PA2 Report

Kevin Lin

2/20/2025

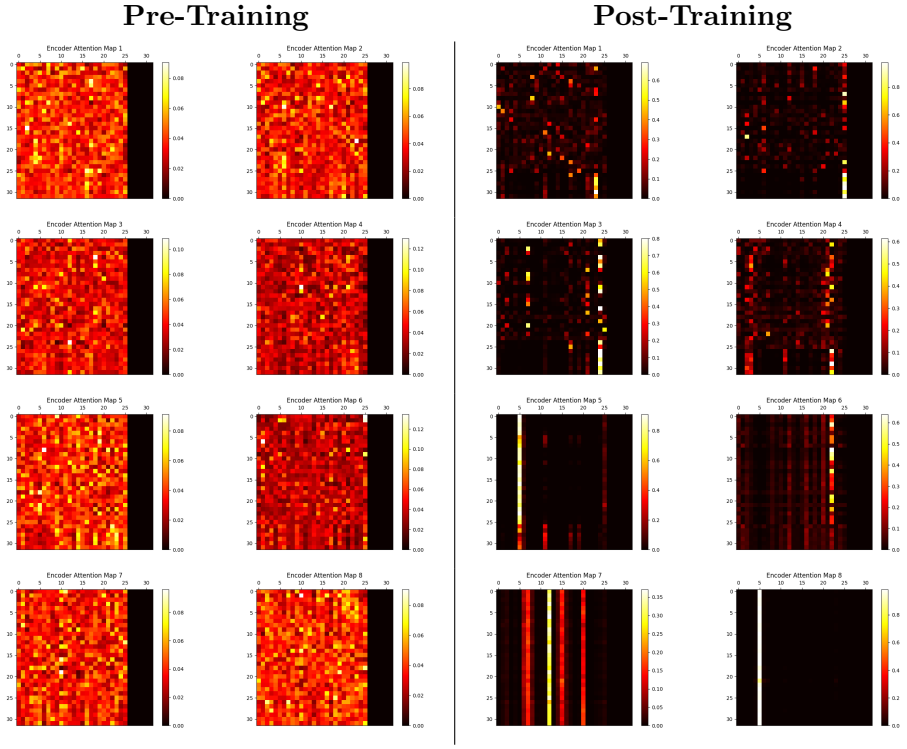
Part 1: Encoder w/ Classifier

We implement the encoder similar to discussed in the Karpathy video, in `transformer.py`. The encoder represents each input token with two learned components: a token embedding which maps vocabulary IDs to `n_embd` size dimensional vectors, and a positional embedding which maps positions of 0 to `block_size-1` to `n_embd` size dimensional vectors. These are summed to form the initial sequence representation, a standard transformer pattern. Because the classification inputs are padded to a fixed length, the encoder constructs a padding mask from the input tokens (index 0). This mask is used inside the attention to prevent the model from "reading" padding tokens as meaninful context. Here, attention is non-causl, meaning that each token can attend to any other token in the sequence (excluding padding), which is appropriate for classification where the full sentence is available.

