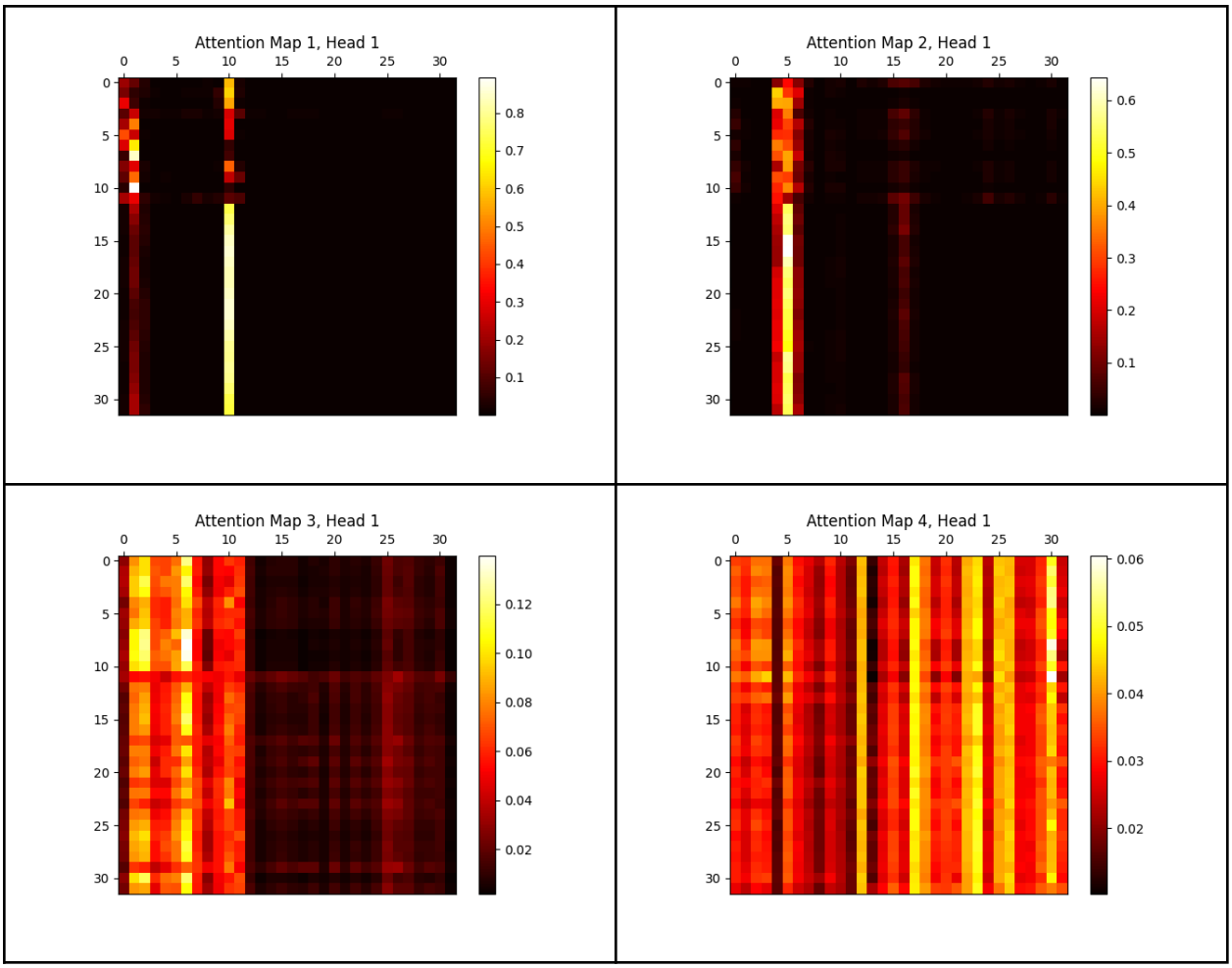
CSE PA2 Report

Encoder: For my encoder I first started by using a custom multi head attention class, Position Wise FeedForward class, and a Positional Encoding Class that I got from here https://www.datacamp.com/tutorial/building-a-transformer-with-py-torch . The multi head attention class implements multi head attention by setting up these linear layers query, value, key, and output transformation. Then computes attention scores and applies them to values and then reshapes tensor for parallel computation across multiple heads. Then I used the position wise feedforward network to be applied to each position separately. Lastly I used the positional encoding class to add positional information to the input embeddings allowing the model to understand the order of tokens in a sequence. I modified these functions to return the attn maps so that I could plot them. Then I created the encoder layer which combines multi head attention and the feed forward networks with layer normalization and dropout. I then created the transformer classifier class which is the main model class that integrates all components for the task. In this class I had an embedding layer, positional encoding, encoder layers, and a classifier layer. In the forward section, the tokenized sentences are first embedded and then positionally encoded. Then they are passed into the encoder stack where multiple encoder layers process the input and each layer applies self attention and feed forwarding. After that the output of the encoder is mean pooled. Using the value from the mean pooling, the classifier layer takes this input and makes a prediction. These were my results on the training set as well as my results on the test data.

```
Epoch [1/15], Loss: 1.1063, Accuracy: 41.92%
Epoch [2/15], Loss: 1.1793, Accuracy: 48.33%
Epoch [3/15], Loss: 1.0563, Accuracy: 53.78%
Epoch [4/15], Loss: 0.7484, Accuracy: 59.70%
Epoch [5/15], Loss: 0.8175, Accuracy: 67.30%
Epoch [6/15], Loss: 0.9499, Accuracy: 70.89%
Epoch [7/15], Loss: 1.1532, Accuracy: 77.77%
Epoch [8/15], Loss: 0.4498, Accuracy: 80.31%
Epoch [9/15], Loss: 0.4773, Accuracy: 84.03%
Epoch [10/15], Loss: 0.3236, Accuracy: 88.24%
Epoch [11/15], Loss: 0.1950, Accuracy: 87.38%
Epoch [12/15], Loss: 0.4231, Accuracy: 90.01%
Epoch [13/15], Loss: 0.1054, Accuracy: 91.20%
Epoch [14/15], Loss: 0.1595, Accuracy: 92.69%
Epoch [15/15], Loss: 0.1794, Accuracy: 93.12%
```

```
In [89]: compute_classifier_accuracy(model,test_loader)

Out[89]: 85.06666666666666
```



