CSE 597 - Assignment 2
Transformers for POS Tagging

Overview

In this assignment, you will implement a Transformer encoder and apply it on a part-of-speech tagging task.

Submission Instruction

Follow these steps exactly, so the Gradescope autograder can grade your homework. Failure to do so will result in a zero grade:

1. You *must* download the homework template files from Canvas. Each template file is a python file that gives you a headstart in creating your homework python script with the correct function names for autograding.
2. Upload your transformer.py file to Gradescope by its due date.

Environment Setup

This assignment requires **Python 3.8**. The dependencies for this project include:

- **nltk** for loading and working with parse tree data structures
- **sentencepiece** for subword tokenization
- **torch** for modeling and training
- **tqdm** for displaying training progress

You can install these packages by

```
$ pip install --upgrade nltk sentencepiece torch tqdm
```

File Structure

The starter code contains the following

1. **model.py**: A baseline model for POS tagging.
2. **transformer.py**: **This is what you'll need to implement.** Some code is provided, and a baseline model is given in model.py, but important functionality is

not included. Please read the documentation in transformer.py and the instruction below carefully to understand what needs to be done.

3. **pos_tagger.py**: data processing and training code for POS tagger models.
4. **data**/: Dataset of Penn Treebank including three files for three splits: **train**, **dev**, **test**.

Data

We use the standard Penn Treebank data splits for parsing: sections 2-21 are used for training, section 22 for validation, and section 23 for testing. Each line is a constituency parsing tree, but we will only use POS tags for this assignment. You can take a look at the format of the data:

```
$ head -n 2 data/train

(TOP (S (PP (IN In) (NP (NP (DT an) (NNP Oct.) (CD 19) (NN review)) (PP (IN of) (NP (`` ``) (NP (DT The) (NN Misanthrope)) ('' '') (PP (IN at) (NP (NP (NNP Chicago) (POS 's)) (NNP Goodman) (NNP Theatre))))) (PRN (-LRB- -LRB-) (`` ``) (S (NP (VBN Revitalized) (NNS Classics)) (VP (VBP Take) (NP (DT the) (NN Stage)) (PP (IN in) (NP (NNP Windy) (NNP City))))) (, ,) ('' '') (NP (NN Leisure) (CC &) (NNS Arts)) (-RRB- -RRB-)))) (, ,) (NP (NP (NP (DT the) (NN role)) (PP (IN of) (NP (NNP Celimene)))) (, ,) (VP (VBN played) (PP (IN by) (NP (NNP Kim) (NNP Cattrall)))) (, ,)) (VP (VBD was) (VP (ADVP (RB mistakenly)) (VBN attributed) (PP (TO to) (NP (NNP Christina) (NNP Haag))))) (. .)))
(TOP (S (NP (NNP Ms.) (NNP Haag)) (VP (VBZ plays) (NP (NNP Elianti))) (. .)))
```

Part-of-Speech Tagging

In this task, we will label each word token in a sentence or corpus with a part-of-speech tag. We provide you with all of the data processing code and a baseline model. You can run the data preprocessing and the baseline model by

```
$ python pos_tagger.py --model=baseline
```

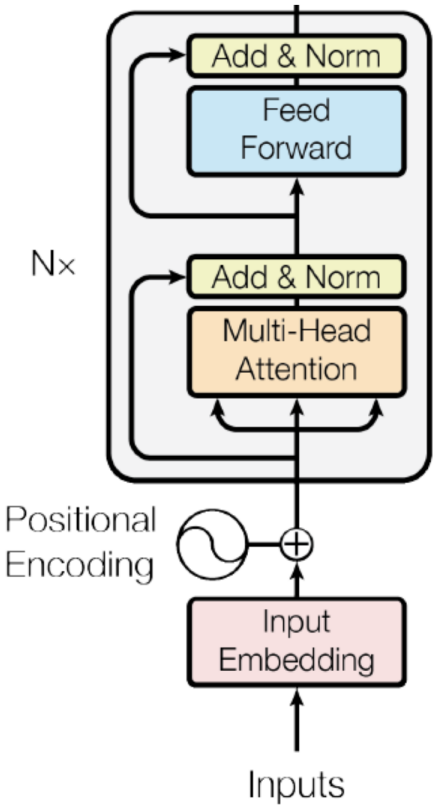
After the training finishes, you should be able to get **87.6** accuracy on the dev set as below.

```
epoch 1: 100%|████████████████████████████████████████████████| 575/575
[00:06<00:00, 89.39batch/s, mean_loss=0.77]
```

```
Obtained a new best validation metric of 0.873, saving model checkpoint to
baseline_model.pt...h/s, mean_loss=0.77]
epoch 2: 100%|██████████████████████████████████████████████| 575/575
[00:06<00:00, 91.63batch/s, mean_loss=0.364]
Obtained a new best validation metric of 0.875, saving model checkpoint to
baseline_model.pt.../s, mean_loss=0.364]
epoch 3: 100%|██████████████████████████████████████████████| 575/575
[00:06<00:00, 89.20batch/s, mean_loss=0.349]
Obtained a new best validation metric of 0.875, saving model checkpoint to
baseline_model.pt.../s, mean_loss=0.349]
epoch 4: 100%|██████████████████████████████████████████████| 575/575
[00:06<00:00, 87.10batch/s, mean_loss=0.342]
epoch 5: 100%|██████████████████████████████████████████████| 575/575
[00:06<00:00, 91.34batch/s, mean_loss=0.337]
Obtained a new best validation metric of 0.876, saving model checkpoint to
baseline_model.pt.../s, mean_loss=0.337]
training:
100%|████████████████████████████████████████████████████████| 5/5
[00:35<00:00,  7.13s/epoch]
Reloading best model checkpoint from baseline_model.pt...
```