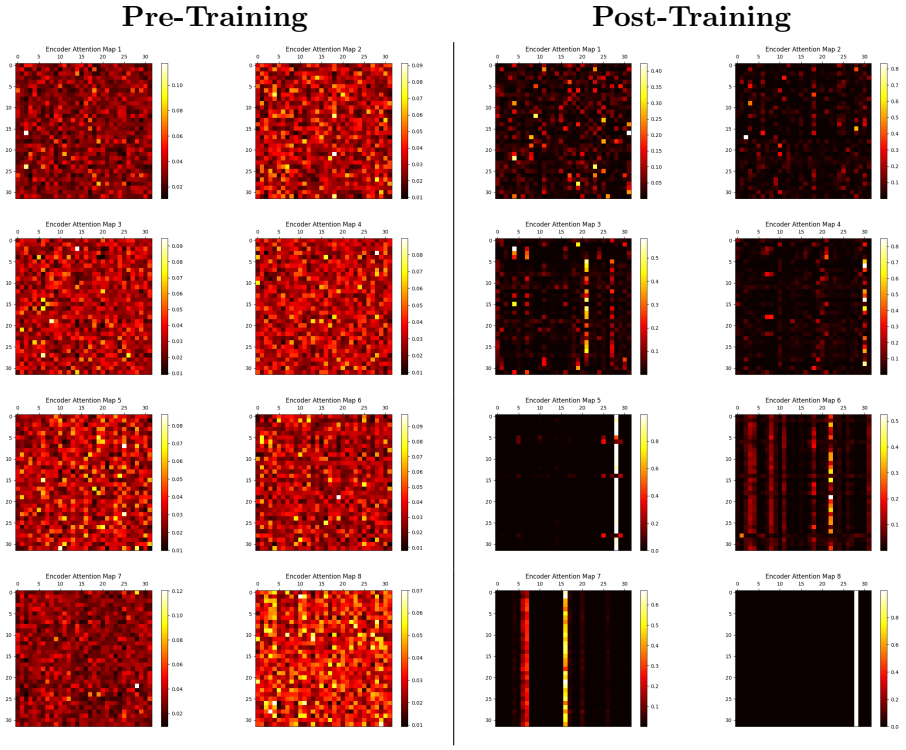
The encoder is built as a stack of `n_layer` identical `EncoderBlocks`. Each block follows the typical transformer structure of: layer normalization, multi-head self attention, residual connection, layer normalization, position-wise feedforward network, and another residual connection. The multi-head attention allows the model to learn multiple interaction patterns between tokens in parallel, while the feedfoward layer then refines each token representation independently. Residual connections help stabilize training and allow gradients to flow more easily through the network. The final output of the encoder is a contextualized representation for evry position in the input sequence, which is then passed to the classifier.

For classification, the model converts the encoder's token level outputs into a single vector per example using mean pooling across the time dimension (average of all the token embeddings, ignoring padding). This pooled representation is then fed into a two-layer feedfoward classifier: linear projection to hidden layer of size `n_hidden`, ReLU activation, then another linear projection to `n_output=3` class logits. We don't utilize dropout to keep the classifier simple. The classifier is trained end-to-end with standard cross-entropy loss, using Adam optimization.

For sanity checks, we test two sentences. The first is a short sentence: "This is a test sentence for sanity check, it has almost thirty words in it to fill the majority of the attention map graph." The second is a longer sentence: "This is a really long sentence that is meant to for the sanity test check, it has more than thirty two words in it so that we can see how the attention map looks when the sentence length exceeds the given block size." The goal here is not to test accuracy, but rather to inspect the attention maps and ensure that the model is behaving as expected across different input lengths.



We can see that for the short sentence, the attention maps shows that the model is attending to all tokens in the sequence, as expected. No attention mass was assigned to the padding positions, hence the black squares on the right most columns. Tokens attend to themselves and other tokens in the sequence, with some variation across heads. This indicates that the multi-head attention is functioning correctly, and the padding mask is properly preventing attention to padding tokens.



Now for the long sentence, we can see that the attention maps show that the model is attending to all tokens up to the block size limit of 32, and then ignores the rest of the tokens as expected. Across both tests, the encoder successfully produced consistent attention maps and contextualized representations, confirming the proper implementation of the mutli-head self attention, padding mask, and stable residual and normalization behavior across varying sequence lengths.

We train the encoder and classifier jointly from scratch on the `train_CLS` dataset, and test on the `test_CLS` dataset with the following hyperparameters:

```
vocab_size: 5755
block_size: 32
embed_size: 64
num_heads: 2
num_layers: 4
n_input: 64
n_hidden: 100
n_output: 3
batch_size: 16
```

```
learning_rate: 0.001
total_params: 576467
```

After training for 15 epochs, we achieve a final test accuracy of 87.60%, and final train accuracy of 99.24%. The training and test accuracies across epochs are shown in the table below, along with the loss and time taken for each epoch.