These were my attention maps for my first head. In attention map 1 I observed that there is a bright vertical line at position 10 indicating that tokens here are receiving a lot of attention while the rest of the map is dark meaning that in this context position 10 is important. Attention map 2 was very similar in that there is a bright vertical line at position 5, showing focus on early tokens. Attention map 3 illustrates a transition from bright to dark in early to later tokens yet again showing focus on earlier tokens. Lastly map 4 shows a more even distribution which may suggest that in this stage the head is capturing broader contextual information. A quick note, my sanity check didn't get the probabilities to sum to 0 but when I added a check in my function I observed that they were equal to one. A quick note, I had to edit the sanity check function to

display these attention maps as my encoder was outputting both heads.

```python
def attention(self, query, key, value, mask=None):
    attn_scores = torch.matmul(query, key.transpose(-2, -1)) / self.scale

    attn_scores = attn_scores.masked_fill(mask == 0, -1e9)

    attn_probs = torch.softmax(attn_scores, dim=-1)

    # Check if attention probabilities sum to one
    sum_check = torch.allclose(attn_probs.sum(dim=-1), torch.ones_like(attn_probs.sum(dim=-1)))
    if not sum_check:
        print("Warning: Attention probabilities do not sum to one!")
        print("Sum of probabilities:", attn_probs.sum(dim=-1))

    return torch.matmul(attn_probs, value), attn_probs
```

```
In [*]: #train
        # Loss and optimizer
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

        # Training loop
        for epoch in range(15):
            correct_predictions = 0
            total_samples = len(train_CLS_dataset)
            for xb, yb in train_loader:
                xb, yb = xb, yb
                optimizer.zero_grad()
                # Forward pass
                outputs = model(xb)
                loss = criterion(outputs[0], yb)
                # Backward pass and optimization
                loss.backward()
                optimizer.step()

            print(f'Epoch [{epoch+1}/{15}], Loss: {loss:.4f}')
```

```
tensor([[[1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
         [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000]],

        [[1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
         [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000]],

        [[1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
         [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000]],

        ...,

        [[1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
         [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000]],

        [[1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
         [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000]],

        [[1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000],
         [1.0000, 1.0000, 1.0000,  ..., 1.0000, 1.0000, 1.0000]]],
```

