1 Outline

In this assignment, you are asked to design Transformer model for sentiment analysis.

2 Specification

In this assignment, you are asked to write a Python code to implement a sentiment analysis model for IMDB movie review dataset. Specifically, you need to implement Transformer from paper "Attention is all you need" (Vaswani et al 2018). The implementation details are provided in the following sections.

3 IMDB movie review dataset

Our goal is to classify the reviews from IMDb datasets to be of either positive or negative sentiment. The label of a review is binary: 1 if the review is positive, 0 negative.

We provide the dataset file `imdb_dataset_dict.pt` which contains the dataset of movie reviews in IMDb site. The dataset is constructed as follows: the dataset of review texts are obtained from `IMDB` function provided by `torchtext` module. The text datasets are converted to a tensor dataset to be used for PyTorch. A review in the dataset is tokenized, and converted to a tensor vector with fixed length of 384. This means we only consider reviews not exceeding 384 words (there are much longer reviews in `IMDB` dataset) There are some special tokens which do not represent words but have special meanings.

- `<UNK>` Unknown tokens – words which are not in typical English dictionary are treated as unknown ones (Out-Of-Vocabulary or OOV tokens)
- `<PAD>` Padding token. A review whose length is shorter than 384 words will be padded with this token.

For this assignment, you will need dataset file `imdb_dataset_dict.pt`. You can also print sample reviews by setting `show_sample_reviews` to True.

4 Model

We consider the encoder part of Transformer architecture in Fig. 1. This is `TF_Encoder` module in `hw7.py` file. The input to the encoder is a sequence of tokens which are the index of words in the vocabulary. For example, if "cat" is assigned with word index 7 out of say 20,000 total words in the vocabulary, number 7 is input to the encoder.

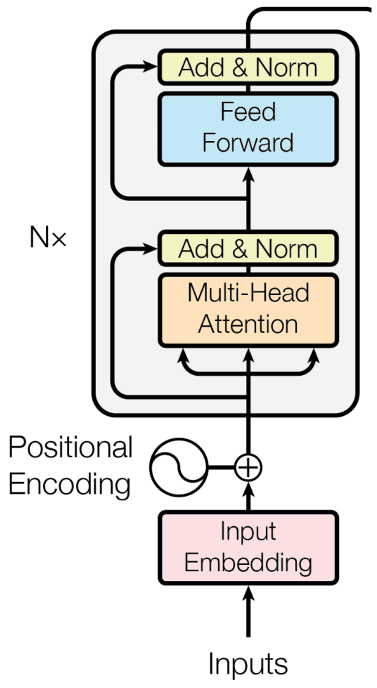

Figure 1: Transformer encoder architecture.

The tokens are converted to embedding (a vector, see "Input Embedding" in Fig. 1) which is implemented by `nn.Embedding` layer provided by PyTorch. Then positional encoding is applied ("Positional Encoding" in Fig. 1) to the embedding. Then it is fed into a sequence of N layers of `TF_Encoder_Block` module (in Fig. 1). Each `TF_Encoder_Block` consists of `MultiHeadAttention` module and feed forward (linear) modules and some normalization. We explain the modules in detail.

4.1 class `MultiHeadAttention`

The class implements multi-head attention in Fig. 2. The class have two methods to be implemented: `__init__` and `forward` which are **Q1** and **Q2**.

4.1.1 `__init__` method

The method has the following format:

`__init__(self, d_model, d_Q, d_K, d_V, numhead, dropout)`: Constructs an object for multi-head attention. Input parameters are as follows:
Parameters:

- `d_model` :*int*, size of feature vector in the Transformer.

