CSE 256 WI 2026, UCSD: PA2 (100 points)

Transformer Blocks

Released: February 2, 2026 → Due: February 20, 2026

In this assignment, you will experiment with different parts of the transformer architecture covered in the lectures. Below is a brief summary of the tasks you will be working on:

1. **Encoder**: 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 [1].

2. **Decoder**: 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.

3. **Exploration**: You will experiment with different parts of the architecture such as the positional encoding, sparse attention patterns, etc. You will also try to improve the accuracy of the classifier or the perplexity of the decoder on the test sets.

Getting Started: If you are new to implementing transformer blocks, we warmly recommend the tutorial on implementing transformer decoders by Andrej Karpathy[2]. We also encourage you to read the original paper by Vaswani et al. (2017) [4].

Libraries: Notice that transformer blocks (encoder, decoders, ...) have been implemented in various libraries such as PyTorch. **Credit will not be given for using these libraries. You are expected to implement the transformer blocks from scratch.**

Preliminaries

Data

The dataset you will be working with consists of speeches from three American politicians. For **classification**, the task is to predict, given a short segment from a speech by one of the politicians, which politician it is from. It is a three-way classification task. The relevant files are tab-separated. Each line in the files contains a label, followed by a tab, followed by the speech segment. The label is an integer between 0 and 2, inclusive, representing the politician. The integer-to-politician mapping is as follows: `0: Barack Obama, 1: George W. Bush, 2: George H. Bush`. You will train on: `train_CLS.tsv`, and report **accuracy** on: `test_CLS.txt`. For **language modeling**, the task is to predict the next word in a sequence of words. The relevant files contain unlabeled text. You will train on `train_LM.txt`, and report **perplexity** numbers on: `test_LM_obama.txt`, `test_LM_wbush.txt`, and `test_LM_hbush.txt`.

[1] Usually the encoder is pretrained first, in this assignment we will train it from scratch instead of pretraining and then fine-tuning on a downstream task.

[2] `https://www.youtube.com/watch?v=kCc8FmEb1nY&ab_channel=AndrejKarpathy`

Starter Code

Starter code is provided to help you get started; however, feel free to ignore it and write your own code from scratch if you prefer. Start with the **same hyperparameters, defined in `main.py`**, to ensure that results are approximately comparable to the ones we expect. In part 3, you will be free to deviate from the base settings.

- `dataset.py` contains PyTorch Dataset classes for the classification and language modeling tasks.
- `tokenizer.py` contains a simple tokenizer class that tokenizes the input text into words.
- `utilities.py` contains helper functions for sanity checking your attention implementation. Given an attention matrix, it checks if the rows sum to 1 and also visualizes the attention matrix. Your encoder will need to return attention matrices, which you can use this function to check.
- `main.py` contains default parameters to use, demonstrates example usage of dataset loading, and is where you should add your training loops. The feedforward classifier can also be implemented here.
- `transformer.py` is currently empty. You will need to implement the encoder and decoder classes, attention heads, and feedforward layers, etc., here.

Positional Encoding

For the base setting, both the encoder and decoder should use simple absolute positional embeddings. Thus, you will need two embedding tables for the encoder and decoder: one for the tokens and one for the positions. The positional embeddings are added to the token embeddings before being fed into the encoder or decoder. In the exploration part, you may experiment with different positional encodings such as relative positional encodings (e.g., AliBi, mentioned in class).

Tokenizer

The tokenizer provided is a simple word-level tokenizer using the `NLTK` toolkit to split sentences into words. `NLTK` uses a bit more information than just splitting on spaces, but it is still quite basic word-level splitting. Use this tokenizer for the base setting. In the exploration part, you may experiment with different tokenizers.

Part 1: Encoder Trained With Classifier (40 Points)

For this part, you will train a transformer encoder with a feedforward neural network classifier on top of it, end-to-end. This setup is quite common in various NLP tasks. The encoder processes input sentences and converts them into high-dimensional, context-rich embeddings. These embeddings capture the semantics of the input text and the relationships between the words. The classifier takes the output from the transformer encoder and makes predictions. In our case, the classifier will predict which politician spoke the given speech segment. **There is no pretraining involved in this part; you will train the encoder and classifier jointly from scratch**.

Part 1.1: Encoder Implementation

Your first task is to implement the transformer encoder following the hyperparameters defined in `main.py`. The output of the encoder is the sequence of embeddings for each word in the input sentence. To provide the embeddings to the classifier, use the mean of the embeddings across the sequence dimension. Later, in Part 3, you can experiment with different ways of providing the embeddings, such as using the `[CLS]` token.

Part 1.2: Feedforward Classifier Implementation

Next, implement the Feedforward classifier. This classifier receives the output from the transformer encoder and makes predictions about which politician spoke the given speech segment. It consists of a simple feedforward network with one hidden layer, use dimensions specified in `main.py`.

