Assignment (4)

24789 - Intermediate Deep Learning (Spring 2023)

Out Date: April 4th
Due Date: April 12th 11:59 pm

Submission file structure

p1_report.pdf
p2.pdf

Please submit **p1_report.pdf** to **Homework 4 Theory** section and **p2.pdf** to **Homework 4 Programming** section. The first report should contain the answer to **Problem 1**, and the second report should be the Jupyter Notebook (code of **Problem 2**) saved in pdf format.

You can refer to Python3 tutorial, Numpy documentation and PyTorch documentation while working on this assignment. Any deviations from the submission structure shown below would attract penalty to the assignment score. Please use Piazza for any questions on the assignment.

Theory Exercises (10 points)

PROBLEM 1

Basics of Attention [10 points]

The attention mechanism is a new type of learnable neural network layer that has attracted much interest. It has been the fundamental building block for a wide array of very successful deep learning models, such as GPT [1] and AlphaFold2 [2].

The attention mechanism is essentially computing a weighted average of input features with the weights α_{li} dynamically computed based on the query (q_i)/key(k_i)/value(v_i) [1]:

$$\alpha_{li} = \frac{\exp\left(f_{attn}\left(q_l, k_i\right)\right)}{\sum_j \exp\left(f_{attn}\left(q_i, k_j\right)\right)}, \quad o_l = \sum_i \alpha_{li} \cdot v_i. \tag{1}$$

a) [6 points] Suppose you are applying a layer of attention to a machine translation task. The input x_i is the encoding of each word in the sentence, which contains three words:

$$X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.8 & -0.2 \\ -0.5 & 0.5 \\ 0 & 0.5 \end{bmatrix}. \tag{2}$$

The following weight matrices are applied to the input features to project them into query, key, and value:

$$q_i = x_i W_q; \quad k_i = x_i W_k; \quad v_i = x_i W_v \tag{3}$$

$$\text{where: } W_q = \begin{bmatrix} 0.5 & 0.5 \\ 0 & 1 \end{bmatrix}, \quad W_k = \begin{bmatrix} 1 & 0 \\ -0.5 & 0.5 \end{bmatrix}, \quad W_v = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4}$$

The attention function $f_{\text{attn}}(\cdot, \cdot)$ used here is a simple dot product, i.e. $f_{\text{attn}}(q_i, k_j) = q_i^T k_j$. Now compute the final output features O using formula given in Eq. (1) and Eq. (3), where O is defined as:

$$O = \begin{bmatrix} o_0 \\ o_1 \\ o_2 \end{bmatrix}. \tag{5}$$

b) [4 points] Suppose every element of $q_i = [q_i^0, q_i^1, \ldots, q_i^{d-1}]$ and $k_j = [k_j^0, k_j^1, \ldots, k_j^{d-1}]$ subject to the same Gaussian distribution $\mathcal{N}(0, \sigma^2)$, i.e. $q_i^l \sim \mathcal{N}(0, \sigma^2), k_j^l \sim \mathcal{N}(0, \sigma^2), \forall l \in \{0, 1, \ldots, d-1\}$. Assume all elements are independent to each other, compute the variance of: $q_i^T k_j = \sum_{l=0}^{d-1} q_i^l k_j^l$.

Hint: For two independent variable X, Y, the variance of their product XY can be computed via: $\text{Var}(XY) = E(X^2Y^2) - (E(XY))^2$, where $E(\cdot)$ denotes expectation(for standard Gaussian it's simply 0). In addition, since X, Y are independent, they have following useful properties: $E(XY) = E(X)E(Y)$; $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$.

[1] Here it is assumed that query/key/value are all calculated from the same input sequence, which is known as self-attention. Attention can also be used to fuse features from two different input sources, where query is calculated from one source, key and value are calculated from another. This is usually termed cross-attention.

Programming Exercises (70 points)

PROBLEM 2

Part i: Prepare for a Transformer (20 points)

In the first part of this problem, you are asked to implement standard scaled dot product attention (both causal and non-causal versions) and positional encoding from ***Vaswani et al.***[3]. They are closely related to the Transformer you will implement and train in the next part of the assignment. We have provided a **Jupyter Notebook (pde_gpt.ipynb)** template for you, please follow the instruction and comment on the template when implementing your code.

a) [5 points] In problem 1 you have computed dot product attention by hand. Now you will implement a slightly different version of it, which is called scaled dot product attention. Rewriting Eq. (1) in matrix format:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \tag{6}$$

Based on Eq. (6), you will implement the `scaled_dot_product_attention` function.

b) [5 points] Up till now, it is always assumed that the context of calculation is the whole sequence, which means when you are calculating the attention weight of a specific query (α_{li} in Eq. (1)), you will consider all the keys $k_i, \forall i \in \{0, 1, \ldots, N-1\}$, with N being the length of the sequence. However, this setting is problematic for the following scenarios.

- The attention model is used to approximate a probability density function $P(x_t|x_{t-1}, x_{t-2}, \ldots)$, so it is unreasonable to make the next prediction of x_t based on future x_{t+m}, as it breaks the causality in the data and farther future is usually unknown before you have the nearest future predicted.
- In NLP task, it is impossible that every sentence has the same length. Recall that in mini-batch training, we need to stack the data array of each sample together to form a bigger array, which requires the dimension of each data array to be aligned. To achieve the same size, people usually pad shorter sentences with zeros. These zeros are rather meaningless when we calculate the attention weights.

A simple and efficient remedy is to modulate the attention weight matrix using boolean masking. You will implement the `masked_scaled_dot_product_attention` function that will mask out unnecessary key/query dot product before computing softmax.

