CSE 256 - PA2 - Transformer Blocks

Due: *Wed, Nov 6th*

Answered by: Mathias Otnes

1 Encoder Trainer With Classifier

For this part, we're asked to implement an transformer encoder connected to a feed forward classifier which is going to classify which politician a given speech segment is from. The detailed implementation can be viewed in the attached code, and will not be described further in this report.

To sanity check the classifier, we're provided a utility function for visualizing attention maps and to verify the probability distributions internally created in the transformer blocks. Two attention maps for the sentence "*And that is why I intend to personally pursue this outcome with all the patience and dedication that the task requires.*" are visualized in figure [1] and [2].

Figure [1] shows one of 2 attention heads in the first transformer block of the encoder. The first layer will typically catch positional relationships and grammar for close words. We can for example see that word 20 attends to word 19 which corresponds to "*requires*" attending to "*task*", which makes sense because "*requires*" is a verb connected to the subject which is "*task*".

The downstream layers typically catches more semantic and abstract concepts. As we can see in the attention map for the fourth transformer block layer, there are many vertical lines. These vertical lines indicates important words for our classification problem, and are likelly words that are frequently used by one of our three politicians. We can also see that the weight of elements that's off-diagonal are a bit bigger relative to the early-layer attention map, which indicates that the attention in the fourth layer catches longer range relationships between words. This is typical for more semantical relationships.

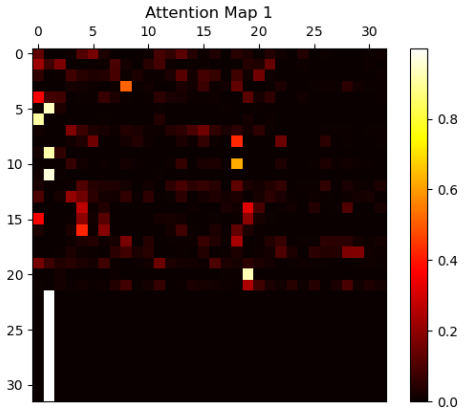

Figure 1: Attention map for head 1 in transformer block 1 in the classifier model.

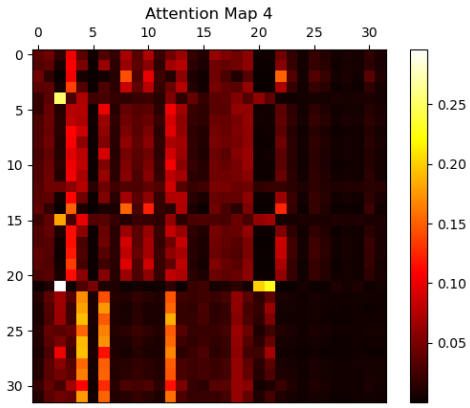

Figure 2: Attention map for head 1 in transformer block 4 in the classifier model.

We've gotten the dataset and hyperparameters for training provided in the handed out code. The val-

idation loss for 15 epochs of training using these values combined with my implementation can be seen in figure [3]. This matches the expected result of being in the low to mid 80s. My final accuracy was 84.52%.

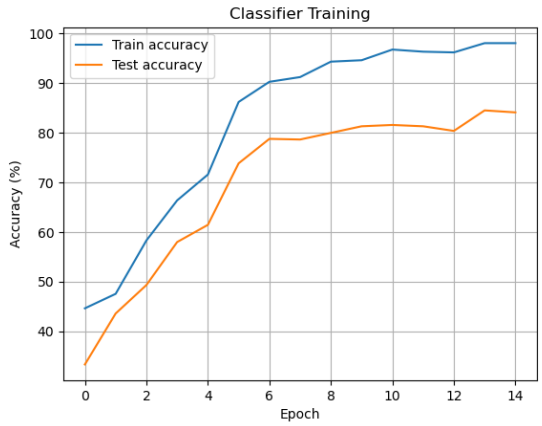

Figure 3: Training and validation loss for classifier model.

To determine the number of trainable parameters in the encoder, I utilized PyTorch's functionality to sum over all parameters that require gradients:

```
total_params = sum(
    p.numel()
    for p in self.parameters()
    if p.requires_grad
)
```

This code iterates over all trainable parameters and sums the total number of elements in each parameter tensor (`p.numel()`). This provides an accurate count of all trainable parameters without manual calculation. My encoder model has 789,184 parameters which is partitioned as follows:

- Embedding Layer: 368,320
- Positional Encoding: 320,000
- Transformer Blocks: $25,216 \times 4$

2 Pretraining Decoder Language Model

In this task, we're asked to implement a decoder-only language model. This is very similar to the encoder model, with the exception that we're training to predict the next token in a sequence. To do this, we need to mask parts of the sentence to ensure that any given token don't attend to tokens that appears later in a sequence, because we don't have access to future tokens in auto-regressive generation. The detailed implementation can be seen in the attached code.

