Malachi Eberly
Assignment 6

Part 1

To start off, I imported numpy and used np.argmax() on the tag_score to see which part of speech was decided upon. Once I had that, it was simple to format a print statement that printed the current word, the predicted part of speech, and the target part of speech. I then put this code into a test function that took in an input of the words and target parts of speech. Running this for both the sentences in the training data, I got the output seen on the right. This worked exactly as expected because the training set is so small.

```
0 The  DET  (DET)
1 dog  NN  (NN)
2 ate  V  (V)
3 the  DET  (DET)
4 apple  NN  (NN)

0 Everybody  NN  (NN)
1 read  V  (V)
2 that  DET  (DET)
3 book  NN  (NN)
```

```
0 Clifford  NN  (NN)
1 the  DET  (DET)
2 big  ADJ  (ADJ)
3 red  ADJ  (ADJ)
4 dog  NN  (NN)
5 ate  V  (V)
6 the  DET  (DET)
7 tasty  ADJ  (ADJ)
8 chicken  NN  (NN)
9 slowly  ADV  (ADV)

0 Everybody  NN  (NN)
1 read  V  (V)
2 that  DET  (DET)
3 crazy  ADJ  (ADJ)
4 book  NN  (NN)
5 very  ADV  (ADV)
6 quickly  ADV  (ADV)
```

I then made the examples more complex by adding in different parts of speech. I changed the two sentences to "Clifford the big red dog ate the tasty chicken slowly" and "Everybody read that crazy book very quickly". This introduced both adjectives and adverbs. The model worked just as well with this addition as can be seen on the left.

I tried creating two new sentences after training and seeing how the model performed on them. I used "the crazy dog quickly ate the red book" and "Clifford the big chicken ate the dog quickly". I didn't capitalize "the" because it wasn't capitalized in training. After testing these sentences, I found that they still performed almost perfectly, as can be seen on the right. The only one that didn't match up was "dog" in the second sentence, which was incorrectly labeled an adjective. This makes sense because the model was used to seeing an adjective following the word "the" before the noun came.

```
0 the  DET  (DET)
1 crazy  ADJ  (ADJ)
2 dog  NN  (NN)
3 slowly  ADV  (ADV)
4 ate  V  (V)
5 the  DET  (DET)
6 red  ADJ  (ADJ)
7 book  NN  (NN)

0 Clifford  NN  (NN)
1 the  DET  (DET)
2 big  ADJ  (ADJ)
3 chicken  NN  (NN)
4 ate  V  (V)
5 the  DET  (DET)
6 dog  ADJ  (NN)
7 quickly  ADV  (ADV)
```

```
0 The  ADJ  (DET)
1 crazy  ADV  (ADJ)
2 dog  NN  (NN)
3 chased  ADV  (V)
4 the  ADV  (DET)
5 big  ADV  (ADJ)
6 red  ADV  (ADJ)
7 chicken  NN  (NN)
```

Finally, I tried testing a sentence that included a word not trained with. I tested on "The crazy dog chased the big red chicken" with the new word being "chased". I also added "The" to the dictionary so I could use a capital T to start the sentence. This time the sentence was very poorly predicted, as seen on the left. The nouns were correctly predicted, but there were far too many incorrect adverbs and other mispredictions.

Part 2

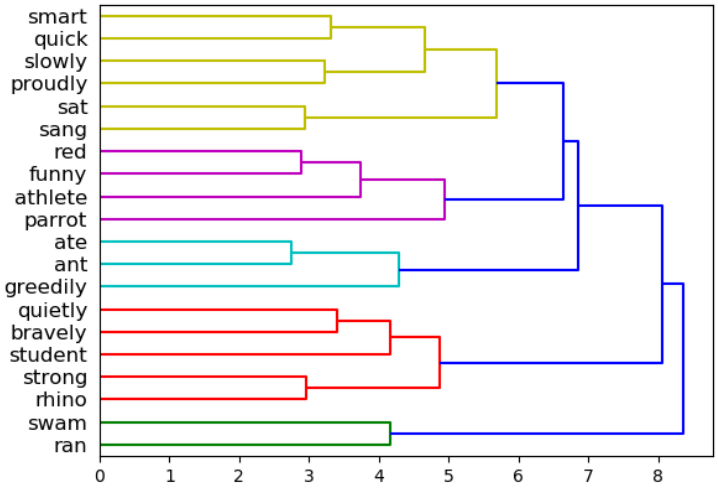
I added the print statement using numpy reformatting to show the words in alphabetical order next to their embedded vector value:

```
'This       [-0.433 -0.115  1.797 -0.478 -2.718 -0.224  0.342  1.109 -0.528  0.085]
And         [-0.386  0.547 -0.803  0.306 -2.032 -1.107  0.157 -0.598 -0.884  0.61 ]
And         [-0.386  0.547 -0.803  0.306 -2.032 -1.107  0.157 -0.598 -0.884  0.61 ]
How         [ 1.673  0.009  0.984  0.88  -1.451 -1.18  -0.461 -0.561  0.396 -0.984]
If          [-1.2   -0.048 -2.003 -0.491 -1.546 -0.173  0.729  0.058 -0.709 -0.527]
Proving     [-1.882 -0.776  2.025 -0.087  2.357 -1.037  1.576 -0.629  2.408  0.279]
Shall       [ 0.157  1.253  1.329 -0.496 -1.98   1.797  0.101  0.339 -0.645 -0.286]
Then        [ 0.406  0.342 -0.35  -0.234 -0.544  0.274 -0.502  1.735  1.139  0.735]
This        [-0.398 -1.93   0.929  0.913 -0.792 -0.584 -0.783 -1.423  1.609 -0.033]
Thy         [ 0.25   0.369 -0.51  -0.1    0.378  0.417 -0.754  0.793  0.076 -1.079]
To          [-0.015  1.041  0.645  0.8    0.41   0.301  1.382 -1.033  0.715 -1.564]
```

