CSE 256 PA2 Report

Sanidhya Singal (ssingal@ucsd.edu)
Department of Computer Science and Engineering
University of California San Diego

Spring 2024

Part 1: Classification

Here, I implement a transformer encoder and train it jointly from scratch with a feedforward classifier for a downstream task of predicting which politician delivered a given speech segment.

Part 1.1: Encoder Implementation

Here, I describe the architecture of the Encoder in a sequential order:

- Token embedding table: An embedding layer that generates a 64-D embedding vector for each token in the vocabulary. There are 5,755 tokens in the vocabulary.
- Position embedding table: An embedding layer that generates a 64-D embedding vector for each position in the input context. The maximum context length is 32 (block size).
- 4 Transformer Blocks, where each Block of consists of the following:
  - Multi-head attention layer: This layer consists of the following:
    - 2 Attention heads: Each head has 3 weight matrices each of size 64×32 denoting the weights corresponding to key K, query Q, and value V vectors. Each head essentially computes the following:

$$\text{self-attention} = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

    - Projection layer: A linear layer that maps the output of self-attention to a 64-D vector by first concatenating the output vectors from the 2 attention heads, and then, applying a weight matrix of size 64×64.
  - Feed forward network: A two layer network which maps the 64-D input vector to a 256-D hidden space and back to 64-D output vector after applying the ReLU activation function.
  - 2 LayerNorm layers that perform layer normalization on the 64-D vector.
  - Residual connections that add the output vector to the original input vector.

For low-level implementation details, please refer to `README.md`.

Part 1.2: Feedforward Classifier Implementation

Here, I describe the architecture of the Classifier in a sequential order:

- Encoder: A transformer Encoder as described in previous section.
- 2 Linear layers: The first layer maps the 64-D input vector to a 100-D hidden vector and applies the ReLU activation function. The second layer then maps the 100-D hidden vector to a 3-D output vector, which represents the probabilities corresponding to each of the 3 politicians.

For low-level implementation details, please refer to `README.md`.

Part 1.3: Classifier Training

Using the CrossEntropy loss as the criterion and AdamW (with a learning rate of 10^{-3}) as the optimizer, I train and evaluate the classifier for 15 epochs, noting down the train loss, train accuracy, and test accuracy at each epoch. This data is stored in `part1_classification_task.json` file. For more details, please refer to `README.md`.

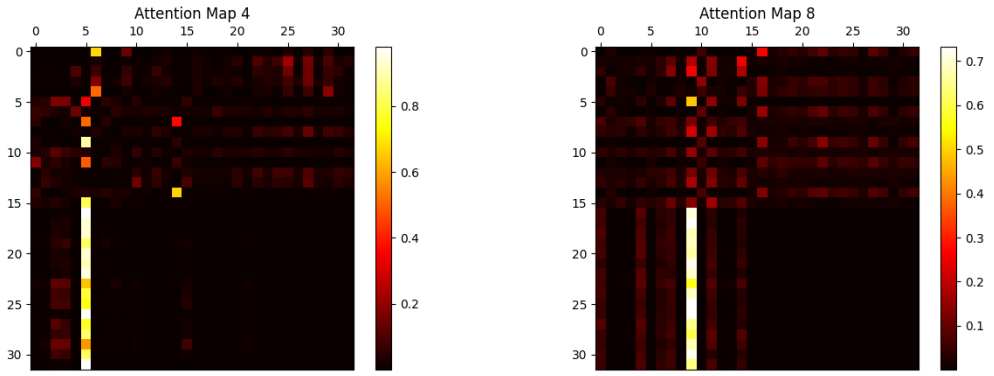
Part 1.4: Sanity Checks

I perform sanity checks using the following sentences, as seen in the training data:

> Sentence 1: "This is an immensely important day, a day that belongs to all of you".
>
> Sentence 2: "None of these changes happened overnight".