Epoch	Loss	Train Acc. (%)	Test Acc. (%)	Time (s)
1	1.0440	44.65	33.33	3.77
2	1.0559	48.33	38.00	3.22
3	0.8916	59.94	52.27	3.16
4	1.2615	69.79	60.93	3.10
5	0.7971	78.97	69.87	3.04
6	0.3426	84.23	75.33	3.26
7	0.2845	91.30	80.27	3.38
8	0.2748	92.93	79.87	3.23
9	0.0460	96.03	82.53	3.19
10	0.0970	97.37	83.87	3.37
11	0.2767	93.74	79.73	3.32
12	0.0604	97.47	85.47	3.24
13	0.0054	99.52	86.00	3.15
14	0.0026	99.24	87.60	3.18
15	0.0461	97.99	84.67	3.51

Part 2: Decoder Language Model

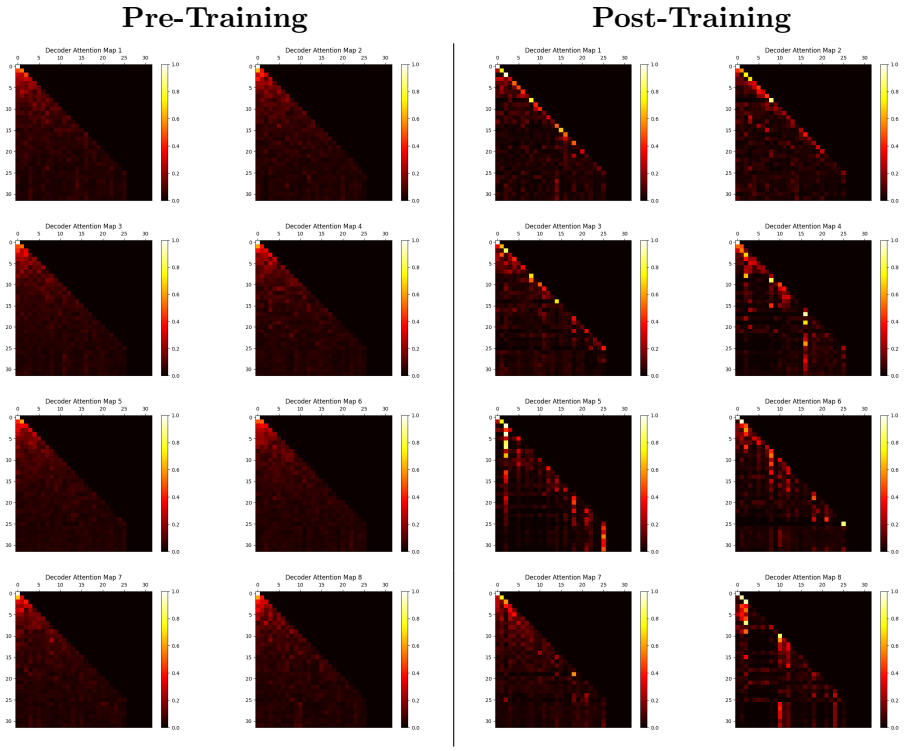
We implement the decoder similar to discussed in the Karpathy video, in `transformer.py`. This decoder replace the encoder-classifier architecture with a decoder-only transformer trained for autoregressive language modeling. The overal structure is similar to the encoder: token and positional embeddings are ssummed to produce input representations, which are then passed through a stack of transformer blocks of multi-head self attentions, feedfoward layers, residual connections, and layer normalizations. The key differene is that the decoder operate sunder a causal constraint, which enforces autoregressive behavior so that each token can only attend to itelf and preceding tokens.