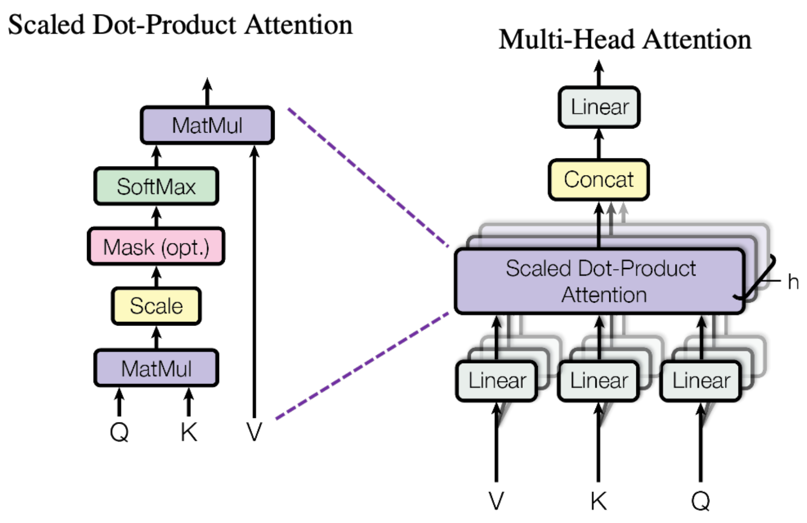

Figure 2: Multi-head Attention

- `d_Q, d_K, d_V` :*int*, size of Q, K, V for each head of the multi-head attention. Typically passed as `(d_model / numhead)` from `TF_Encoder_Block`.
- `numhead` :*int*, number of heads in multi-head attention.
- `dropout` :*float*, dropout probability.

In `__init__` method, you will need to create class objects such as

- `nn.Linear` layer for `Q,K,V` for multi-head attention ("Linear" blocks in lower-right of Fig. 2). In the `forward` operation, the input features `x_Q,x_V,x_K` of length `d_model` will be projected to outputs of feature length `d_Q,d_V,d_K` for each head. Note that there are a total of `numhead` heads, so make sure to create Linear layer according to the specification.
- `nn.Linear` layer after multihead attention results are joined ("Linear" block after "Concat" in Fig. 2)
- `nn.Dropout` layer for dropout

4.1.2 `forward` method

The method computes the scaled dot-product attention (see left of Fig. 2) has the following format:
`forward(self, x_Q, x_K, x_V, src_batch_lens=None)`: Input parameters are as follows:
Parameters:

- `x_Q, x_K, x_V` :*tensor*, Q, K, V inputs having shape `(B,T_T,d_model)` `(B,T_S,d_model)` and `(B,T_S,d_model)` respectively. `B` denotes the batch size, `T_S,T_T` denote source and target sequence lengths, respectively.
- `src_batch_lens` :*tensor*, has shape `(B,)`. Contains the length information of batched source. Example: suppose the batch size is 3, and if the first, second, and third input data (tokens of words from review) has length 3, 8, 5. Then `src_batch_lens` will be `[3,8,5]`. Suppose `T_S` = 10. Then the first input from batch will have 3 word tokens and 7 `<PAD>` tokens. The second input will consist of 8 word tokens and 2 `<PAD>` tokens. The third input from batch will have 5 word tokens and 5 `<PAD>` tokens.

  Note these `<PAD>` tokens should be **ignored** when we compute the attention coefficients. That is, the attention coefficient computed from softmax operation should be sufficient small on input positions with `<PAD>`. You are free to implement any kind of approximation as long as the coefficients are small enough. Use `src_batch_lens` to find out which part of source input is `<PAD>`.

Returns:

- `out`:*tensor*.

  Output tensor of shape `(B, T_T, d_model)`

Operation:
Inputs `x_Q,x_K,x_V` is first projected to each head through linear layers (as in Fig. 2). Then the following Scaled Dot-Product Attention is Fig 2 is applied to each head:

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_K}}\right)V$$

which uses `Matmul,Scale,Softmax` in left of Fig. 2. `Mask (opt.)` layer masks out source tokens which are `<PAD>`, so that they have negligible effect on computing softmax. The masking can be achieved using `src_batch_lens`: refer to the description above in `src_batch_lens`.
Where to put dropout: There are two places to put dropout, according to the implementation guide in the original paper:

1. Before applying attention coefficients to V. That is, if we define A as

   $$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

   then apply dropout to A, then multiply with V.
2. After applying the final `Linear` layer. Thus, right before returning `out`, apply dropout.

