CS 1678/2078 Homework Transformer

April 7, 2025

Abstract

In this assignment you will be implementing a transformer model and using it to predict the next character. To submit this assignment, upload a `.pdf` to Gradescope containing your responses to the questions below. You are required to use LaTeX for your write up. Upload a zip of your code to the Code portion of the assignment.

1 Building a Transformer

One of the goals of this assignment is for you to build your own transformer model in PyTorch. Specifically, this means creating a neural network with the following structure, where x is the input to the network of t tokens:

$$\begin{aligned} y &= \text{embedding}(x) + \text{pos\_enc}(1:t) \\ h^1 &= \text{transformer\_layer}(y) \\ h^2 &= \text{transformer\_layer}(h^1) \\ &\vdots \\ h^k &= \text{transformer\_layer}(h^{k-1}) \\ \hat{x} &= \text{linear\_layer}(\text{layer\_norm}(h^k)), \end{aligned}$$

where $\hat{x}$ represents a prediction of the next token for each of the input token, i.e., $\hat{x}_i$ is a prediction of x_{i+1}, embedding is an embedding layer which maps a token to a vector, pos_enc is provides the positional encoding of each token (this must have the same dimensions as the embedding layer output), transformer_layer is group of operations defined below that contain self-attention and a MLP and will be repeated k times. The last linear layer maps the embedding space back to the space of tokens to compute probabilities over the next token.

The transformer will be trained using the negative log-likelihood of the next token prediction. You should use the PyTorch class CrossEntropy to compute the loss. The operations of the transformer are described further below.

1.1 Embedding and Positional Encoding

The embedding layer is a mapping from a one-hot encoding of each token to a vector embedding. This layer is just a linear operation of xW, where $x \in [0,1]^d$ is a one hot vector representing one of d possible tokens and $W \in \mathbb{R}^{d \times n}$ maps that token to an n dimensional embedding space. For efficiency x never has to be explicitly represented by a one-hot vector and instead is just an integer. The layer returns the i^{th} row of the matrix if x represents the i^{th} possible token.

The positional encoding layer returns the positional encoding for each position of the sequence, i.e., if the sequence has length L, then the positional encoder returns a matrix of dimension $L \times n$, where n is the same dimension as the embedding layer. The positional encoding for the t position is

$$p(t) = \begin{bmatrix} \sin(w_1 t) \\ \cos(w_1 t) \\ \sin(w_2 t) \\ \cos(w_2 t) \\ \vdots \\ \sin(w_{n/2} t) \\ \cos(w_{n/2} t) \end{bmatrix},$$

where $w_i = \frac{1}{N^{2i/n}}$ is the frequency of the sine and cosine waves and N is a hyperparameter the controls the how low the frequencies go. Smaller frequencies means representing long periods of time. For numerical stability reasons the computation of w_i is broken up as follows

$$w_i = e^{2i \frac{-\ln N}{n}}.$$

Note that these position encoding do not have to be computed every time the network is run. Instead they can compute precomputed for some maximum sequence length, then only the L encodings that are needed for the input sequence are used.

Also notice that the positional encoding is added to the embedding layer representation for the character. Since this embedding layer will be optimized, it will be able to learn a feature representation that works with the positional encoding.

1.2 Transformer Layer

Using PyTorch implement your own transformer layer. This layer should will be similar to what we discussed in class and contain the following sequence of operations for the input x

$$y = x + \text{self\_attention}(\text{layer\_norm}(x))$$
$$z = y + \text{dropout}(\text{MLP}(\text{layer\_norm}(y))),$$

where self_attention is multi-head self-attention, layer_norm normalizes the activations of the hidden units, MLP is a MLP perception with one hidden layer and ReLU activation function, and dropout is a dropout layer that randomly sets layer outputs to zero with probability $p = 0.2$. We have not discussed dropout so far in the course, but it is a regularization method that helps prevent overfitting in neural networks. Note that all of these observations preserve the width of the network, i.e., the dimensions of x, y, and z are the same. For each of the components above you may use the built in pytorch classes and functions, but you must write your own TransformerLayer class, which brings these all together. It is important to read the pytorch documentation for each operation to make sure you have the correct dimensions and inputs for each operation.