We implement the causal mask by creating a lower triangular mask which is applied inside the self attention mechanism. For a sequence length of T, the mask ensures that position i can only attend to positions $0 \dots i$ and not any future positions, which are explicilty masked out before the softmax

operation by assigning them $-\infty$ attention scores. The same padding mask is applied from before after.

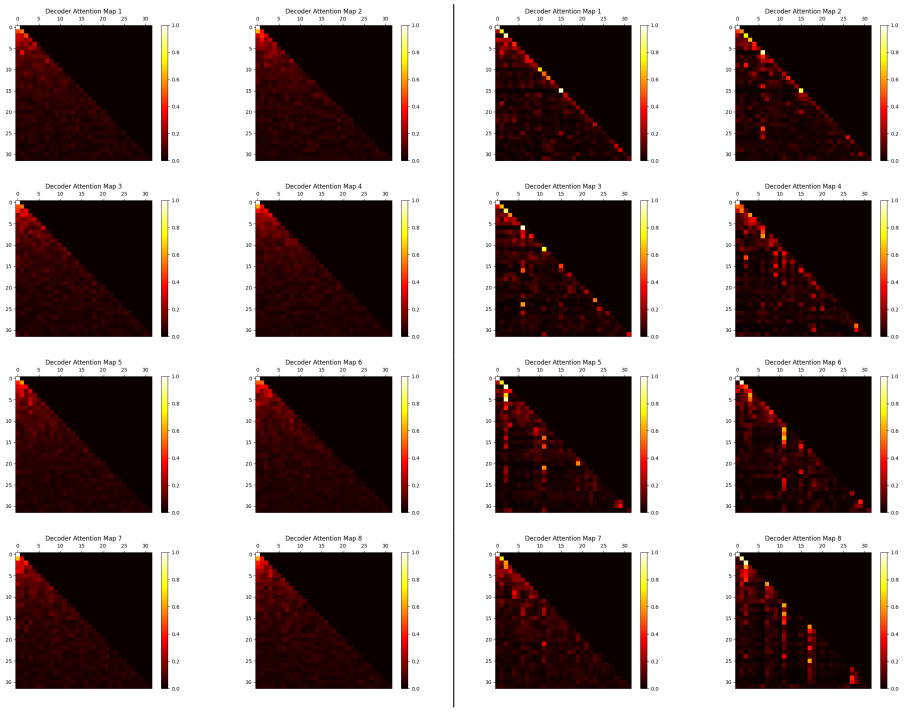
The decoder produces a contextualized representation at every time step. A final linear projection layer maps these representations to vocabulary logits. During training, the model predicts the next toekn at each position, and cross-entropy loss is computed across all time steps. Here, we evaluate using perplexity, which is the exponential of the average cross-entropy loss, and measures how well the model predicts the next token in the sequence.

We sanity test the decoder using the same two short and long sentences as before.



Here we can see that for the short sentence, each position is only attended to itself and earlier tokens, leading to the characteristic triangular attention pattern. Padding positions again, are properly masked out, leading to the black columns on the right.

Pre-Training **Post-Training**



Likewise, for the long sentence, we can see that the attention maps show the same pattern up to the block size limit of 32, confirming that the causal mask is properly implemented and the decoder is behaving as expected across different input lengths.