Part 1.3: Joint Encoder & Classifier Training

For training, the input data is passed through the transformer encoder to generate embeddings, which are then fed into the classifier. The predictions from the classifier are compared against the true labels to compute a loss. This loss is used to update the weights of both the encoder and the classifier through backpropagation. Both components are trained simultaneously, enabling the encoder to learn representations that are specifically useful for the speech segment classification task.

Part 1.4: Sanity Checks

Sanity check your attention implementation using the helper function provided in `utilities.py`. This function will verify that the rows of the attention matrix sum to 1 and will also visualize the attention matrix. Include plots of attention matrices for one or two sentences in your report (you can choose any layer(s) and any head(s)). Discuss what you observe in these plots.

Part 1.5: Evaluation

Compute the accuracy of the classifier on the test set `test_CLS.txt`. Report this accuracy in your final report. **Test accuracy is expected to be in the low to mid 80s**.

In your report, include the final accuracy as well as the accuracy after each of the 15 epochs. Additionally, report the number of parameters in your encoder.

Part 2: Pretraining Decoder Language Model (30 Points)

Part 2.1: Decoder Implementation

For this part, you will implement a transformer decoder. The decoder is similar to the encoder but it uses the masked self-attention mechanism to prevent the model from peeking at future tokens during training. If you forget to mask out future tokens, fortunately it will be easy to spot it, as your train perplexity will get close to 1 very quickly. Your feedforward hidden dimensionality is 100, and the activation function is ReLU. The input and output dimensionality of the feedforward layer is the same *n_embed*.

Part 2.2: Decoder Pretraining

Pretrain your decoder on the language modeling task, predicting the next word in a sequence given the previous words. The output of the decoder consists of probabilities, one for each word in the vocabulary. The model should be trained to minimize the **cross-entropy loss** between the predictions and the true next word in the sequence.

For language modeling, we can train on all the batches for the entire dataset, but that takes a while, so we'll limit it to 500 iterations (batches) in our base setting. For a batch size of 16 and a block size of 32, this results in approximately $256,000$ tokens being processed. State-of-the-art language models are trained on trillions of tokens, so this is a very small dataset — we do not expect the perplexity numbers to be super low.

Part 2.3: Sanity Checks

Sanity check your attention implementation using the helper function provided in `utilities.py`. This function will verify that the rows of the attention matrix sum to 1 and will also visualize the attention matrix. Include plots of attention matrices for one or two sentences in your report (you may choose any layer(s) and any head(s)) and discuss what you observe.

Part 2.4: Evaluation

Report the perplexity of the decoder on the test sets: `test_LM_obama.txt`, `test_LM_wbush.txt`, and `test_LM_ghbush.txt`. **Perplexity on the training set is expected to be in the high 100s, and Obama, H. Bush, and W. Bush perplexities are expected to be in the 300s and 400s range**.
Report perplexity numbers for the different politicians in your final report, and discuss potential reasons for the differences in perplexity.
In your report, include the final perplexity, as well as the perplexity after every 100 iterations, and after completing all 500 iterations. Additionally, report the number of parameters in your decoder.

Part 3: Architectural Exploration (30 Points)

In this section, you are encouraged to explore changes to the transformer architecture. You can experiment with various components of the architecture, such as:

- **Positional Encoding:** Explore alternatives to traditional positional encodings, such as AliBi proposed by Press et al. [3].
- **Sparse Attention Patterns:** Reduce computational overhead by implementing sparse attention, where only a subset of previous tokens is attended to. Examples include:
  - Local window attention and Blockwise attention, as used in Longformer by Beltagy et al. [1]
- **Disentangled Attention Patterns:** Implement disentangled attention patterns as in DeBERTa by He et al [2].
- **Other Architectural Changes:** Feel free to experiment with any other modifications that you find interesting or beneficial.

Discuss your architectural exploration, the results you obtained, and the insights you gained from these experiments in your final report. Use plots, tables, and other visualizations to help convey your results, and help us better understand your architectural exploration.

Submission Instructions

Submit on Gradescope.

- **Code:** You will submit your code together with a neatly written README file with instructions on how to run your code. We assume that you always follow good practice of coding (commenting, structuring), thus these factors are not central to your grade.

  Ideally allow us to run your code with a single command, e.g., `python main.py` with the default hyperparameters, and three options "part1", "part2", "part3" to run the three parts of the assignment.

- **Report:** Your writeup should be 5 pages or less in PDF (with reasonable font sizes).

  You won't be penalized for exceeding the page limit, but quality is preferred over quantity. Avoid including low-level code documentation, which belongs in comments or the README. Use tables or figures to present results instead of copying the entire system output.

References

[1] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020.

[2] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: decoding-enhanced bert with disentangled attention. In *ICLR*. OpenReview.net, 2021.

[3] Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *ICLR*, 2022.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, pages 5998–6008, 2017.