4.2 class `TF_Encoder_Block`

The class implements Transformer encoder block in gray box in Fig. 1. The class have two methods to be implemented: `__init__` and `forward` which are **Q3** and **Q4**.

4.2.1 `__init__` method

The method has the following format:

`__init__(self, d_model, d_ff, numhead, dropout)`: Constructs an object for encoder block. Input parameters are as follows:

Parameters:

- `d_model` :*int*, size of feature vector in the Transformer.
- `d_ff` :*int*, size of feature vector in Feed Forward block.
- `numhead` :*int*, number of heads in multi-head attention.
- `dropout` :*float*, dropout probability.

In `__init__` method, you will need to create class objects such as

- `MultiHeadAttention` block for Multi-head Attention layer in Fig. 1. In your instantiation of `MultiHeadAttention` object, you can pass `d` to `d_V`, `d_Q`, `d_K` where `d=int(d_model/numhead)`. Although you can set `d_model` and `numhead` arbitrarily, it is recommended that their ratio is integer.
- `nn.LayerNorm` block for "Add&Norm" after multi-head attention in Fig. 1
- Feed forward block for "Feed Forward" layer in Fig. 1. Feed forward block consists of the following sequence of layers:

  ```
  Linear1 - ReLU - Dropout - Linear2 - Dropout
  ```

  `nn.Linear1` takes `d_model` dimensional input feature and `d_ff` dimensional output feature. `nn.Linear2` takes `d_ff` dimensional input feature and `d_model` dimensional output feature. `Dropout` layers take the `dropout` arguments as the dropout probability.
- `nn.LayerNorm` block for "Add&Norm" after feed forward layer in Fig. 1

4.2.2 `forward` method

The method computes the scaled dot-product attention (see left of Fig. 2) has the following format:

`forward(self, x src_batch_lens=None)`: Constructs an object for multi-head attention. Input parameters are as follows:

Parameters:

- `d_model` :*int*, size of feature vector in the Transformer.
- `x` :*tensor*, x input feature having shape `(B,T_S,d_model)`.
- `src_batch_lens` :Same as explained previously. You should pass `src_batch_lens` to your `MultiHeadAttention` object instantiated in this class.

Returns:

- `out`:*tensor*.

  Output tensor of shape `(B, T_S, d_model)`

Operation:
`forward` function should perform

1. Feed input x to multi-head attention layer. Note that the operation of this layer is self-attention, so set the input properly.
2. attention output is added to x (skip connection), then perform layer normalization
3. then the output is fed into feed forward layer
4. feed forward output is added to its input (skip connection), then perform layer normalization

4.3 class `TF_Encoder`

The class implements the entire Transformer encoder in Fig. 1. The class have two methods to be completed: `__init__` and `forward` which are **Q5** and **Q6**.

4.3.1 `__init__` method

The method has the following format:

```
__init__(self, vocab_size, d_model, d_ff, numlayer, numhead,
dropout):
```

Parameters:

- `vocab_size,` :*int*, size of vocabulary, i.e., the total number of words recognized by the model.

  `d_model` :*int*, size of feature vector in the Transformer.
- `d_ff` :*int*, size of feature vector in Feed Forward block.
- `numlayer` :*int*, number of `TF_Encoder_Block` in the encoder (N in Fig. 1)
- `numhead` :*int*, number of heads in multi-head attention.
- `dropout` :*float*, dropout probability.

