SI 630: Homework 2 Part 4: Attention-based classification

This last part of homework 2 will have you *using* the vectors we learned from your word2vec implementation to do classification. You should complete the initial word2vec part before before starting on this.

Broadly, this last part of the homework consists of a few major steps:

1. Load in the data, word vectors, and word-indexing
2. Define the attention-based classification network
3. Train your model at least one epoch (2+ is recommended though).
4. Perform exploratory analyses on attention
5. Test the effects of freezing the pre-trained word vectors (see homework PDF for details)

After Step 2, you should be able to train your classifier implementation on a small percent of the dataset and verify that it's learning correctly. **Please note that this list is a general sketch and the homework PDF has the full list/description of to-dos and all your deliverables.**

Estimated performance times

We designed this homework to be run on a laptop-grade CPU, so no GPU is required. If your primary computing device is a tablet or similar device, this homework can also be *developed* on that device but then run on a more powerful machine in the Great Lakes cluster (for free). Such cases are the exception though. Following, we report on the estimated times from our reference implementation for longer-running or data-intensive pieces of the homework. Your timing may vary based on implementation design; major differences in time (e.g., 10x longer) usually point to a performance bug.

- Reading data, tokenizing, and converting to ids: ~20 seconds
- Training one epoch: ~18 minutes
- Training one epoch using frozen embeddings: ~3 minutes
- Evaluating on dev/test set: ~5 seconds

In [ ]:
```
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader

np.random.seed(42)
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import init
from tqdm.auto import tqdm, trange
from collections import Counter
import random
from torch import optim

import pandas as pd
import pickle

from torch.utils.tensorboard import SummaryWriter

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn.metrics import f1_score
import seaborn as sns

# Sort of smart tokenization
from nltk.tokenize import RegexpTokenizer

# Attention plotting
import matplotlib.pyplot as plt
```

In [ ]:
```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("Running on the GPU")
else:
    device = torch.device("cpu")
    print("Running on the CPU")
```

```
Running on the GPU
```

Load in the necessary parameters from the word2vec code

In [ ]:
```
# Load the word to index mapping we used for word2vec and use the same type
```

```python
# of tokenizer. We'll need to use this to tokenize in the same way and keep
# the same word-to-id mapping

import json

tokenizer = RegexpTokenizer(r'\w+')

def keystoint(x):
    return {int(k): v for k, v in x.items()}

with open("word_to_index.txt", 'r') as fp:
    word_to_index = json.load(fp)
with open("index_to_word.txt", 'r') as fp:
    index_to_word = json.load(fp, object_hook=keystoint)
```

Out[ ]:
```
'This'
```

Define the Classifier Model

Just like we did for word2vec, let's define a PyTorch `nn.Module` class here that will contain our classifier.