…

```
winters     [-1.295 -0.128  1.065 -2.353  1.166  0.171  0.529  2.889  0.16  -0.363]
within      [ 1.285 -1.228 -0.373  1.082 -1.243  0.813  0.091 -0.57   1.771  1.748]
worth       [ 0.772 -1.074 -0.203 -0.561 -0.622 -0.978  0.874  0.987  0.25  -0.793]
youth's     [-2.338 -0.83  -0.106 -1.179 -0.092  0.563 -0.587 -2.063  0.431  0.337]
```

I then changed the sentence to be a long string of sentences using a few different adverbs, adjectives, nouns, and verbs. By manipulating the code from the example to fit the new sentence and orienting/labeling the dendrogram, I got a pretty solid graph seen on the right. In terms of accuracy, it successfully paired some parts of speech together, but it struggled with a lot of it.



Part 3:

I started out by timing running transformer.py with the GPU and CPU to compare running times:

GPU	108.773 seconds
CPU	794.145

This shows just how much quicker the GPU is in this scenario, as the CPU is 630% slower.

When looking at the data used, I was expecting to see around 45,000 entries, since that was the statistic given for the approximate word count in one dictionary for the provided article. I printed the lengths of the training data, validation data, and the testing data:

len(train_data)	102,499
len(val_data)	21,441
len(test_data)	24,185

The length of the training data was much bigger than I anticipated, but this only confirmed that it was using words of the English language. After doing some more research, I found that there are around 170,000 active words in the language, so this number makes sense after counting in the words not used in the validation and test data sets. After accounting for those, the total number of words used comes to around 148,000, which is far greater than most humans' vocabulary. The "One Weird Trick" I see in the training report is the reporting of the ppl, or the perplexity of the language model. After each epoch and set of batches, the perplexity decreases. Perplexity is defined as the exponentiated average negative log-likelihood of a sequence. This is evaluated in the windows we defined. Here's the link for more info:
https://huggingface.co/docs/transformers/perplexity

Next I began to switch from one script to the three: transformer.py, transformer_train.py, and transformer_test.py. I saved the file best_model_params.pt in the local directory the same way I had done on Assignments 4 and 5. I did some research to find out how to find the total number of parameters that the model has, and here's the code I ran to check:

```
pytorch_total_params = sum(p.numel() for p in model.parameters())
print("Total number of parameters:", pytorch_total_params)
```

It returned a value of 12,025,582 parameters. This made sense seeing as the file was 52.1 megabytes, compared to a measly 31.8 kilobytes for mnist.pt on the past assignment.

In splitting up the files, first I successfully moved all the non-class code into the transformer_train.py, which initially could be run to both train and test the data. Once I confirmed this work, I began the difficult task of separating the training and testing. I struggled with this because of the overlapping variables, but eventually by putting a lot of the functions and global variables, I was able to get it working pretty smoothly. I then imported tools for building vocabulary from the iterator, and eventually I got the words printing out successfully.

Finally, I began to dig into the PyTorch source code to try to understand the WikiText-2 dataset and make sense of the individual lines. I downloaded the zip file and began scrolling through the source training and testing text, but quickly realized it was much more coherent and put together than the output was, using full sentences. After wondering a bit why the output was so convoluted, I saw a pattern. Looking at the last item in each line, I saw that the column made up the coherent speech. By going in order of the columns instead of left to right, I found a readable line coming directly from the source text, such as “had played any role in the creation of the character.” This explanation made sense after taking another look at the Jupyter notebook description, where it explained that the batchify function would turn the data into columns.