In `__init__` method, you will need to create class objects such as

- `nn.Embedding` layer: maps input token to embedding (vector). This is already implemented
- `nn.Dropout` layer: dropout layer. This is already implemented
- N of `TF_Encoder_Block` layer: You need to instantiate `numlayer` number of sequential layers of `TF_Encoder_Block` as in Fig. 1.

4.3.2 `forward` method

The method computes the forward operation of the entire encoder (Fig.1).

`forward(self, x, src_batch_lens=None)`: Constructs an object for multi-head attention. Input parameters are as follows:

Parameters:

- `x` :*tensor*, a batch of input tokens having shape `(B,T_S)`. `B` is batch size, `T_S` is the sequence length of tokens. Regardless of the length of review words in each batch, each batch is padded to length `T_S`.
- `src_batch_lens` :Same as explained previously. You should pass `src_batch_lens` to your `MultiHeadAttention` object instantiated in this class.

Returns:

- `out`:*tensor*.

  Output tensor of shape `(B, T_S, d_model)`

Operation:

`forward` function should perform

1. Inputs batch of tokens x to embedding layer, i.e., maps each batch to `T_S` length vector.
2. Dropout is applied
3. Positional encoding is applied, and added together.
4. The resulting output is fed into N layers of encoder blocks. You need to implement this.

5 Dataset (Important!)

The IMDB dataset is a file named `imdb_dataset_dict.pt`. The file is uploaded as `.zip` file – it will be uncompressed about 130MB. Download the `.zip` file from the blackboard, uncompress it, and place this file into your local directory where your If you are running Colab, you may have to upload `imdb_dataset_dict.pt` to the directory where you run your notebook.

If you properly place the file and run the code, the train and test dataset objects will be instantiated: `train_dataset` and `test_dataset`. Also there are two helper dictionary objects `src_word_dict` and `src_idx_dict`, which converts token text to token index and vice versa. Search the code for these objects.

6 Training and Testing

You need the script for training and testing in `hw7.py`. Specifically, complete Q1–Q6. Determine proper hyperparameters like learning rate. **GPU usage is recommended for your training.** Otherwise training may be slow. In Colab environment, go to the menu `RunTime->Change runtime type`, and select GPU. **GPU quota in Colab is limited, so use it wisely.** As explained in class, build and debug your model on CPU first. If you believe your model is completed, train it on GPU.

7 What to submit

You will be given two files `hw7.py, imdb_dataset.pt`.

1. **You are asked to complete Q1–Q6 in `hw7.py`.**

   Submission instructions are

   - Submit modified Python file `hw7.py`.
   - Upload your files at Blackboard before deadline. (Please submit the file in time, no late submission will be accepted).

8 How your module will be graded

If done correctly, you will be able to achieve accuracy over 80% within 10 epochs of training with proper batch size and learning rate. **The goal is to achieve over 80% of test accuracy.** The test accuracy is the accuracy value printed on screen after

```
*** Now test phase begins!  ***
```

appears in the output window.

Before testing, your model will be trained until one of the following two events occur, whichever comes earlier.

1. Your model will be trained for at most **40** epochs.
2. Your model will be trained for **2 hours** with GPU similar to Google Colab environment.

So before submitting your model, please try to train your model on the standard Google Colab environment, and check whether your model can achieve over 80% test accuracy under 2 hours of training. **If your model is properly designed, it should take about 2–3 minutes per training of 1 epoch with CPU, and less than 1 min under GPU environment by Colab.**

9 Grading

- 25 points if your classifier achieves test accuracy more than 80%. **(Extra credit +3 points if your test accuracy is $\geq$ 85%.)**
- 8 points if your classifier achieves test accuracy between 65–80%.
- 2 points if your classifier achieves test accuracy below 65%.
- 0 point if you do not submit the file by deadline.

In the blackboard, you can upload your files as many times as you like, before the deadline. The last uploaded file will be used for grading. After deadline, the submission menu will be closed and you will not be able to make submission.