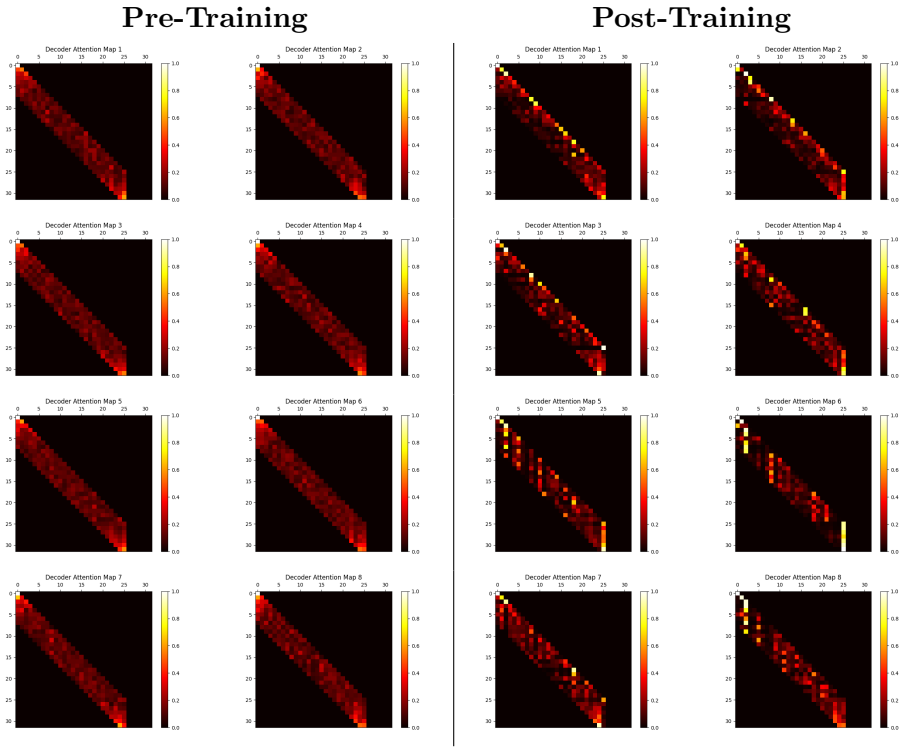
We train the decoder on the `train_LM` dataset, and evaluate on the `text_LM` dataset with the same hyperparameters as before. The only difference is that we train on all batches for the entire dataset, and is limited to 500 iterations to save time due to the large amount of tokens being processed. Due to the difference in architecture, we now have a total of 863243 parameters. We successfully obtain a final training perplexity of 162.97, Obama perplexity of 375.02, W. Bush perplexity of 476.34, and H. Bush perplexity of 414.64. Our results per every 100 iterations are shown in the table below:

Iter	Loss	Train PPL	Obama PPL	W. Bush PPL	H. Bush PPL
1	8.8477	6489.78	6421.51	6602.67	6504.76
100	6.2868	573.67	696.73	801.88	716.42
200	5.9216	411.98	554.66	645.45	569.12
300	5.7691	290.08	462.50	543.28	481.75
400	5.3495	212.91	400.33	491.96	436.82
500	5.0014	162.97	375.02	476.34	414.64

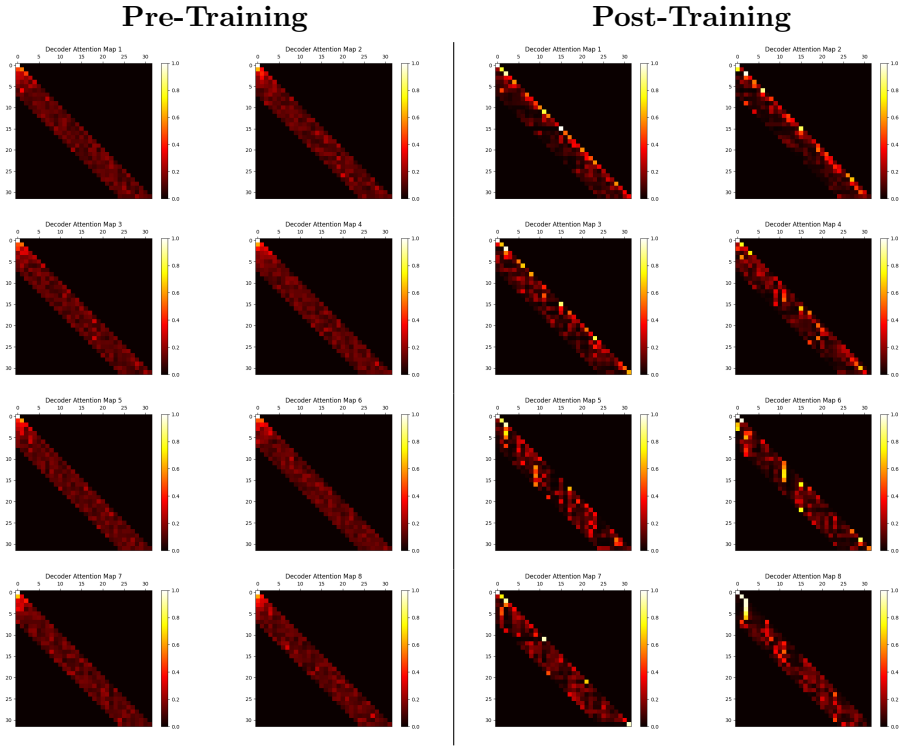
Part 3: Architectural Exploration (Local Window Attention)