In [ ]:
```python
class DocumentAttentionClassifier(nn.Module):

    def __init__(self, vocab_size, embedding_size, num_heads, embeddings_fname):
        '''
        Creates the new classifier model. embeddings_fname is a string containing the
        filename with the saved pytorch parameters (the state dict) for the Embedding
        object that should be used to initialize this class's word Embedding parameters
        '''
        super(DocumentAttentionClassifier, self).__init__()

        # Save the input arguments to the state

        self.vocab_size = vocab_size
        self.embedding_size = embedding_size
        self.num_heads = num_heads


        # Create the Embedding object that will hold our word embeddings that we
        # learned in word2vec. This embedding object should have the same size
        # as what we learned before. However, we don't to start from scratch!
        # Once created, load the saved (word2vec-based) parameters into the object
        # using load_state_dict.


        self.target_embeddings = nn.Embedding(vocab_size, embedding_size)
        self.context_embeddings = nn.Embedding(vocab_size, embedding_size)

        pre_trained_dict = torch.load(embeddings_fname)

        self.load_state_dict(pre_trained_dict)


        # Define the attention heads. You have two options:
        #
        # 1) the worse way to implement this is to define your heads using an Embedding
        #    and then access them individually later in forward(). This will be slower
        #    but will probably still work
        #
        # 2) the ideal way is to think of your attention heads as rows in a matrix--
        #    just like we do for word2vec. While this is kind of the same as how
        #    we represent things like in an Embedding, the key difference is that we
        #    can now use **matrix operations** to calculate the different r and a
        #    vectors, which will be much faster (and less code). To do this, you'll
        #    need to represent the attention heads as a Tensor directly (not a layer)
        #    and make sure pytorch runs gradient descent on these parameters.
        #
        #  It's up to you which to use, but try option 2 first and see what you do
        #  in the forward() function

        self.head_matrix = torch.rand(embedding_size, num_heads) - 0.5

        self.softmax = nn.Softmax(1)


        # Define the layer that goes from the concatenated attention heads' outputs
        # to the single output value. We'll push this output value through the sigmoid
        # to get our prediction

        self.linear = nn.Linear(num_heads * embedding_size, 1)
        pass


    def forward(self, word_ids):

        # Pro Tip™: when implementing this forward pass, try playing around with pytorch
        # tensors in a jupyter notebook by making "fake" versions of them. For example:
        #
        # word_embeds = torch.Tensor([[1,6,2], [9,1,7]])
        #
        # If you have two word embeddings of length 3, how can you define the attention
        # heads to get the 'r' vector? Trying things out in the simple case will let you
        # quickly verify the sequence of operations you want to run, e.g., that you can take
        # the softmax of the 'r' vector to get the 'a' vector and it has the right shape
        # and values

        # Hint 1: If you're representing attention using Option 2, most of this code is just
        #         matrix multiplications

        # Hint 2: Most of your time is going to be spent figuring out shape errors and what
        #         operations you need to do to get the right outputs. This is normal.

        # Hint 3: This is the hardest part of this last part of the homework.


        # Get the word embeddings for the ids

        word_embeddings = self.target_embeddings(word_ids)

        attention_matrix = self.softmax(word_embeddings @ self.head_matrix.to(device))

        weight_average_document = torch.bmm(attention_matrix.transpose(1, 2), word_embeddings)

        result = self.linear(weight_average_document.view(-1, self.num_heads * self.embedding_size))

        return torch.sigmoid(result).view(-1), attention_matrix

        # Calcuate the 'r' vectors which are the dot product of each attention head
        # with each word embedding. You should be getting a tensor that has this
        # dot product back out---remember this vector is capturing how much the
        # head thinks the vector is relevant for the task


        # Calcuate the softmax of the 'r' vector, which call 'a'. This will give us
        # a probability distribution over the tokens for each head. Be sure to check
        # that the softmax is being calculated over the right axis/dimension of the
        # data (You should see probability values that sum to 1 for each head's
        # ratings across all the tokens)


        # Calculate the re-weighting of the word embeddings for each head's attention
        # weight and sum the reweighted sequence for each head into a single vector.
        # This should give you n_heads vectors that each have embedding_size length.
        # Note again that each head should give you a different weighting of the
        # input word embeddings


        # Create a single vector that has all n_heads' attention-weighted vectors
        # as one single vector. We need this one-long-vector shape so that we
        # can pass all these vectors as input into a layer.
        #
        # NOTE: if you're doing Option 2 for representing attention, you don't
        # actually need to create a new vector (which is very inefficient).
        # Instead, you can create a new *view* of the same data that reshapes the
        # different heads' vectors so it looks like one long vector.


        # Pass the side-by-side attention-weighted vectors through your linear
        # layer to get some output activation.
        #
        # NOTE: if you're feeling adventurous, try adding an extra layer here
        # which will allow you different attention-weighted vectors to interact
        # in making the model decision



        # Return the sigmoid of the output activation *and* the attention
        # weights for each head. We'll need these later for visualization
        pass
```

In [ ]:

```
x = torch.Tensor([[[1,6,2], [9,1,7]], [[4, 5, 7], [5, 9, 8]]])
y = torch.Tensor([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
```

```
s = nn.Softmax(1)
l = nn.Linear(12, 1)

torch.sigmoid(l(torch.bmm(s(x @ y).transpose(1, 2), x).view(-1, 12))).view(-1)
```

Out[ ]:

```
tensor([0.7357, 0.2031], grad_fn=<ViewBackward0>)
```

Load in the datasets

You can keep these as pandas data frames.

In [ ]:

```
sent_train_df = pd.read_csv('sentiment.train.csv')
sent_dev_df = pd.read_csv('sentiment.dev.csv')
sent_test_df = pd.read_csv('sentiment.test.csv')
```

Convert each dataset into a list of tuples of the form `([word-ids,...], label)`. Both the word ids and the label should be numpy arrays so they will get converted into Tensors by our data loader. Note that you did something very similar for creating the word2vec training data. This process will require tokenizing the data in the same way as you did for word2vec and using the same word-to-id mapping (both of which you loaded/created above).