I'm using the same sanity check utility as when working with the encoder, and figure [4] and [5] shows the attention maps for layer 1 and 4 respectively for one head in the decoder trained on the Obama training set during inference of the speech segment; "*America, we weaken those ties when we allow our political dialogue to become so corrosive that people of good character aren't even willing to enter into public service; so coarse with rancor that Americans with whom we disagree are seen not just as misguided, but as malevolent.*" from the Obama training set.

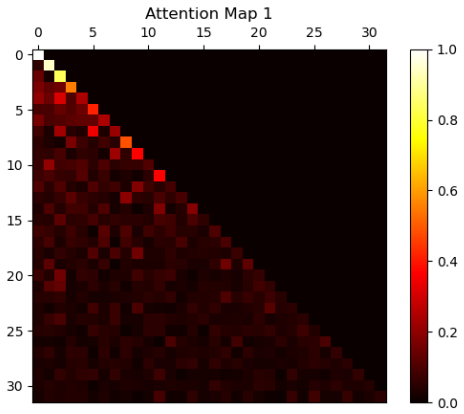

Figure 4: Attention map for head 1 in transformer block 1 in the decoder language model.

From the first attention map [4] we can see similar

characteristics as with the encoder model. Nearby words tend to attend to each other. For example "*weaken*" attends to "*ties*", which is reasonable because "*weaken*" is a transitive verb acting on the direct object "*ties*". The thing that seperates the decoder from the encoder model is that words don't attend to other words that shows up later in the sentence. For example the first word "*America*" only attends to itself. This is because we're using masked multi-head attention instead of multi-head attention.

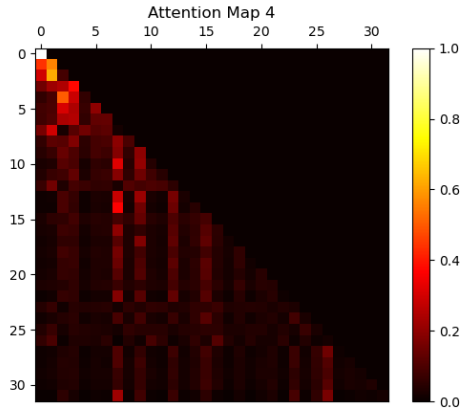

Figure 5: Attention map for head 1 in transformer block 4 in the decoder language model.

From the second attention map [5] which corresponds to the last transformer block layer in the decoder, we see that many of the words that are very context dependent attend to many of the other words. This can be seen by the vertical lines. Example of these words are "*we*", "*our*" and "*to*", which don't provide much information without context. This indicates that this later layer catches more complex semantics and relationships.

The training perplexities can be viewed in table [1]. These are roughly in the same ballpark as what is to be expected by the assignment description. The perplexity is a measurement of accurately the model predicts the next token in a given dataset, where lower perplexity means higher accuracy.

Dataset / Iteration	**100**	**200**	**300**	**400**	**500**
train_LM.txt	585	434	307	224	169
test_LM_obama.txt	527	357	361	290	221
test_LM_hbush.txt	534	388	341	288	242
test_LM_wbush.txt	573	389	362	313	272

Table 1: Perplexity scores across datasets and iterations

There are several reasons why the *train_LM.txt* dataset yields lower perplexity than the others. Primarily, it's likely due to the language being more easily predictable and the dataset's larger size—approximately $28,000$ tokens compared to the smaller presidential datasets, which range from $4,000$ to $5,000$ tokens. This larger volume gives the model more exposure to various token combinations, enhancing its ability to generalize effectively.

It's also noteworthy that the Obama dataset resulted in better perplexity scores than those of H. Bush and W. Bush. This could be attributed to differences in rhetorical styles. Perhaps Obama speaks in a more predictable manner, while W. Bush may rely more on unexpected elements in his rhetoric. Additionally, the contexts of their presidencies might have influenced how they crafted their speeches. For instance, some of W. Bush's speeches may refer to events like 9/11, but since this context isn't directly provided to the decoder model, it might struggle with predicting such references.

