CPEN455: Deep Learning
Programming Assignment 2

Released: 2025 Mar. 7

In this assignment, you will implement a Transformer [1] architecture from scratch. A Colab notebook is provided with this documentation, which you need to complete. Please pay attention to the following notes:

- The notebook is available at this link.
- **You are only allowed to modify the code inside the START/END blocks.**
- For each module, one test case is provided at the end of the notebook. You may use the test cases to get a sense of how the input/output formats are, and as a sanity check for your implementation. For each test case, your code must give a response within 10 seconds.
- Make sure your code is readable (by leaving comments and writing self-commented code).
- Unless otherwise specified, you are not allowed to use any module from *torch.nn* module.
- For each subsection, there is a corresponding START/END block in the provided colab that you need to fill in.
- All the code must be written by yourself, and you are not allowed to copy any code from other sources such as your classmates or the internet.
- If you use ChatGPT or other LLMs to help finish the assignment, please clearly mark which questions you use them. We may require you to submit your prompts. Please make sure you store them properly.
- Failing to support GPU training will result in a loss of points, as efficient hardware utilization is essential for performance evaluation.
- If the training process is excessively slow, points will be deducted, as computational efficiency is a key criterion for assessment.

1 Transformer

In this problem, we will implement a Transformer from scratch and train it on a toy task. The setup of the problem is as follows: We are given training/validation/test sets of size $N_{\text{train}}, N_{\text{val}}, N_{\text{test}}$, respectively. Each dataset contains a set of strings consisting of four English alphabet characters "c","p","e", and "n". The task is to verify if the sequence contains "cpen" as a contiguous substring or not. For instance "ecpenp" contains this substring and has label 1, while "cpeeen" does not contain the substring and has label 0. Each string s_i $(1 \leq i \leq N)$ in each dataset contains sequences of length n. For exact details of this task, you may refer to the implementation of *SubstringDataset* in the notebook. An illustrative figure of this task is shown in Figure 1. Throughout this assignment,

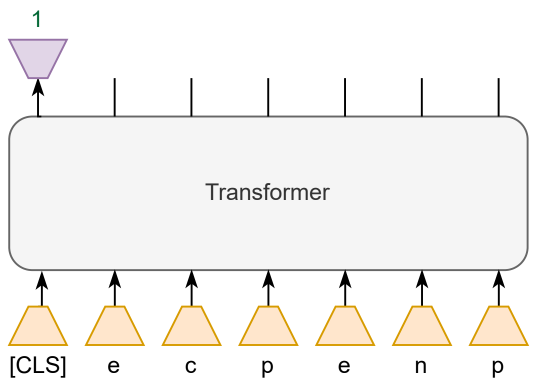

Figure 1: Substring matching using a Transformer. A string is first tokenized into a sequence of characters. A "[CLS]" token is added to the beginning of the sequence. The sequence is encoded and passed through a Transformer. Finally, the output of the "[CLS]" token is decoded as the class label. The label is 1 if the string is matched, 0 otherwise.

we omit the batch size in the equations for simplicity but your implementations must take into account the batch size.

Preprocessing and Tokenization

First, we need to convert our dataset, which is represented as string data, to a vectorized format. To this end, we first need to break the string into small pieces. This process is called *Tokenization.* Next, we have to convert each of the small pieces into a vectorized format, usable by a neural network.
1.1 [10pts] Implement the tokenization and vectorization functionality in the Tokenizer class. This function takes in a string, (1) splits the string, and returns a list of characters (or *tokens* in Transformer terminology) (2) (optionally) add a "[CLS]" token to the beginning of the list (later, we will see why we need this). (3) convert each token into a one-hot vector and return the resulting matrix. For a string of length n, this function must return a row-wise one-hot matrix $\mathbf{X} \in \mathbb{R}^{(n+1)\times d_{\text{voc}}}$ where $d_{\text{voc}} = 4 + 1 = 5$ is the size of our token vocabulary.