In [ ]:

```
train_list = [(np.array([word_to_index[token] if token in word_to_index else word_to_index['<UNK>'] for token i
dev_list = [(np.array([word_to_index[token] if token in word_to_index else word_to_index['<UNK>'] for token in
test_list = [np.array([word_to_index[token] if token in word_to_index else word_to_index['<UNK>'] for token in
```

In [ ]:

```
print(test_list[0])
```

```
[  611   956   664     3    29  1221    12     7   299    21     0    21
     1  4412     4  7155    92    63  2699   260   449   301   229     7
   781   478    63   253   102   343    24     7   790   355  3329 30162
   449     7   299    21    61     4   110  6611    73     0    21  9401
  8387     7  3017   820     7   781    43   404   409   545   100   668
   105    63   478]
```

If this worked you should see XXXX train, XXXX dev, and XXX test instances.

In [ ]:

```
len(train_list), len(dev_list), len(test_list)
```

Out[ ]:

```
(160000, 20000, 20000)
```

Build the code training loop

We'll evaluate periodically so before we start training, let's define a function that takes in some evaluation data (e.g., the dev or test sets) and computes the F1 score on that data.

In [ ]:

```
def run_eval(model, eval_data):
    '''
    Scores the model on the evaluation data and returns the F1
    '''
    with torch.no_grad():

        labels = []
        predictions = []

        for data, label in eval_data:

            pred = model(data.to(device))[0].round().cpu().numpy()
            predictions.append(pred)
            labels.append(label.numpy())



        return f1_score(np.array(labels), np.array(predictions))

        pass
```

In [ ]:

```
np.array([np.array(1), np.array(1)])
```

Out[ ]:

```
array([1, 1])
```

Now that you have data in the right format and a neural network designed, it's time to train the network and see if it's all working. The training code will look surprisingly similar to your word2vec code.

For all steps, be sure to use the hyperparameters values described in the write-up.

1. Initialize your optimizer and loss function
2. Create your network
3. Load your dataset into PyTorch's `DataLoader` class, which will take care of batching and shuffling for us (yay!)
4. **see below:** Create a new `SummaryWriter` to periodically write our running-sum of the loss to a tensorboard
5. Train your model

For step 4, in addition to writing the loss, you should write the F1 score on the dev set to the writer as well, using the specified number of steps.

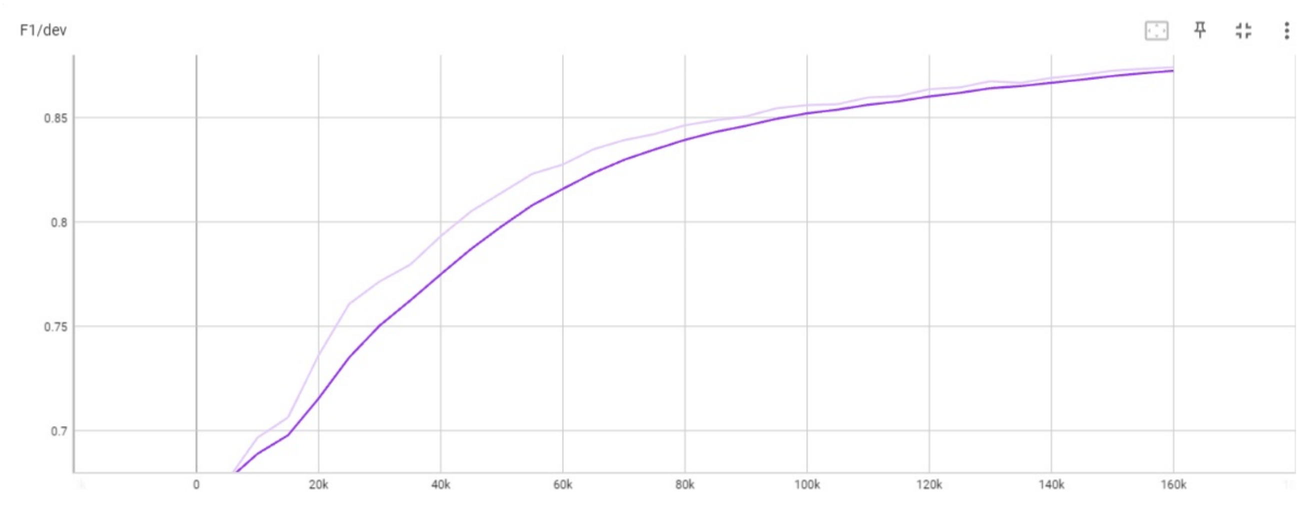
NOTE: In this training, you'll use a batch size of 1, which will make your life *much* simpler.