Finally, to report the number of parameters in my decode model, I used the same aproach as with the encoder. It consists of $1,181,963$ parameters:

- Embedding Layer: $368,320$
- Positional Encoding: $320,000$
- Transformer Blocks: $25,216 \times 4$
- Linear Head: $374,075$
- Layer Norm: 128

3 Architectural Exploration

In this part of the project, I explored the Disentangled Self-Attention mechanism introduced by He et al. [1]. To implement this, I closely followed the algorithm presented in their original paper as well as their official implementation provided on GitHub [2].

The key difference between the standard Transformer decoder and the one with disentangled attention lies in the attention mechanism itself. Instead of using the conventional multi-head attention, the Disentangled Self-Attention incorporates relative positional embeddings and disentangles content and positional information. This approach allows the model to better capture the relationships between tokens based on both their content and their relative positions.

The implementation details are provided in the attached file *disentangled_attention.py*. The decoder maintains the same hyperparameters and architecture as the original, with the following exceptions:

- **Disentangled Transformer Blocks:** We replace the standard Transformer blocks with Disentangled Transformer blocks.
- **Positional Encoding Layer Removed:** Since the relative positional encodings are integrated into the disentangled attention mechanism, we exclude the separate positional encoding layer.

The results obtained from training the decoder with the Disentangled Self-Attention mechanism are presented in table [2]:

Dataset / Iter	**100**	**200**	**300**	**400**	**500**
train_LM.txt	506	292	189	135	99

Table 2: Perplexity scores across datasets and iterations using disentangled attention

Comparatively, the original decoder achieved a perplexity of 169 after the same number of iterations with identical hyperparameters. This indicates that the decoder with disentangled attention reaches a lower perplexity faster.

Additionally, the disentangled decoder has slightly fewer parameters than the original, standing at $1,157,387$ parameters:

- Embedding Layer: $368,320$
- Linear Head: $374,075$
- Layer Norm: 128
- Transformer Blocks: $103,716 \times 4$

The improved performance of the decoder with Disentangled Self-Attention can be attributed to several factors:

Relative Positional Embeddings:
The disentangled attention mechanism incorporates relative positional embeddings directly into the attention computation. This allows the model to consider the relative distances between tokens, enabling it to generalize better to sequences of varying lengths and capture long-range dependencies more effectively.

Disentangling Content and Position Information:
By separating content and positional information in the attention mechanism, the model can focus on learning meaningful representations of the tokens themselves while appropriately accounting for their positions in the sequence. This disentanglement can reduce confusion between token identity and position, leading to more accurate attention distributions.

Efficient Parameter Utilization:
Despite having slightly fewer parameters, the model achieves better performance. This suggests that the Disentangled Self-Attention mechanism makes more efficient use of its parameters by focusing on crucial aspects of the data, such as relative positions and content relevance.

Elimination of Absolute Positional Embeddings:
Removing the separate positional embedding layer

reduces the risk of overfitting to specific positions in the training data. Relative positions are more robust to shifts and variations in the input sequences. Figure [6] shows the attention maps, in which it's clear that words only attend to words that are relatively close.

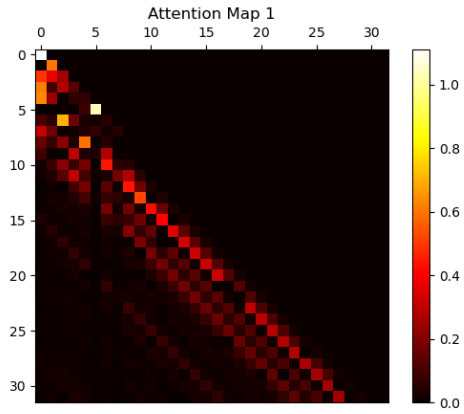

Figure 6: Attention map for head 1 in transformer block 1 in the disentangled attention decoder language model.

In conclusion, the experiment demonstrates that incorporating the Disentangled Self-Attention mechanism leads to a notable improvement in language modeling performance. The decoder achieves a lower perplexity compared to the original model with standard attention, indicating better predictive capabilities. Because the model also has less parameters, we know that it utilizes these parameters better than the normal transformer decoder.

References

[1] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention, 2021.

[2] Pengcheng He. *Disentangled Attention Implementation*, n.d. Accessed on 30 October 2024.