2 Language Modeling

You will be training the transformer model on a next token prediction task. Specifically we will be using the tiny Shakespeare dataset which is a collection of scripts from Shakespeare's plays. For this part of the assignment you will need to parse the entire text file and find all the unique characters, build a dictionary that maps a character to an integer representing one of the characters, and build a dictionary to reverse that mapping, i.e., map from an integer to a character. The last dictionary is used to map model outputs to generate text.

Since it is not possible to train on the whole dataset at once, you will also need to create a data sampler that will sample a mini-batch of subsequences of length L from the dataset. The data sampler will also return the sequence of targets which is the same first sampled sequence, but shifted to the right by one, i.e., the input data is $x_{t:t+L}$ and the targets are $x_{t+1:t+L+1}$. For simplicity of the model we will assume all subsets are of length L, which means if the data set has T characters in it, the last possible subsequence that can be chosen starts at position $T - L - 1$.

3 Questions

Find the best hyperparameters that you can to minimize the validation loss. For non-small models it will be computationally expensive to train a these models on a laptop. If you have access to a GPU I highly recommend using it for this assignment. One way to get access to a GPU is to use Google's colab. You will be given a small amount of GPU credits to test your code on if you haven't used it before. I recommend testing your code locally on your own computer then trying it out on the cloud or other resources. If you do not have GPU access then you can keep your model size small. The hyperparameters below should be runable on a modern laptops 60 minutes. These hyperparameters produced an ok model that mostly produced correct looking words, but it was not very coherent.

Context Size	Embedding Size	Number of Transformer Layers	Number of heads
32	64	4	4

Try playing around with these parameters to see which ones impact the model's ability to produce good look samples the most. After finding the best set of hyperparameters train the model again without the positional encoding. Compare the two models in terms of their performance, ability to produce good samples, and examine how the attention mechanism places higher or lower weight on different aspects. To do this you will need to find "prompts" or initial context vectors that the model will then use to generate a sequence of tokens. Find prompts for which both models work well and do not work well.

1. List the hyperparameters and the best validation losses for both models

Context Size	?
Embedding Size	?
Number of Transformer Layers	?
Number of heads	?
width of MLP	?
batch size	?
N	?
validation loss with positional encoding	?
validation loss without positional encoding	?

2. Provide prompts and model outputs that make each model perform good and bad. Limit the model generated response to 500 characters.

   Positional encoding doing well:

   Prompt: ¡prompt here¿

   Place prompt for model with position encoding doing well here

   Same prompt for model without position encoding here

   Positional Encoding not doing well:

   Prompt: ¡prompt here¿

   Place prompt for model with position encoding doing not well here

   Same prompt for model without position encoding here

   Model without Positional Encoding doing well:

   Prompt: ¡prompt here¿

   Place prompt for model with position encoding here

   Same prompt for model without position encoding here

   Model without Positional Encoding doing not well:

   Prompt: ¡prompt here¿

   Place prompt for model with position encoding here

   Same prompt for model without position encoding here

3. Provide the plots of the average attention weights for both models with and without positional encoding. You can use the provided models to answer the following questions.