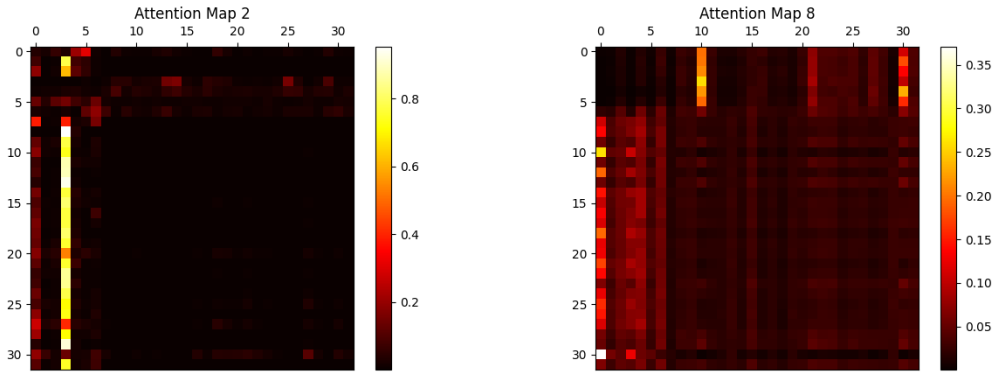
Based on the assertion statements, my output passes the sanity checks. Please refer to figures 1 and 2 for the corresponding attention matrices. I make the following **inferences**:

1. The attention maps vary across layers and across heads, which implies that each head is attending to different tokens in the input sequence.
2. There is no masking in the attention maps indicating an encoder-like architecture where the current query token can attend to keys in the past as well as future.



(a) Attention Map for 2nd Head of Layer 2 (b) Attention Map for 2nd Head of Layer 4

Figure 1: Comparison of Attention Maps for Sentence 1



(a) Attention Map for 2nd Head of Layer 1 (b) Attention Map for 2nd Head of Layer 4

Figure 2: Comparison of Attention Maps for Sentence 2

3. The attention maps in the starting layers seem to focus more on 1-2 tokens, whereas those in the later layers seem to focus on relatively larger number of tokens. This looks reasonable as the later layers combine the information from earlier layers and therefore can attend to a larger number of tokens in the input sequence.

Part 1.5: Evaluation

Please refer to figure 3. It mentions the train_loss, train_accuracy, and test_accuracy for each epoch. Also, note that the classifier model has 586, 947 parameters along with a vocabulary of 5, 755 tokens.

epoch	train_loss	train_accuracy	test_accuracy
1	1.082	45.6979	34.6667
2	1.062	51.0038	43.8667
3	0.9969	62.9541	56.8
4	0.888	69.6463	61.4667
5	0.7505	80.5449	70.6667
6	0.5763	84.2256	73.8667
7	0.4264	93.5946	80.8
8	0.2721	95.9369	83.3333
9	0.2004	96.8929	84.2667
10	0.1207	98.1836	83.8667
11	0.0603	99.044	85.4667
12	0.1271	97.4665	83.7333
13	0.0819	98.4226	85.8667
14	0.0389	99.5698	86.2667
15	0.019	99.6654	86.0

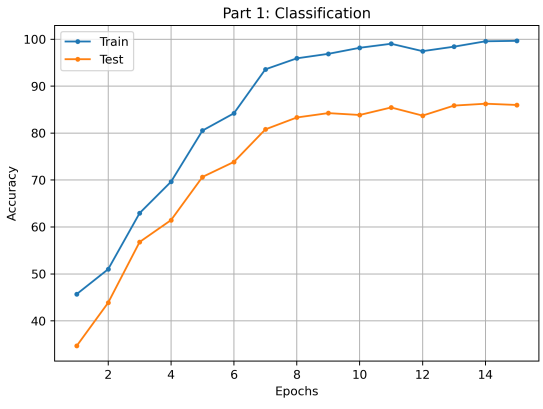


Figure 3: Evaluation of the classification task: train_accuracy vs test_accuracy for each epoch

I make the following **inferences**:

1. Both the training loss and accuracy improve significantly over the epochs. The training loss decreases from 1.082 in epoch 1 to 0.019 in epoch 15, while the training accuracy increases from 45.6979% to 99.6654%. This indicates that the model is learning effectively and fitting the training data increasingly well.
2. The test accuracy also shows a substantial increase from 34.6667% in epoch 1 to 86.2667% in epoch 14, suggesting that the model's performance on unseen data is improving over time. However, there is a slight fluctuation in test accuracy in later epochs, indicating possible overfitting after epoch 11, where the training accuracy is near perfect, but test accuracy gains are minimal or slightly decrease.

Part 2: Language Modeling

Here, I implement a word-level, GPT-like transformer decoder, pretrain it on an autoregressive language modeling task, and report perplexity numbers on speeches from different politicians.