Your Tasks

Your task is to implement the Transformer architecture (https://arxiv.org/pdf/1706.03762.pdf) and apply it to tagging. Here is a diagram of the architecture you will implement:



This portion is referred to as the "Transformer Encoder". In the paper there is also a decoder portion for generating text one token at a time; such a decoder is not needed for this project. The key elements of the Transformer are a multi-headed attention mechanism, and a feed-forward layer. Each sub-layer (whether multi-head attention or feed forward) uses a residual connection followed by Layer Normalization (nn.LayerNorm in pytorch). Both residual connections and normalizations are crucial to being able to train a model that's more than a couple layers deep.

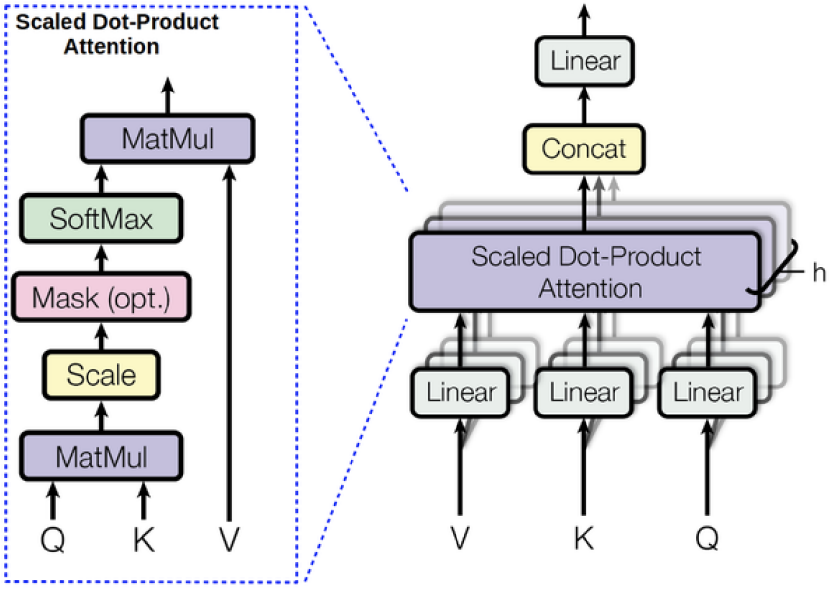
You are going to implement the following classes in `transformer.py`:

- `class MultiHeadAttention(nn.Module)`
- `class PositionwiseFeedForward(nn.Module)`
- `class TransformerEncoder(nn.Module)`
- `class TransformerPOSTaggingModel(POSTaggingModel)`

1. The first part is multi-head self-attention in `class MultiHeadAttention(nn.Module)`. In this layer, you will need to:

- Apply linear projections to convert the feature vector at each token into separate vectors for the query, key, and value.

- Apply attention, scaling the logits by $1/\sqrt{d_{kqv}}$.
- Ensure proper masking, such that padding tokens are never attended to.
- Perform attention n_head times in parallel, where the results are concatenated and then projected using a linear layer.



You should include two types of dropout in your code (with probability set by the dropout argument):

- Dropout should be applied to the output of the attention layer (just prior to the residual connection, denoted by "Add & Norm" in the first figure)
- Dropout should also be applied to attention probabilities, right after the softmax operation that's applied to query-key dot products. This type of dropout stochastically prevents a query position from attending to a fraction of key positions, which can help generalization. (Note that the probabilities will no longer sum to 1, but that's okay - they will still have an expectation of 1 due to PyTorch's dropout rescaling)

Notes:

- Query, key, and value vectors should have shape `[batch_size, n_heads, sequence_len, d_qkv]`
- Attention logits and probabilities should have shape `[batch_size, n_heads, sequence_len, sequence_len]`
- Vaswani et al. define the output of the attention layer as concatenating the various heads and then multiplying by a matrix W^O. It's also possible to implement this is a sum without ever calling `torch.cat`: note that

$$\text{Concat}(head_1, \ldots, head_h)W^O = head_1 W_1^O + \ldots + head_h W_h^O \text{ where } W^O = \begin{bmatrix} W_1^O \\ \vdots \\ W_h^O \end{bmatrix}$$

2. The other component is the position-wise feed forward layer in `class PositionwiseFeedForward(nn.Module)`. This layer's architecture is sometimes called dense-relu-dense, because it consists of two dense linear layers with ReLU nonlinearity in the middle. The dropout here is typically applied at the output of the layer instead of next to the non-linearity in the middle.

3. Combining the two gives the full transformer encoder architecture `class TransformerEncoder(nn.Module)`.

4. Unlike with recurrent neural networks, word order is not encoded in the Transformer architecture directly. Instead, positions of words are provided in the form of position embeddings that are added to the feature vector of each word. The exact formulation of the position embeddings tends to be implementation-dependent, and a number of approaches have been proposed in the literature. The `class AddPositionalEncoding(nn.Module)` provides a version of positional encoding. We have provided the implementation for this.

5. Finally, complete `class TransformerPOSTaggingModel(POSTaggingModel)`. Your goal is to achieve a target of 95% accuracy, or higher, on the validation set.

After you finish your implementation, you can run your model by

```
$ python pos_tagger.py --model=transformer
```

After the training finishes, you should be able to get **95.7** accuracy on the dev set.