In [ ]:
```
# TODO: Set your training stuff, hyperparameters, models, tensorboard writer etc. here

model = DocumentAttentionClassifier(len(word_to_index), 50, 4, 'word2vec.pt').to(device)
train_loader = DataLoader(dataset = train_list, batch_size = 1, shuffle = True)
dev_loader = DataLoader(dataset = dev_list, batch_size = 1, shuffle = True)
writer = SummaryWriter()
criterion = torch.nn.BCELoss()
optimizer = torch.optim.AdamW(model.parameters(), lr = 5e-5)


pre_trained_dict = torch.load('word2vec.pt')

for (name, param) in model.named_parameters():
    if name in pre_trained_dict:
        param.requires_grad = True


# HINT: wrapping the epoch/step loops in nested tqdm calls is a great way
# to keep track of how fast things are and how much longer training will take

for epoch in trange(1):

    loss_sum = 0

    # TODO: use your DataLoader to iterate over the data
    for step, data in enumerate(tqdm(train_loader, desc = "Training")):

        # NOTE: since you created the data np.array instances,
        # these have now been converted to Tensor objects for us
        word_ids, label = data

        # TODO: Fill in all the training details here

        word_ids = word_ids.to(device)
        label = label.to(device)

        label = label.to(torch.float32)

        optimizer.zero_grad()
        prediction, attention_matrix = model(word_ids)

        loss = criterion(prediction, label)
        loss_sum += loss
        loss.backward()
        optimizer.step()

        # TODO: Based on the details in the Homework PDF, periodically
        # report the running-sum of the loss to tensorboard. Be sure
        # to reset the running sum after reporting it.

        if step % 500 == 499:
            writer.add_scalar("Loss/train", loss_sum, step + epoch * len(train_loader))
            loss_sum = 0

        if step % 5000 == 4999:
            f1 = run_eval(model, dev_loader)
            writer.add_scalar("F1/dev", f1, step + epoch * len(train_loader))


        # TODO: it can be helpful to add some early stopping here after
        # a fixed number of steps (e.g., if step > max_steps)



# once you finish training, it's good practice to switch to eval.
model.eval()
```

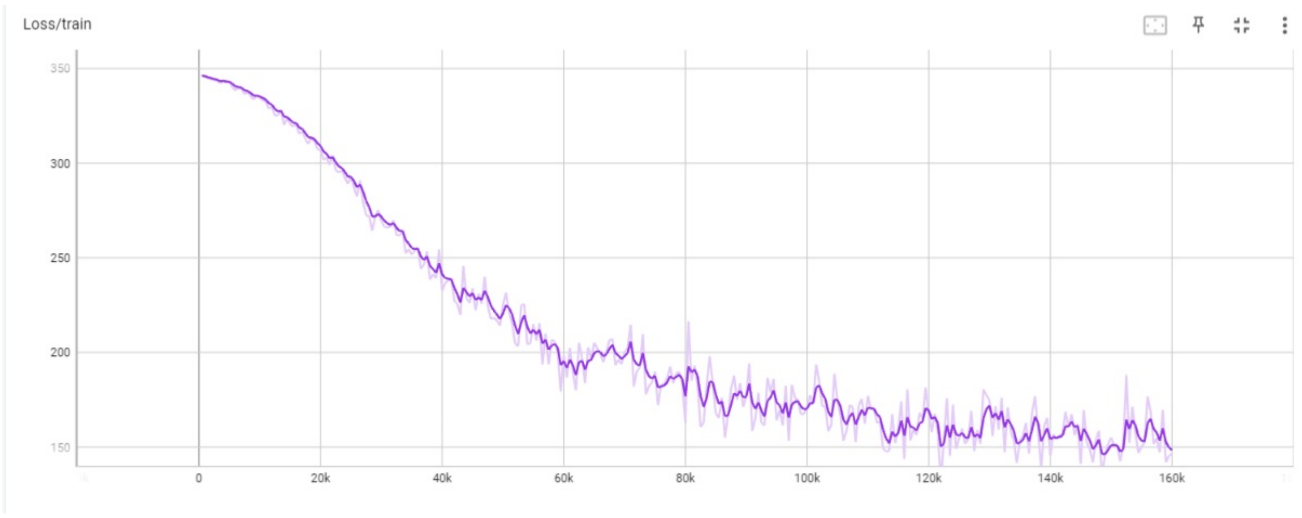
```
  0%|          | 0/1 [00:00<?, ?it/s]
Training:   0%|          | 0/160000 [00:00<?, ?it/s]
```

```
Out[ ]: DocumentAttentionClassifier(
          (target_embeddings): Embedding(35193, 50)
          (context_embeddings): Embedding(35193, 50)
          (softmax): Softmax(dim=1)
          (linear): Linear(in_features=200, out_features=1, bias=True)
        )
```