Part 2.1: Decoder Implementation

Here, I describe the architecture of the Decoder in a sequential order:

- Similar to the Encoder, the Decoder consists of 2 embedding layers and 4 layers of transformer Blocks, along with a final LayerNorm layer, and a Linear layer. Please refer to Part 1.1 for more details.
- A final LayerNorm layer that perform layer normalization on the 64-D vector.
- LM Head layer: A linear layer that maps the 64-D input vector to 5, 755-D output vector, which represents the probabilities corresponding to each token in the vocabulary.

For low-level implementation details, please refer to `README.md`.

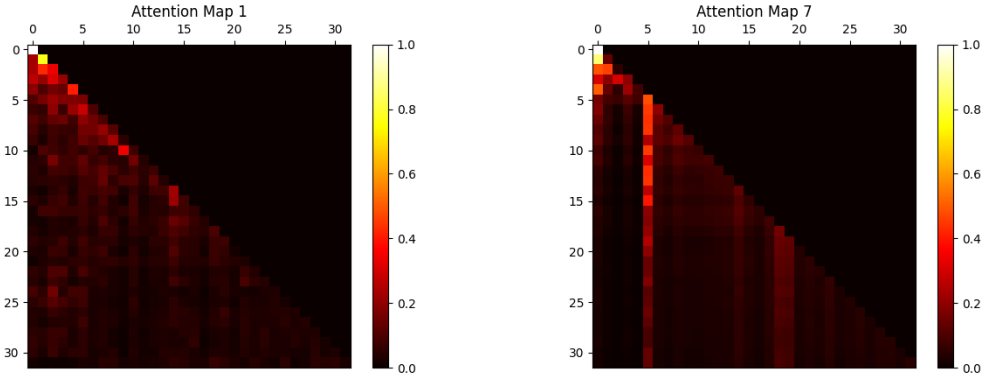
Part 2.2: Decoder Pretraining

Using AdamW (with a learning rate of 10^{-3}) as the optimizer, I train and evaluate the decoder for 500 epochs, noting down the train perplexity, hbush test perplexity, obama test perplexity and wbush test perplexity after every 100 epochs. This data is stored in `part2_language_modeling_task.json` file. For more details, please refer to `README.md`.

Part 2.3: Sanity Checks

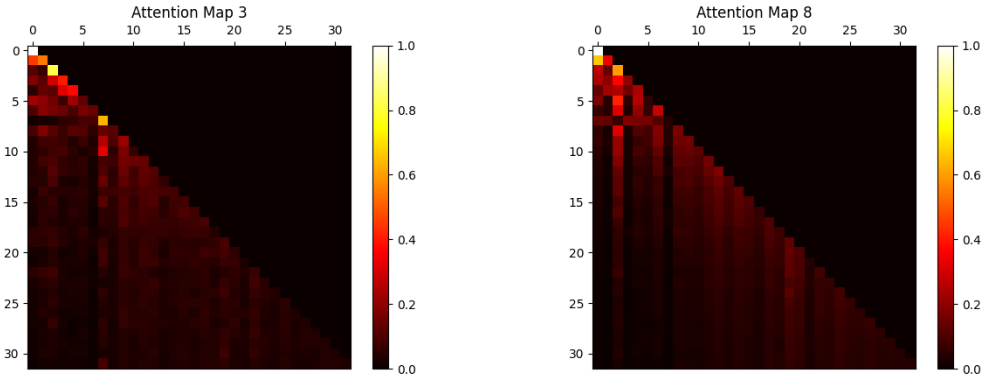
I perform sanity checks using the following sentences, as seen in the training data:

> Sentence 1: “Our relations abroad were strained”.
>
> Sentence 2: “America, we cannot turn back”.



(a) Attention Map for 1st Head of Layer 1 (b) Attention Map for 1st Head of Layer 4

Figure 4: Comparison of Attention Maps for Sentence 1



(a) Attention Map for 1st Head of Layer 2 (b) Attention Map for 2nd Head of Layer 4

Figure 5: Comparison of Attention Maps for Sentence 2

Based on the assertion statements, my output passes the sanity checks. Please refer to figures 4 and 5 for the corresponding attention matrices. I make the following **inferences**:

1. The attention maps vary across layers and across heads, which implies that each head is attending to different tokens in the input sequence.
2. The masking in the attention maps indicates a decoder-like architecture where the current query token can attend to keys only in the past.

3. The attention maps in the starting layers seem to focus more on 1-2 tokens, whereas those in the later layers seem to focus on relatively larger number of tokens. This looks reasonable as the later layers combine the information from earlier layers and therefore can attend to a larger number of tokens in the input sequence.