Lastly, I had 8 parameters in my encoder model illustrated here.

```python
class TransformerClassifier(nn.Module):
    def __init__(self, vocab_size, d_model, num_heads, d_ff, num_layers, num_classes, max_seq_length=512, dropout=0.1):
```

Decoder:

In the decoder I had the same code for multi headed attention, I also used the same architecture from the encoder and similar architecture from the encoder transformer. The main difference in the transformer decoder was that the output was from a linear layer not a classifier and in the forward method of the transformer Decoder I utilized a causal mask. In addition, in the forward method of the decoder I applied this mask to each layer. Finally, the outputs from

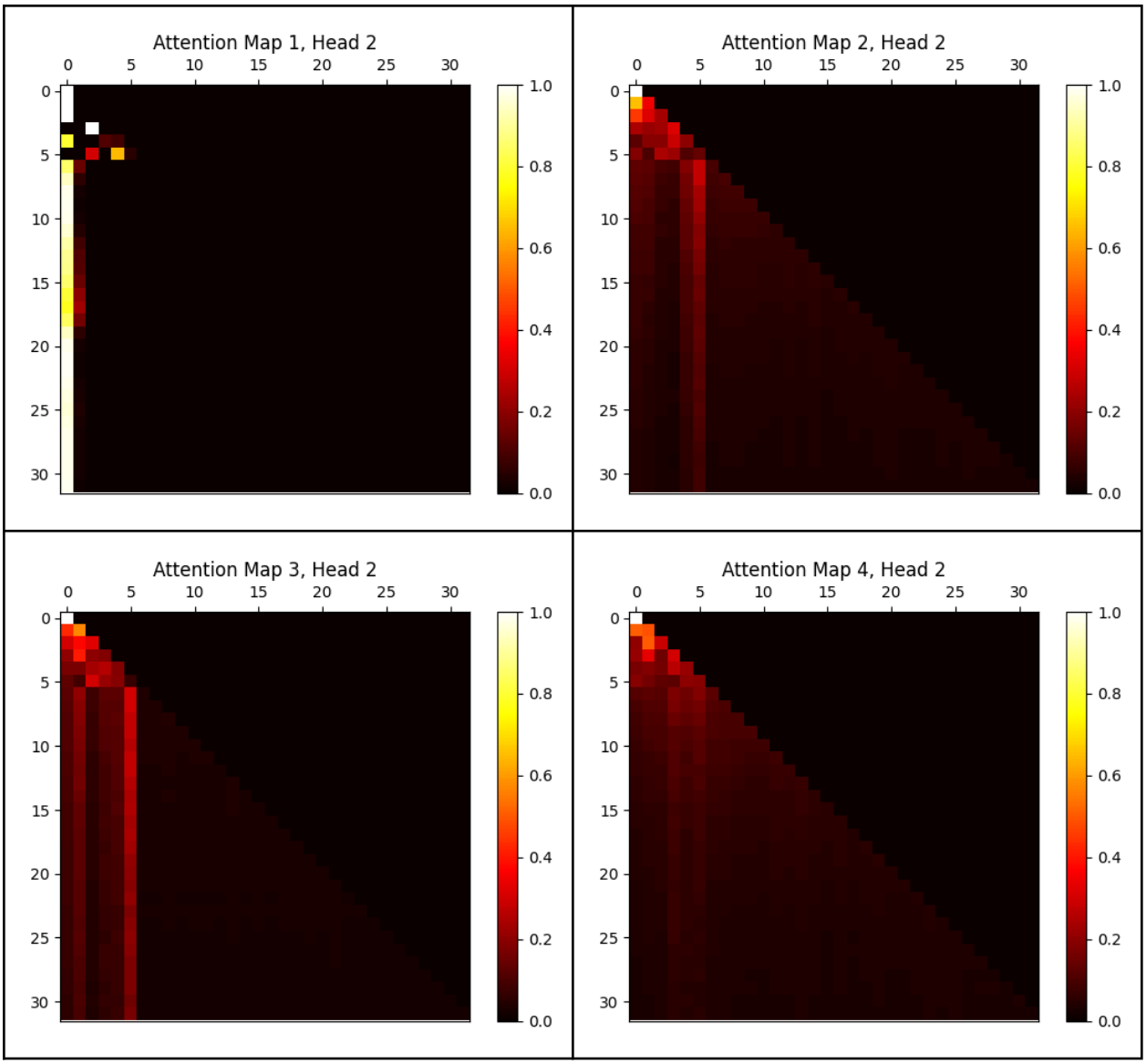
the decoder are passed through a linear layer and are converted into logits. This is how I trained the model. I also used two dropout layers in the forward method of my decoder layer and I used a dropout of 0.01.

```
# for the language modeling task, you will iterate over the training data for a fixed number of iterat
for i, (xb, yb) in enumerate(train_LM_loader):
    if i >= max_iters:
        break
    xb, yb = xb, yb

    optimizer.zero_grad()

    outputs = decoderModel(xb)

    loss = criterion(outputs[0].view(-1, vocab_size), yb.view(-1))

    loss.backward()
    optimizer.step()

    if i % 100 == 0:
        print(f"Iteration: {i}, Loss: {loss.item()}")
```