   (a) What do trends do you notice in the attention weights? Do the lower level layers capture different properties than the upper level? Are there any intuitive connections you can see in the pattern of attention weights? Give specific examples

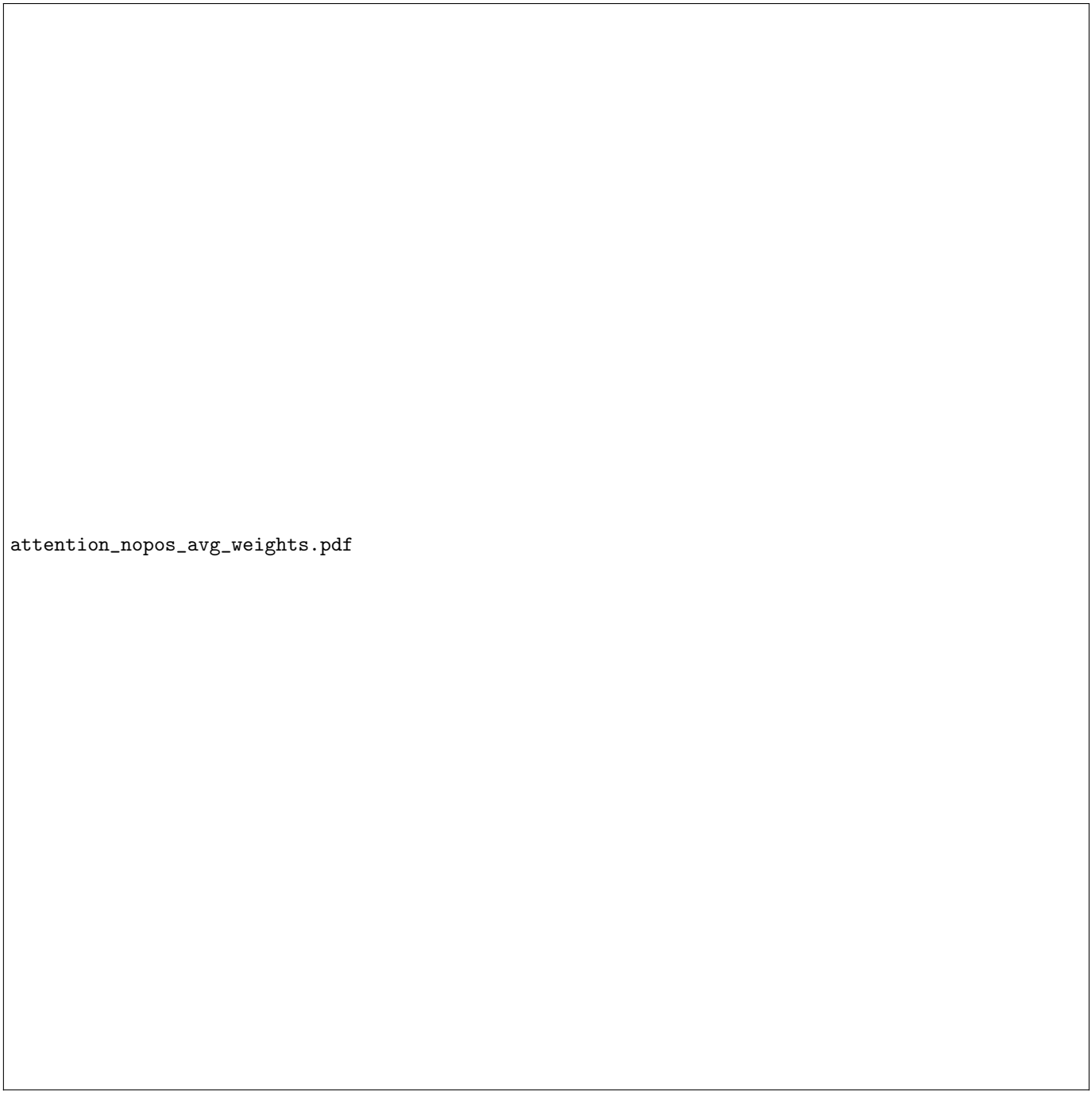
   Answer:

   (b) Plot the averaged attention weights of the model without the positional encoding. Is it any different than the model with the positional encoding? Reference specifics in the figure.

   Answer:

Replace the image files with the ones for your own project. Then delete this line.

attention_avg_weights.pdf


attention_nopos_avg_weights.pdf