Part 2.4: Evaluation

Please refer to figure 6 and table 2. It mentions the train_perplexity, hbush_test_perplexity, obama_test_perplexity and wbush_test_perplexity after every 100 epochs. Furthermore, the decoder model has 943, 739 parameters along with a vocabulary of 5, 755 tokens.

Table 2: Evaluation of the language modeling task: train_perplexity vs test_perplexity for each epoch

epoch	train_perplexity	hbush_test_perplexity	obama_test_perplexity	wbush_test_perplexity
100	572.1758	712.5004	682.0963	798.6691
200	388.5926	554.0688	516.1316	622.2266
300	271.3539	460.7576	429.7112	534.091
400	196.3669	417.7756	369.7496	497.4196
500	149.8655	412.154	358.34	465.4696

I make the following **inferences**:

1. The training perplexity consistently decreases as the number of epochs increases. This indicates that the model is learning and improving its performance on the training data over time. Starting from a perplexity of 572.1758 at 100 epochs, it reduces to 149.8655 by 500 epochs.
2. While the perplexities on the test sets for George H.W. Bush (hbush), Barack Obama (obama), and George W. Bush (wbush) also decrease over epochs, the rates of improvement vary across these test sets. The perplexity values for Obama's speeches tend to be lower compared to those for the Bushes, particularly George W. Bush, suggesting that the model generalizes better to Obama's language style or content. For instance, at 500 epochs, the perplexity for Obama's test set is 358.34, which is lower than that for George H.W. Bush (412.154) and George W. Bush (465.4696).

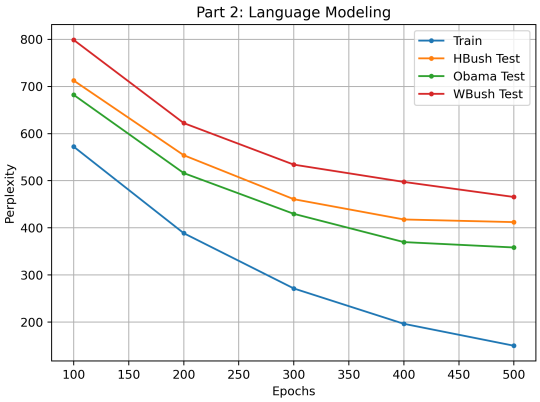


Figure 6: Evaluation of the language modeling task: train_perplexity vs test_perplexity for each epoch

3. Differences in perplexity among the politicians can be attributed to several factors:
   (a) Usually Obama's speeches are more thematically consistent, while the Bushes cover more diverse and complex topics.
   (b) The political discourse, historical context, and issues of each era might influence language predictability.
   (c) More so, I believe that Obama's structured and consistent speech is easier for the model to predict, while the Bushes' varied styles result in increased perplexity.

Part 3: Exploration

Here, I experiment with the ALiBi positional encoding, thereby modifying the transformer architecture. I also try to improve the accuracy of the classifier and the perplexity of the decoder on the test sets using better initialization of weights.

Part 3.1: Architectural Exploration

The transformer architecture is almost the same as in Parts 1 and 2, the only difference is that the transformer now uses ALiBi (Attention with Linear Biases) positional embeddings instead of absolute positional embeddings. This means the following:

- The Classifier and Block classes are unchanged.
- The Encoder and the Decoder have only 1 embedding layer (instead of 2, as in Parts 1 and 2) each. Essentially, the position embedding table is now removed.
- Multi-head attention has a new parameter called "m" which has a different constant value (power of 2) for each transformer head. In our case, its value is $\frac{1}{2^4}$ for the first head and $\frac{1}{2^8}$ for the second head.
- The attention head now implements ALiBi where it adds a bias matrix to the attention weights to encode position of the key K vectors relative to the position of the query Q vector. The value V vectors do not encode position information.

Advantages of ALiBi positional embeddings over absolute positional embeddings: ALiBi shines in transformers by understanding word relationships instead of absolute positions. This lets it handle sequences longer than trained on, unlike absolute embeddings that are limited by their position vectors. ALiBi is also faster to compute and can outperform absolute embeddings on smaller datasets, making it a strong choice for efficient transformers.

For low-level implementation details, please refer to `README.md`, and for model outputs, please refer to `part3_architectural_exploration_classification_task.json` and `part3_architectural_exploration_language_modeling_task.json`.