This was the loss I got from training and the perplexity on the training dataset as well as the results from the test data of different politicians.

```
Vocabulary size is 5755
Training Decoder model...
Iteration: 0, Loss: 8.85818099975586, Perplexity :6347.67626953125
Iteration: 100, Loss: 6.487338542938232, Perplexity :592.04296875
Iteration: 200, Loss: 6.047165393829346, Perplexity :436.2115783691406
Iteration: 300, Loss: 6.018743515014648, Perplexity :324.30560302734375
Iteration: 400, Loss: 5.757797718048096, Perplexity :251.2510223388672
Training Perplexity:  202.1020050048828
HBush:
Iteration: 0, Loss: 5.251516819000244, Perplexity :419.4127502441406
Iteration: 100, Loss: 5.214040756225586, Perplexity :409.65216064453125
Iteration: 200, Loss: 4.920149803161621, Final Perplexity :398.738037109375
Obama:
Iteration: 0, Loss: 5.069107532501221, Perplexity :355.6205139160156
Iteration: 100, Loss: 4.978749752044678, Perplexity :360.6820068359375
Iteration: 200, Loss: 4.874146938323975, Perplexity :368.5876159667969
WBush:
Iteration: 0, Loss: 4.787746429443359, Perplexity: 501.9560241699219
Iteration: 100, Loss: 4.4545488357543945, Perplexity: 508.09771728515625
Iteration: 200, Loss: 4.347790241241455, Final Perplexity: 546.963623046875

(base) C:\Users\William\OneDrive\Desktop\cse156\CSE156_PA2_FA24\PA2_code>
```

A reason for the varying complexity between Obama and both Bushes may be that both Bushes vary more compared to Obama's meaning that both Bushes may use different vocabulary as well as sentence structure while Obama may use similar vocabulary and sentence structure in speeches to the training data. In addition, both Bushes may talk more about specific issues during their time while Obama may have holistically addressed issues that encompass the training data better as a whole. Lastly, W Bush may be using very different language and may be talking about different topics compared to our training data.



These were my attention maps for my decoder. In attention map one, attention is highly concentrated vertically in position 0 meaning that in this attention head the attention is very focused on the beginning tokens. However in attention map 2,3,4 we can see a more diagonal token where tokens are attending to previous tokens in a more sequential manner. Lastly, we can see the mask being applied as all values above the diagonal are 0. These are the parameters are used for my decoder.

```
class TransformerDecoder(nn.Module):
    def __init__(self,vocab_size,d_model,num_heads,d_ff,num_layers,max_seq_length=512,
                 dropout=0.1):
```