Problem 18

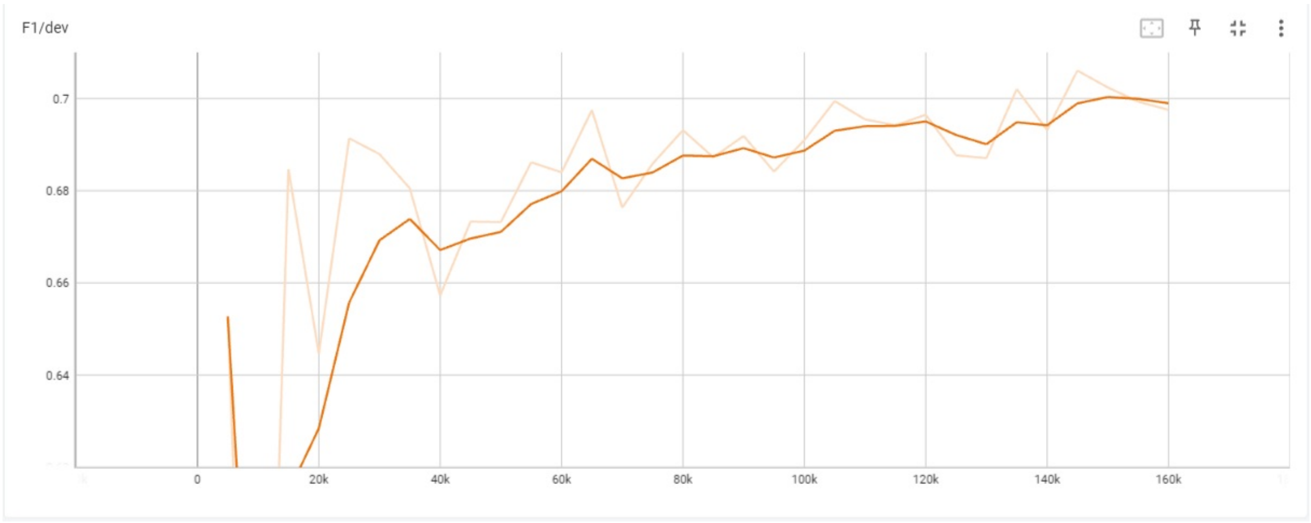


Plot for F1

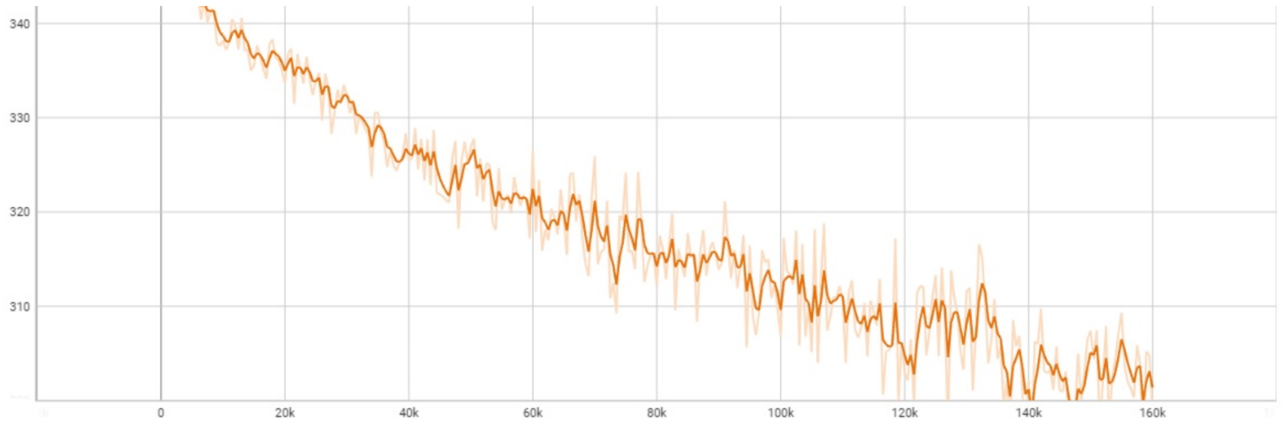


Plot for Loss

Problem 19



Plot for F1



Plot for loss

I think perhaps we'd better not freeze the word vectors in this setting. The word vectors might also be optimized during this classification training process.

Problem 20

```
In [ ]: test_loader = DataLoader(dataset = test_list, batch_size = 1, shuffle = False)

        predictions = []

        with torch.no_grad():
            for step, data in enumerate(tqdm(test_loader)):
                prediction, attention_matrix = model(data.to(device))

                predictions.append(1 if prediction > 0.5 else 0)
```

```
  0%|          | 0/20000 [00:00<?, ?it/s]
```

```
In [ ]: test_pred = np.array(predictions, dtype = int)
        result = pd.DataFrame({'prediction': test_pred}, columns = ['prediction'])
        result.to_csv('output.csv', index = True)
```

Kaggle username: Zhehong Wu

Inspecting what the model learned

In this last bit of the homework you should look at the model's attention weights. We've written a visualization helper function below that will plot the attention weights. You'll need to fill in the `get_label_and_weights` method that uses the model to classify some new text and structures the attention output in a way that's specified.