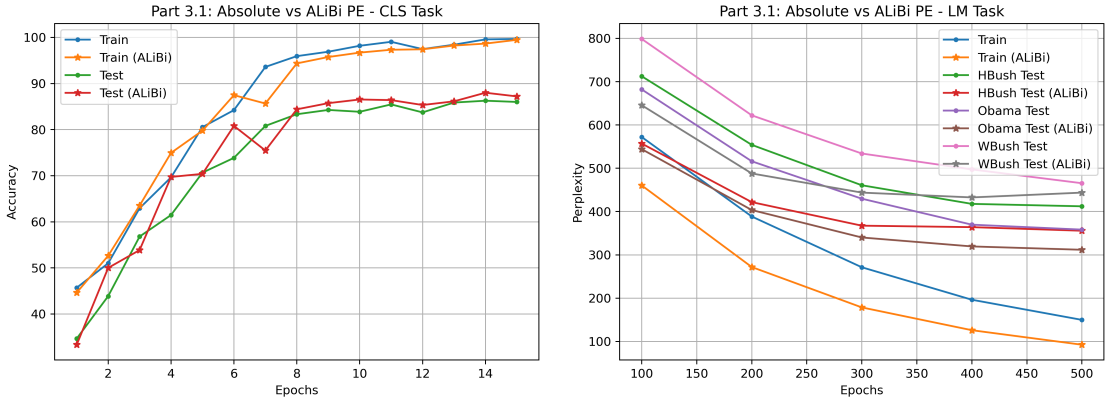
For evaluation, please refer to tables 3 and 4 and figure 7.

Table 3: Evaluation of the classification task with ALiBi positional embeddings: train_accuracy vs test_accuracy

epoch	train_loss	train_accuracy	test_accuracy
3	0.8911	63.5277	53.8667
6	0.4618	87.4761	80.8
9	0.1965	95.7457	85.7333
12	0.1156	97.4187	85.3333
15	0.0357	99.4742	87.2

Table 4: Evaluation of the language modeling task with ALiBi positional embeddings: train_perplexity vs test_perplexity

epoch	train_perplexity	hbush_test_perplexity	obama_test_perplexity	wbush_test_perplexity
100	460.4176	557.0902	544.1691	645.6212
200	271.8882	421.8129	403.5544	488.0081
300	178.6571	367.411	340.1469	443.8047
400	125.9899	364.0269	319.5039	432.6651
500	92.4801	355.716	311.7719	443.7675



(a) Classification Task

(b) Language Modeling Task

Figure 7: Comparison of evaluation metrics: Absolute vs ALiBi positional embeddings

I make the following **inferences**:

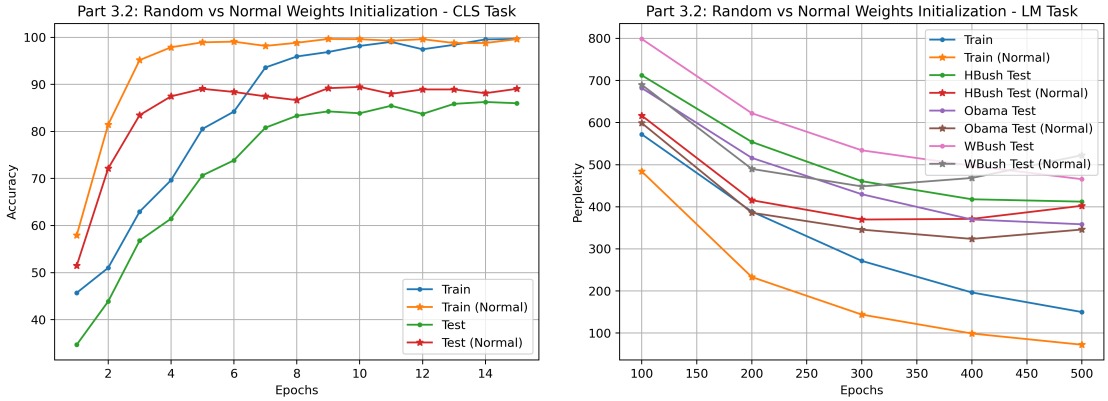
- Classification Task:
  1. Although both models show significant improvements in training metrics, achieving very high accuracy and very low loss, the classifier with ALiBi positional encoding achieves lower training loss towards the end.
  2. The classifier with ALiBi positional encoding outperforms that with absolute positional encoding in terms of test accuracy, particularly in the later epochs. The maximum test accuracy for the former (88.0%) is higher than that of latter (86.2667%).
  3. The classifier with absolute positional embeddings exhibits potential overfitting signs around epoch 12, whereas that with ALiBi postional embeddings maintains a more consistent and slightly higher test accuracy.
- Language Modeling Task:
  1. The decoder with ALiBi positional encoding has lower initial and final training perplexity compared to that with absolute positional encoding, suggesting it learns the training data more effectively.
  2. The decoder with ALiBi positional encoding outperforms that with absolute positional encoding in terms of test perplexity across all three datasets.
  3. Overall, The decoder with ALiBi positional encoding demonstrates better learning efficiency on the training data and better generalization across different test sets. Therefore, it is more robust and better suited for the given task.
- Though not shown here, but ALiBi positional embeddings also result in a slightly faster model training and evaluation.