Hint: You might find this function helpful: `TORCH.TENSOR.MASKED_FILL_` .

c) [10 points] A special characteristic of attention is that it is permutation-equivariant with respect to its inputs (you can verify this for yourself by switching the position of x_1 and x_2 in problem 1). For a lot of tasks, positional information is actually very important. For example, switching the order of words in a sentence could result in entirely different meanings. To inform the model of the positional information of each feature vector, ***Vaswani et al.***[3] propose to modulate the input features X with a positional encoding function PE.

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d}}\right) & \text{if} \quad i \bmod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d}}\right) & \text{otherwise} \end{cases} \tag{7}$$

where d is the hidden dimension. As an example, the i-th dimension of feature x_n's positional encoding is $PE_{(n,i)}$. The updated feature is then $x_n^i + PE_{(n,i)}$.

Using above formula, implement the `__init__` and `forward` method in the `PositionalEncoding` class.

Part ii: Use Transformer to predict the dynamics of 1D Burgers' equation (50 points)

For the second part of this problem, you will implement a Transformer [2] to predict the solution function $u(x,t)$ of Burgers' equation at different time steps t, where x denotes the spatial coordinates. More specifically, a 1D CNN encoder will first encode the solution function at a particular time step t_k (of all spatial locations) $u(x, t = t_k), x \in \{x_i\}_{i=1}^n$, to a latent embedding vector $e_k \in \mathbb{R}^d$ where d is the hidden dimension (i.e. number of channels). Transformer $\phi(\cdot)$ will then predict the future embedding in a recurrent way, i.e. $\hat{e}_{k+1} = \phi(e_k), \hat{e}_{k+2} = \phi(\hat{e}_{k+1}), \cdots$. After the target time step T is reached, a 1D CNN decoder will be used to decode latent embeddings back to the physical space (e.g. $\hat{e}_T \mapsto u(x, T), x \in \{x_i\}_{i=1}^n$).

Note that $\hat{e}_{k+2}$ remains unknown until we have $\hat{e}_{k+1}$ predicted, which makes the training very inefficient. To make the training parallelizable (i.e. $\hat{e}_{k+1}, \hat{e}_{k+2}, \cdots$ can be derived in parallel during training), a technique called Teacher Forcing will be used here (see `https://towardsdatascience.com/what-is-teacher-forcing-3da6217fed1c` for detailed explanation).

a) [20 points] Implement a masked multi-head self-attention module (filling in the blanks of `forward`, `__init__` under `CausalSelfAttention` class).

In **Part i** of this problem you have implemented a masked scaled-dot product, now you will extend it to support multi-head, batch computation and impose a causal mask on the attention weight. A causal mask is a mask that will be used to mask out all the upper triangular elements in the n by n attention matrix (QK^T). The multi-head mechanism provides a way to scale up the model without increasing much computation cost. It computes attention on the input with different projection layers in parallel. As described in the original Attention paper, it is formally defined as:

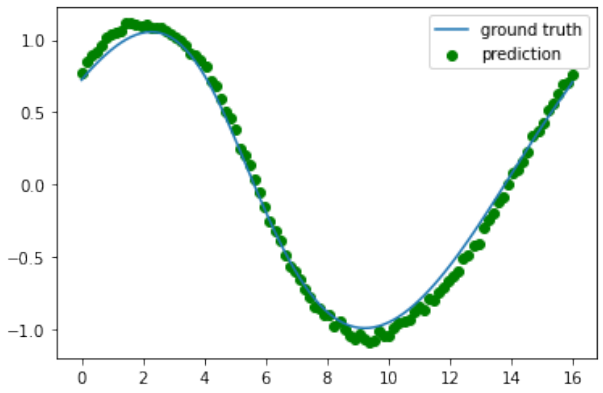
$$\text{MultiHead}(X) = \text{Concat}(\text{head}_i, ..., \text{head}_h)W^O \tag{8}$$

$$\text{where:} \quad \text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V) \tag{9}$$

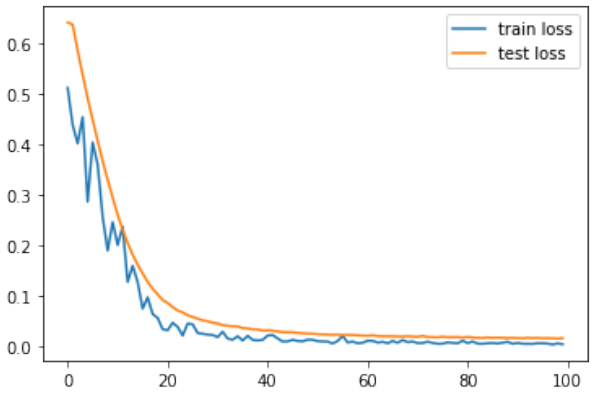
b) [10 points] Implement the forward loop in the `forward` function of `PDEGPT` class (fill in the blanks following the comment on the template).

c) [20 points] Now you can play around with different learning rates/number of attention layers and the number of epochs to train a better model. A correct implementation and reasonable hyperparameter would yield the following final visualization plots. The grading criterion is given as follows:

- 20 points if final evaluation error < 0.05.
- 25 points (20 + 5 bonus) if final evaluation error < 0.02.
- 30 points (20 + 10 bonus) if final evaluation error < 0.01.



(a) Prediction-ground truth at the last frame



(b) Sample training/testing loss curve

[2] To be more precise, it's a Transformer decoder [3]

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

[2] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. Nature, 596(7873):583–589, Aug 2021. ISSN 1476-4687. doi: 10.1038/s41586-021-03819-2. URL https://doi.org/10.1038/s41586-021-03819-2.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.