NOTE: most of the code for `get_label_and_weights` is code you've already written above.

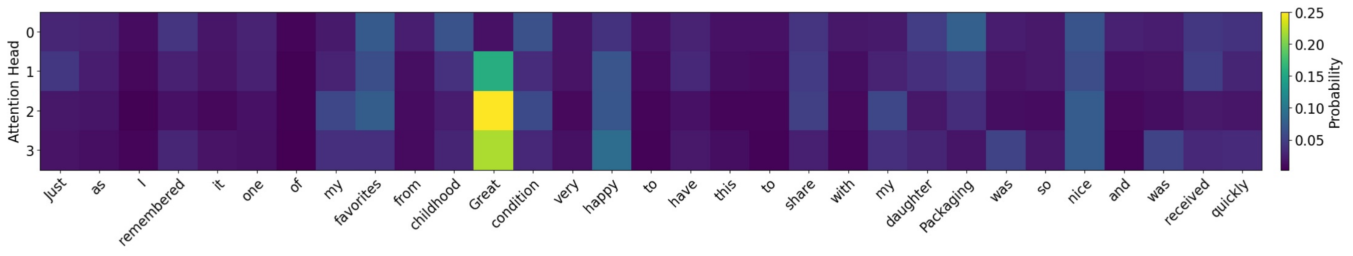
```
In [ ]: def get_label_and_weights(text):
            '''
            Classifies the text (requires tokenizing, etc.) and returns (1) the classification label,
            (2) the tokenized words in the model's vocabulary,
            and (3) the attention weights over the in-vocab tokens as a numpy array. Note that the
            attention weights will be a matrix, depending on how many heads were used in training.
            '''
            with torch.no_grad():

                tokens = tokenizer.tokenize(text)

                word_ids = [word_to_index[token] for token in tokens]

                input = torch.tensor(word_ids).unsqueeze(0).to(device)

                prediction, attention_matrix = model(input)

                result = 1 if prediction > 0.5 else 0

                return result, tokens, attention_matrix.cpu()[0]


                pass
```

Helper functions for visualization

```
In [ ]: def visualize_attention(words, attention_weights):
            '''
            Makes a heatmap figure that visualizes the attention weights for an item.
            Attention weights should be a numpy array that has the shape (num_words, num_heads)
            '''
            fig, ax = plt.subplots()
            # Rescale image size based on the input length
            fig.set_size_inches((len(words), 4))
            im = ax.imshow(attention_weights.T)

            head_labels = [ 'head-%d' % h for h in range(attention_weights.shape[1])]
            ax.set_xticks(np.arange(len(words))) # , labels=words)
            ax.set_yticks(np.arange(len(head_labels))) #, labels=head_labels)

            # Rotate the word labels and set their alignment.
            plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
                     rotation_mode="anchor")

            # Add the words and axis labels
            ax.set_yticklabels(labels=range(attention_weights.shape[1]), fontsize=16)
            ax.set_ylabel('Attention Head', fontsize=16)
            ax.set_xticklabels(labels=words, fontsize=16)

            # Add a color bar to show probability scaling
            cb = fig.colorbar(im, ax=ax, label='Probability', pad = 0.01)
            cb.ax.tick_params(labelsize=16)
            cb.set_label(label='Probability',size=16)
            fig.tight_layout()
            plt.show()
```

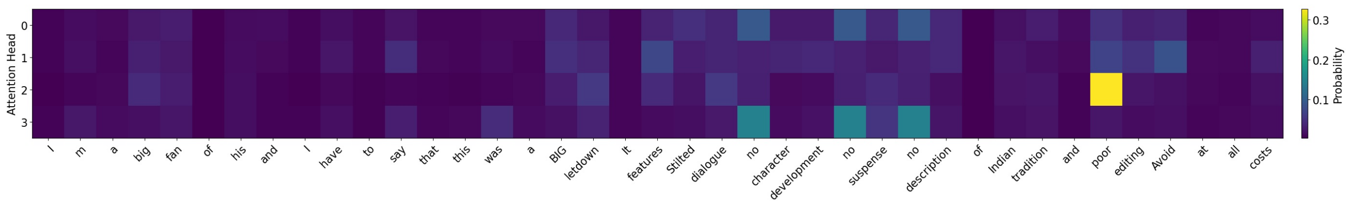
Example messages to try visualizing.

```
In [ ]: s = 'Just as I remembered it, one of my favorites from childhood! Great condition, very happy to have this to sl
        pred, tokens, attn = get_label_and_weights(s)

        visualize_attention(tokens, attn)
        print(pred)
```