Here, we modify the standard full causal self attention mechanism to utilize local window attention instead, a form of sparse. The goal is to restrict each token's perspective to a fixed number of preceding tokens, which reduces computational cost and introduces an inductive bias that may be beneficial for certain types of data when involving short-range dependencies. The local window logic is implemented by modifying the masking logic inside the self attention mechanism. The architecture of the decoder itself (embeddings, stacked Transformer blocks, feedfoward layers, residual connections, etc.) remains unchanged. Now, for a sequence of lenght T, the standard causal mask is a lower triangular matrix that allows position i to only attend to $\max 0, i - W + 1 \ldots i$, where W is a hard coded window size of 8 for our experiment.

We sanity test using the same two short and long sentences as before.



As expected, we can see that each position only attends to itself and the preceding 7 tokens, leading to a banded attention pattern. The rightmost columns are again black due to the padding mask.



As expected, we can see that the same banded attention pattern is observed for the long sentence as well, and that the entire graph is filled up to the block size limit of 32, confirming that the local window attention is properly implemented and the decoder is behaving as expected across different input lengths.

In order to compare our local window attention decoder with the standard full attention model, we train and test on the same datasets with the same hyperparameters. Likewise, we set the same random seeds for both models and load them with the same state dictionaries to ensure both models are training using the exact same batched dataset configurations. After training for 500 iterations, we obtain a final training perplexity of 157.87, Obama perplexity of 368.99, W. Bush perplexity of 460.88, and H. Bush perpexlity of 402.49. Our results per every 100 iterations are shown in the table below: Notice that the windowed attention model actually achieves lower perplexity across the board compared to the full attention model. There are a couple reasons of why this could occur. In a positive light, the local window attention forces the model to focus on local syntax and shorter phrases, which is where most of the predictive power lies in analyzing speeches anyway. The full causal attention also has a larger search space which could be noisier early as we are only training for 500 iterations. Secondly, local

Iter	Loss	Train PPL	Obama PPL	W. Bush PPL	H. Bush PPL
1	8.8014	6519.03	6441.88	6619.83	6517.20
100	6.1430	565.46	694.25	802.84	712.15
200	6.1980	411.66	550.23	639.64	562.41
300	5.7195	286.04	458.59	542.92	471.55
400	5.3605	208.52	404.07	486.27	422.23
500	5.0922	157.87	368.99	460.88	402.49

window attention is a form of inductive bias. The full attention model has more flexibility to learn long range dependencies, but can overfit for these long patterns in a small dataset. Windowing prevents this. However lastly, we can also attribute this performance difference due to the low number of training iterations and limited amount of training data in general, which may not be sufficient for the full attention model to learn, capping performance. With more training, the full attention model may eventually outperform the windowed attention model as it can learn to utilize long range dependencies, but with limited training, the windowed attention model may have an advantage due to its inductive bias and reduced search space.