CS542 (Fall 2023) Programming Assignment 5
Multi-head Attention from Scratch

Due December 3, 2023

Introduction

In this assignment, you'll be implementing the multi-head attention mechanism and applying it to some data to examine the effects of self-attention. We will be building a small BERT-style Transformer that does masked language modeling (MLM). We should not expect great performance out of this Transformer because we are only using a small amount of data, it will only have 3 encoder and decoder layers, and we'll only do a little bit of training, in order to keep things tractable. However, even with this small model we can see some interesting properties of Transformers appear.

Setup

Assuming you already have PyTorch installed, you will also need to have the spaCy and Torchtext packages installed. Run `pip install spacy` in your local environment before getting started, and then run `python -m spacy download en_core_web_sm` to download the tokenizer model. Then, run (`pip install torchtext`).

Assignment

Most of the code has been written for you. You are given `attention_mechanism.py` and supporting classes. This is small BERT-style Transformer that we will train over a subset of the WikiText2 dataset (also provided). The only files you will need to edit for this assignment are `attention_mechanism.py` and `multihead_attention_layer.py`. Your job is to complete the implementation of the multi-head attention mechanism. You will also need to need to implement a couple of helper functions to help you explore the data and the embedding space of your trained model.

Specifically, you'll need to:

- Complete the implementation of `apply_attention` (in `multihead_attention_layer.py`) by calculating the attention weights from the query and key matrices and applying them to the value matrix. This is effectively calculating:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{n}})V$$

  It is broken down into a couple of steps for you due to the introduction of masking and dropout that you aren't responsible for. Complete the calculations as described in the comments in `apply_attention`. You will need to make sure that the shapes of the matrices are correct so they can be multiplied together. This may involve transposing, permuting, squeezing, or unsqueezing. You should familiarize yourself with the relevant PyTorch/Numpy functions that will do these.

- Implement the `cosine_sim` function (in `attention_mechanism.py`) that takes in two vectors `a` and `b` and returns their cosine similarity. Do not use a package's version of this function (i.e., do not call `sklearn`). You need to implement this yourself.

- Implement a `where_is` function (in `attention_mechanism.py`) that will help you identify specific sentences where specific words occur in the data (and where in those sentences those words occur). A version of this function exists in Lecture 26; it takes in a dictionary of sentences and a word of interest. The function should return a dictionary that consists of keys that are sentence indices and values that are lists containing where in the corresponding sentence the word of interest occurs. See the autograder code (test #3) for reference. The correct output for `where_is("President",sentence_dict)` is {`0: [0, 11], 1: []`} because "President" occurs at indices 0 and 11 in sentence 0 and doesn't occur in sentence 1. It may be helpful to filter out dictionary entries where the value list is empty. Both types of implementations should be accepted by the autograder. For instance:

```
word = "Meridian"
print([{k:self.where_is(word,self.val_sentence_ids)[k]}
for k in self.where_is(word,self.val_sentence_ids) if
self.where_is(word,self.val_sentence_ids)[k]])
```

  will print out where in the validation data the word "Meridian" exists, excluding sentences where it is absent.

- Implement `check_embeddings_val` (in `attention_mechanism.py`). You should use your `where_is` function to identify at least 10 words of interest in the validation data. In this function, you should print out information comparing instances of these words to other words based on their cosine similarity. You should examine: 1) different instances of the same focus word within a single sentence, 2) different instances of the same focus word in different sentences, and 3) and instances of the focus word compared to other words (these can be randomly chosen, within the same sentence as the focus word, or in other sentences). This function is called after each training epoch, so you can examine how the embeddings of your words of interest change during training.

- Implement `check_embeddings_test` (in `attention_mechanism.py`). You should use your `where_is` function to identify at least 10 words of interest in the *test* data. This is the same as `check_embeddings_val` but is called only once, after training completes. The words you examine here need to come from the test data but other than that the implementation can follow that of `check_embeddings_val`.

At various points in the code you will see comments of the format: `#Q = [batch size, query len, hid dim]`. These indicate what the shape of that variable should be at this point in the code. Use them to help debug, but know that sometimes the shapes of your matrices may include an extra axis.

The provided code should be able to be run on a GPU, which will help speed-wise, but you can also complete the assignment with only a CPU, because we've trimmed the data down to a tractable size and only train for 8 epochs.

Submission

Submit your completed `attention_mechanism.py` and `multihead_attention_layer.py` files, and a PDF containing your writeup and observation (described below).

Grading

Grades will be determined as follows:

- The automated grader is provided to you run the full battery of sanity checks on your code. These tests are non-exhaustive, so just because your code passes these sanity checks doesn't mean it is bug free or fully correct. Therefore simply passing the autograder is not enough! There are 6 sanity checks run though the autograder that apportion points as outlined below:

    1. **5 points** for loading your code error-free. If you do not change the template code outside of the prescribed areas, this should be no problem.
    2. **5 points** for a correct `cosine_sim` implementation.
    3. **5 points** for a correct `where_is` implementation (correct meaning that it returns the right values *and* in the right format).
    4. **35 points** for correctly completing the `apply_attention` function.
    5. **10 points** for examining the embeddings of at least 10 words from the validation data in the `check_embeddings_val` function.
    6. **10 points** for examining the embeddings of at least 10 words from the test data in the `check_embeddings_test` function. Though there may be some overlap, full credit will not be given if these 10 words are identical to those examined in `check_embeddings_val`.

- The final **30 points** come from the write-up, where you should discuss your observations regarding the behavior of the embeddings of the words you chose to focus on. Some things you can consider looking at:

    – How do words in the same sentence compare vs. words in different sentences?

    – How do the cosine similarities between individual word embeddings change as the model trains?

    – What differences do you observe in the embeddings of the validation data vs. the test data? You will be able to make better comparisons if you look for similar but not identical words of interest in the validation data vs. the test data.

    – Do you notice any patterns across different kinds of words (e.g., proper vs. common nouns, different forms of the same lemma, or different parts of speech)? You should choose your words of interest with an eye toward answering these questions.

Extra Credit

If time permits, you can try any of the following for additional credit. Be sure to include sufficient discussion of these experiments in the writeup. You can earn up to 20 points extra credit.

- Train using more data. Change the value of `MAX_LINES` in `main` to a larger number. How does more data affect the validation embeddings during training and the test embeddings after training? (5 points)
- Train for more epochs. Change the value of `N_EPOCHS` in `main`. How does more training affect the validation embeddings during training and the test embeddings after training? (5 points)
- The provided code draws embeddings from the final encoder layer. You can also draw embeddings from one of the other encoder layers. Modify the code in `evaluate` to draw embeddings from a different encoder layer (or multiple layers) and examine how the embeddings differ between encoder layers. (10 points)