I chose to see the most frequent used UNKNOWN words and less used UNKNOWN words to see what kind of words were not included in the model.

`print(res)` returned words like `'shipping', 'directory', 'accessories', 'dec', 'server', 'blog', 'login', 'password', 'oct', 'browse', 'fax', 'rss', 'faq', 'feb', 'sep', 'url', 'aug', 'downloads', 'applications', 'apr', 'linux', …`. These words are either abbreviations, or tech-related terms, which are prevalent in modern day internet, thanks to lots of tech-related posts from sites like Stack Overflow and Github. So, it would not affect that much on movie-review assessing model.

`print(res[-1])`, which will be the least used words in human vocabulary that were not included in the model had words like `'bizjournalshire', 'yan', 'cpan', 'occ', 'householder', 'yugoslav', 'missionaries', 'routed', 'melanoma', 'originator', 'abatement', …`. It could be easily said that these words are not that significant in movie reviews, either.

The model did not know only 17.53% of words when 8,00 words were considered, and 21.04% of 10,000 words. And these words were still minor proper nouns, or tech-related terms. So this model will not face difficulty of having too much unknown due to not knowing much words.