Positional Encoding

The Transformer model is designed to process sequences of tokens, but it lacks any inherent understanding of the order or position of those tokens within the sequence. This means that the model would treat the same sequence of tokens differently depending on their order within the input. To overcome this limitation, the Transformer architecture introduces positional encoding.

Positional encoding is a way of adding information about the position of each token in the sequence to the input embeddings before passing them through the Transformer layers. This allows the model to distinguish between tokens based on their position in the sequence.

There are various forms of imposing positional encoding such as sinusoidal positional encoding introduced in the original Transformer paper. Here, we will implement another version, which

is learnable positional encoding. The idea behind learnable positional encoding is to dedicate a learnable vector to each position index i, instead of a fixed sinusoidal vector.
1.2 [5pts] Implement the (absolute) learnable positional encoding module. This module has a learnable weight matrix $W_{\text{pos}} \in \mathbb{R}^{d_{\text{maxLen}} \times d_{\text{model}}}$. The i-th row ($1 \leq i \leq d_{\text{maxLen}}$) of this matrix is a learnable vector corresponding to the i-th position in a sequence. This module applies positional encoding to a sequence by element-wise adding rows of this matrix to their corresponding position in the input.

Multi-Head Attention

Attention is the key component of the Transformer, which we will implement next. Attention allows the model to attend to different parts of the input sequence with different weights, enabling it to capture complex relationships between the input tokens. This module consists of *numHeads* heads, each head h consists of three weight matrices: $\mathbf{W}_{K,h}, \mathbf{W}_{Q,h}, \mathbf{W}_{V,h} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{h}}}$, where $d_{\text{h}} = d_{\text{model}}/d_{\text{h}}$. Multi-head attention takes three matrices $\mathbf{X}_K, \mathbf{X}_Q, \mathbf{X}_V \in \mathbb{R}^{n \times d_{\text{model}}}$ as input, where n is the number of tokens. The computational mechanism of each head is as follows:

$$head_h = \text{Softmax}\left(\frac{(\mathbf{X}_Q \mathbf{W}_{Q,h})(\mathbf{X}_K \mathbf{W}_{K,h})^T}{\sqrt{d_{\text{h}}}}\right)(\mathbf{X}_V \mathbf{W}_{V,h}),$$

where Sofotmax is a row-wise softmax operator. The result of each head is a $n \times d_{\text{h}}$ matrix. To merge the outputs of heads and obtain a final output of size $n \times d_{\text{model}}$, heads are concatenated and multiplied by a weight matrix $\mathbf{W}_O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$:

$$\text{Attention}(\mathbf{X}_K, \mathbf{X}_Q, \mathbf{X}_V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_H)\mathbf{W}_O.$$

In this assignment (and in Transformer Encoder in general), we use *Self-attention* where $\mathbf{X}_K = \mathbf{X}_Q = \mathbf{X}_V = \mathbf{X}$ are all the same matrices.
1.3 [20pts] Implement the multi-head attention module.

1.5 [5pts] An alternative way of imposing positional encoding in the Transformer is to apply it in the attention Softmax operator. That is, modify the equation of each head as follows:

$$head_h = \text{Softmax}\left(\frac{(\mathbf{X}_Q \mathbf{W}_{Q,h})(\mathbf{X}_K \mathbf{W}_{K,h})^T + \mathbf{M}_h}{\sqrt{d_{\text{h}}}}\right)(\mathbf{X}_V \mathbf{W}_{V,h}),$$

where $\mathbf{M}_h \in \mathbb{R}^{n \times n}$ is a learnable Toeplitz matrix

$$M_h = \begin{bmatrix} m_{0,h} & m_{1,h} & m_{2,h} & \cdots & m_{n-1,h} \\ m_{-1,h} & m_{0,h} & m_{1,h} & \cdots & m_{n-2,h} \\ m_{-2,h} & m_{-1,h} & m_{0,h} & \cdots & m_{n-3,h} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{-(n-1),h} & m_{-(n-2),h} & m_{-(n-3),h} & \cdots & m_{0,h} \end{bmatrix},$$