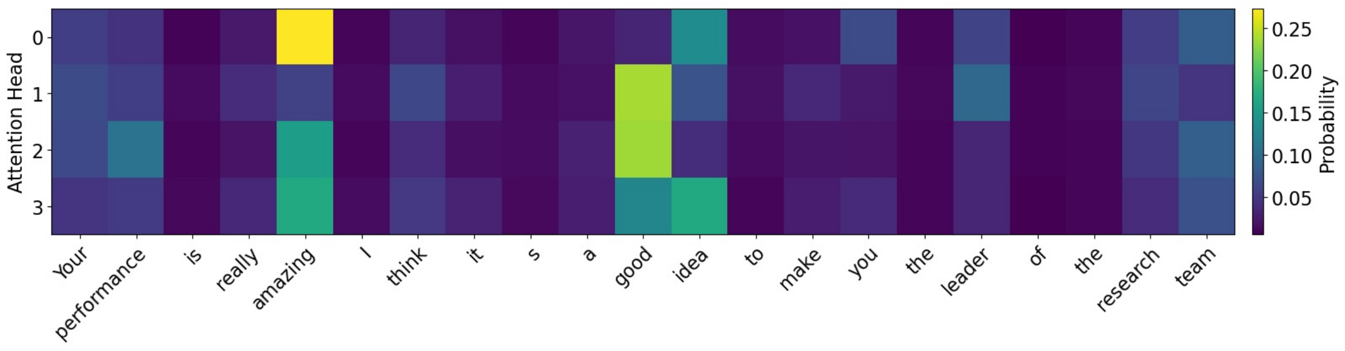
```
1
```

```
In [ ]: s = '''
        I'm a big fan of his, and I have to say that this was a BIG letdown. It features: Stilted dialogue, no characte
        '''
        pred, tokens, attn = get_label_and_weights(s)
        visualize_attention(tokens, attn)

        print(pred)
```



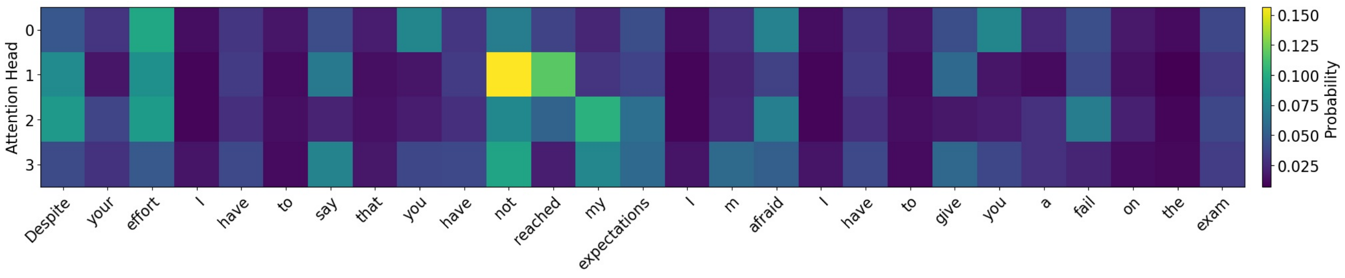
```
0
```

```
In [ ]: s = '''
        Your performance is really amazing. I think it's a good idea to make you the leader of the research team.
        '''
        pred, tokens, attn = get_label_and_weights(s)
        visualize_attention(tokens, attn)

        print(pred)
```



```
1
```

In [ ]:
```
s = '''
Despite your effort, I have to say that you have not reached my expectations. I'm afraid I have to give you a fa
'''
pred, tokens, attn = get_label_and_weights(s)
visualize_attention(tokens, attn)

print(pred)
```