This shows the advantage of using ALiBi positional embeddings over absolute positional embeddings in transformers.

Part 3.2: Performance Improvement

The transformer architecture is almost the same as in Parts 1 and 2, the only difference is that the transformer now initializes the weights by sampling from a normal distribution with mean of 0 and standard deviation of 0.05. This is different from Parts 1 and 2 where the weights are initialized randomly.

Advantages of Normal weights initialization over Random weights initialization: Random initialization is generally discouraged because it can lead to gradients exploding or vanishing during backpropagation, especially in deep architectures like Transformers. Normal initialization such as the Xavier Normal initializes weights from a normal distribution with a zero mean and a variance carefully chosen based on the number of input and output connections to a layer. This helps ensure the gradients have a balanced flow during training.



(a) Classification Task

(b) Language Modeling Task

Figure 8: Comparison of evaluation metrics: Random vs Normal weight initialization

For low-level implementation details, please refer to `README.md`, and for model outputs, please refer to `part3_performance_improvement_classification_task.json` and `part3_performance_improvement_language_modeling_task.json`.

For evaluation, please refer to tables 5 and 6 and figure 8.

Table 5: Evaluation of the classification task with normal weight initialization: train_accuracy vs test_accuracy

epoch	train_loss	train_accuracy	test_accuracy
3	0.4666	95.1721	83.4667
6	0.0722	99.0918	88.4
9	0.0315	99.6654	89.2
12	0.0191	99.6176	88.9333
15	0.035	99.6176	89.0667

Table 6: Evaluation of the language modeling task with normal weight initialization: train_perplexity vs test_perplexity

epoch	train_perplexity	hbush_test_perplexity	obama_test_perplexity	wbush_test_perplexity
100	483.8734	616.5609	598.9333	689.8524
200	233.0706	415.5716	385.8382	489.8964
300	143.7917	369.6368	345.6603	448.3132
400	98.8372	371.1651	323.6051	468.3859
500	72.2168	402.3132	345.888	522.5227

I make the following **inferences**:

- Classification Task:
  1. Using Normal weights initialization, the classifier converges faster to high accuracy in both training and test sets. By epoch 5, it achieves 89.0667% test accuracy, whereas using random weights initialization, it achieves 86.0% at its peak (epoch 14).
  2. Using Normal weights initialization, the classifier exhibits better generalization performance, with a smaller gap between training and test accuracy and higher test accuracy overall.
  3. Using Normal weights initialization, the classifier shows more stable test accuracy in later epochs, suggesting a more robust model compared to random initialization, which shows some variability in test accuracy.
  4. Overall, the classifier with normal initialization outperforms that with random initialization in terms of both convergence speed and generalization performance. While both models show signs of overfitting (as indicated by the high training accuracy compared to test accuracy), the former one manages this better, maintaining a higher and more stable test accuracy throughout the epochs.
- Language Modeling Task:
  1. Using Normal weights initialization, the decoder learns faster and generalizes better for H. Bush and Obama, but shows signs of overfitting or capacity limitations for W. Bush in later epochs.
  2. Using random weights, the decoder has a steady improvement in both training and test perplexity but may have difficulty generalizing as well as that with normal initialization, especially for W. Bush.

This shows the advantage of using better weights initialization such as from a normal distribution in transformers.

Code References

1. Transformer Architecture: `https://github.com/karpathy/ng-video-lecture/blob/master/gpt.py`
2. ALiBi Positional Embeddings: `https://github.com/ofirpress/attention_with_linear_biases/blob/master/fairseq/models/transformer.py`