with $2n-1$ degrees of freedom $m_{-(n-1),h}, \ldots, m_{0,h}, \ldots, m_{n-1,h}$. In each head, this positional encoding adds a scalar value of $m_{i-j,h}$ to the attention score of each pair of tokens with indices i, j ($1 \leq i, j \leq n$). Since in this positional encoding, only the relative position of each pair of tokens is important, this positional encoding is called Relative Positional Encoding (RPE). Implement relative positional encoding (RPE) in the multi-head attention module.

Transformer Backbone

Next, we will proceed with the implementation of Transformer layers and model. Each layer of the transformer contains four components: self-attention, feed-forward layer, and normalization. The feed-forward layer is a two-layer fully connected network with ReLU activation. It contains two weight matrices $\mathbf{W}_1 \in \mathbb{R}^{d_{\text{model}} \times 4d_{\text{model}}}, \mathbf{W}_2 \in \mathbb{R}^{4d_{\text{model}} \times d_{\text{model}}}$, and its output is given by:

$$FC(X) = \text{ReLU}(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2.$$

1.4 [15pts] Implement the transformer layer module. The *prenorm* parameter determines whether the transformer layer is Pre-Norm or Post-Norm.
1.5 [15pts] Implement the transformer model module. This module contains an encoder, *nLayers* layers of transformer layer, and a decoder. The encoder weight $\mathbf{W}_{\text{enc}} \in \mathbb{R}^{d_{\text{input}} \times d_{\text{model}}}$ is a linear layer that takes in tokens of dimension $d_{\text{input}} = d_{\text{vocab}}$ and encodes each of them into a high-dimensional space of d_{model}. After applying the required layers of the Transformer layer, the decoder weight matrix $\mathbf{W}_{\text{dec}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{out}}}$ decodes each token from a high-dimensional space into the output space.

Optimization Scheduling

Next, we implement the optimizer and its scheduler. We choose Adam optimizer as the optimizer. Transformers often require warm-up and cool-down scheduling for stable training and better/faster generalization. That is, in the early steps of optimization, the learning rate is gradually increased from zero to the base learning rate, and in the final stages of optimization, the learning rate is decreased to zero again. Here, we implement a simple scheduler with linear warmup and cooldown.
1.6 [5pts] Implement a scheduler with *warmUp* warmup steps and *maxSteps* total number of steps. In other words, your scheduler must return a zero learning rate at step 0, increase the learning rate linearly to *lr* until step *warmUp*, and decrease it again linearly to zero until step *maxSteps*.

Train Substring Matching

Finally, we have all the pieces necessary to train a substring matching model using the Transformer architecture. The *Trainer* class contains a minimal training and evaluation procedure for this task.
1.7 [15pts] Implement the loss computation function in the Trainer class. The Transformer contains n output vectors for input with n tokens. However, our task is to have a single output for the entire sequence (*i.e.* the label). Hence, we have added a dummy token called "[CLS]" token, and will only look at the output of this token to compute the loss and accuracy. The loss/accuracy computation function should compute the predicted label, and binary cross entropy with the ground truth labels, and return both the loss and accuracy for a given batch. You may use the cross entropy loss function from PyTorch to compute loss.
1.8 [10pts] Run the training procedure to make sure your code is correct and the model trains properly. In case you were not successful in implementing some of the previous parts, you may use a module from PyTorch (e.g., Attention, Transformer, etc) to implement this part and receive the full points for this part. If you have implemented everything correctly, your code should achieve close to perfect accuracy on the test set.

References

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention is all you need. Advances in neural information processing systems, 30.

[2] Raffel, C., Shazeer, N.M., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P.J. (2019). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. ArXiv, abs/1910.10683.