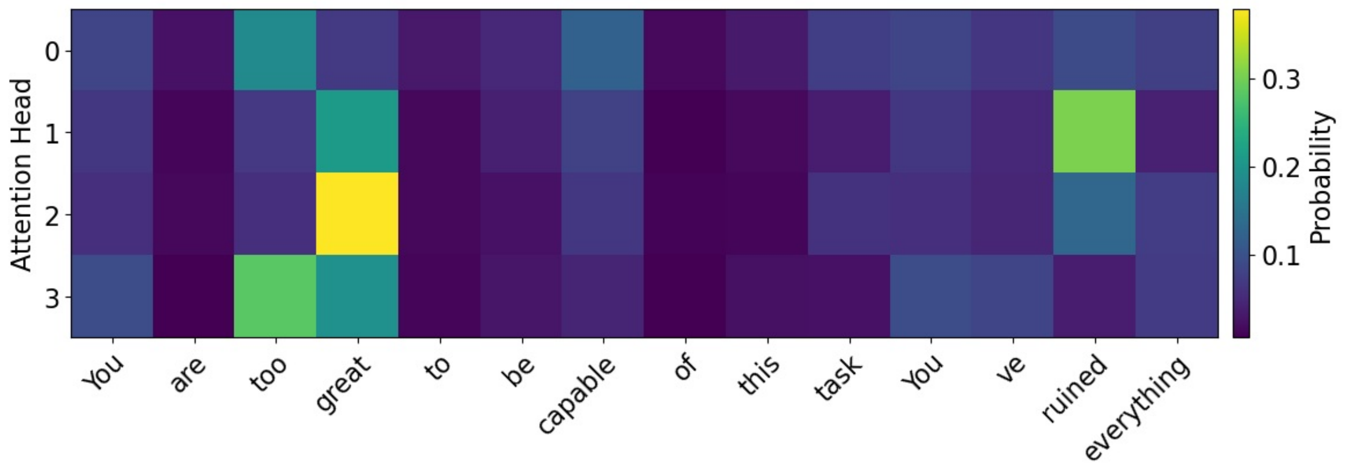


```
0
```

There exists a clear difference on the attention of different heads. It seems that two heads are focusing on some positive words like great and good, while the other two focuses more on some negative words. However, in terms of some neutral words like pronouns, all heads don't focus too much on them. For words like effort, some heads would give a little attention on it since it often appears in a positive sentence.

Problem 21

In [ ]:
```
s = '''
You are too great to be capable of this task. You've ruined everything.
'''
pred, tokens, attn = get_label_and_weights(s)
visualize_attention(tokens, attn)

print(pred)
```



```
1
```

It is easy to fool this model using an irony. The attention looks for great but couldn't realize that this is an irony.

Optional TODOs:

How many instances do we need to learn?

Since the word2vec vectors capture word meaning, do we need to see a lot of examples to train an effective classifier? Maybe we can get away with fewer (or not?). Try making a plot that shows the performance of training on 1 epoch with varying numbers of training examples. What if we just had 10 examples? 100? 1000?

Add more layers to the network

This is the second easiest one, but can be fun. What if you add more layers after you aggregate? Does letting the different attention heads' representations interact give better performance? Find out!

Change the learning rate dynamically

We have a fixed learning rate, but what if we wanted to decrease the learning rate as the model starts to converge? In many cases, this can help the model take smaller but more precise steps towards the best possible parameters. PyTorch supports this with *learning rate schedulers* that tell pytorch how and when to change the learning rate. See if you can get a better performance using a scheduler!

Add support for batch sizes > 1

This is non-trivial but will increase training speed *a lot*. The main issue with increasing batch sizes is that our input sequences (the word ids) in a batch will have different lengths. Under the hood, pytorch is turning your code into a series of very fast matrix operations. However, if those matrices suddenly have difference sizes, the math no longer works. As a result developers (like us) have to do a few things:

- We need to *pad* the sequences with empty values so that all sequences have the same length. You could do this by adding an extra word ID that is the "empty token" and make sure its values are set to 0 (so it won't interact with anything)
- Set up a collate function in our DataLoader that automatically pads each batch's data based on the longest length in the batch
- At inference time, it's also efficient to mask part of the sequence that's the padded part so we ignore the computations for that part in anything downstream. Depending on how you set it up, you may be able to avoid this step.

If you want to dig into this, you might see some of the documentation around packed and padded sequences in pytorch. You won't need to use these functions but they can provide more context for what's happening and why.

Add positional information to the word embeddings

Right now our model doesn't know much about which order the words are in. What if we helped the model in this? One way that people have done this is to *add* some positional embedding to the word embedding, where the positional embedding represents which position in the sequence is in. There are many complicated schemes for this, but one potential idea is to *learn the positional embeddings*. You would keep a separate `Embedding` object for positions with the number of positions up to the length of your longest sequence in the data. Then for the word in the first position, you add `position_embeddings(0)` to it. You can definitely speed that up by passing in a sequence of the positions in the current input and, conveniently, pytorch will let you easily add all the position embeddings to the word embeddings easily (no for loop required).

Will it help here? I have no idea but I'm curious.

No extra credit is given for these; they're just for folks who want to explore more

```
In [ ]:
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js