CS685 Homework 2: Implementing a Transformer (Due April 8th)

Goals: The primary goal of this assignment is to give you hands-on experience implementing a Transformer encoder. Understanding how these neural models work and building one from scratch will help you understand not just language modeling, but also systems for many other applications such as machine translation.

Installation

- The list of installed packages in the autograder is: `numpy`, `nltk`, `spacy`, `torch`, `scipy`, `matplotlib`, `torchvision`.

- Please use Python 3.5+ and a recent version of PyTorch for this project. You should follow the instructions at https://pytorch.org/get-started/locally/ to install PyTorch on your machine. The assignment is small-scale enough to complete using CPU only, so don't worry about installing CUDA and getting GPU support working unless you want to.

- Installing via Anaconda is typically easiest, especially if you are on OS X, where the system Python has some weird package versions. Installing in a virtual environment is recommended but not essential.

Dataset and Code

Data The dataset for this homework is derived from the text8 collection, which comes from Wikipedia. Your method will use *character-level* tokenization and operate over text8 sequences that are each exactly 20 characters long. Only 27 character types are present (lowercase characters and spaces); special characters are replaced by a single space and numbers are spelled out as individual digits (*50* becomes *five zero*).

Framework code: The framework code you are given consists of several files. We will describe these in the following sections. `utils.py` implements an Indexer class, which can be used to maintain a bijective mapping between indices and features (strings). `letter_counting.py` contains the driver code, which imports `transformer.py`, the file you will be editing for this assignment.

Building a Transformer Encoder

You will implement a simplified Transformer (missing components like layer normalization and multi-head attention) from scratch for a simple task. **Given a string of characters, your task is to predict, for each position in the string, how many times the character at that position occurred previously, maxing out at 2.** This is a 3-class classification task (with labels 0, 1, or > 2, which we'll just denote as 2). This task is easy with a rule-based system, but it is not so easy for a model to learn. However, Transformers are ideally set up to be able to "look back" with self-attention to count occurrences in the context. Below is an example string (which ends in a trailing space) and its corresponding labels:

```
i like movies a lot
00010010002102021102
```

We also present a modified version of this task that counts both occurrences of letters before *and after* in the sequence:

```
i like movies a lot
22120120102102021102
```

Note that every letter of the same type always receives the same label, no matter where it is in the sentence in this version. Adding the `-task BEFOREAFTER` flag will run this second version; default is the first version.

`lettercounting-train.txt` and `lettercounting-dev.txt` both contain character strings of length 20. **You can assume that your model will always see 20 characters as input and make a prediction at each position in the sequence.**

Getting started   Run:

```
python letter_counting.py --task BEFOREAFTER
python letter_counting.py --task BEFORE
```

This loads the data but will fail out because the Transformer hasn't been implemented yet. (We didn't bother to include a rule-based implementation because it will always just get 100%.)

Implementation   Implement Transformer and TransformerLayer. You should identify the number of other letters of the same type in the sequence. This will require implementing both Transformer and TransformerLayer, as well as training in `train_classifier`.

Your solutions **should not** use `nn.TransformerEncoder`, `nn.TransformerDecoder`, or any other off-the-shelf self-attention layers. You can use `nn.Linear`, `nn.Embedding`, and PyTorch's provided nonlinearities / loss functions to implement Transformers from scratch.

TransformerLayer   This layer should contain the following components in order:

1. self-attention (single-headed is fine; you can use either masked or unmasked attention)

2. residual connection

3. two linear layers, the first with a non-linearity (e.g., ReLU)

4. final residual connection

You do not need to implement layer normalization (a component of the Transformer which we did not discuss in class) for this assignment. Because this task is relatively simple, you don't need a very well-tuned architecture to make this work. You will implement all of these components from scratch.

You will want to form queries, keys, and values matrices with linear layers, then use the queries and keys to compute attention over the sentence, then combine with the values. You'll want to use `matmul` for this purpose, and you may need to transpose matrices as well. Double-check your dimensions and make sure everything is happening over the correct dimension. Furthermore, dividing your dot products by $\sqrt{d_k}$ as in Vaswani et al. (2017)'s attention paper may help stabilize and improve training.

PositionalEncoding   We provide a `PositionalEncoding` module that initializes a `nn.Embedding` layer and embeds the *index* of each character[1] We intend that you to add

---

[1]The drawback of this in general is that your Transformer cannot generalizes to longer sequences at test time, but this is not a problem here where all of the train and test examples are the same length. If you want, you can explore the sinusoidal embedding scheme from Attention Is All You Need (Vaswani et al., 2017), but this is a bit more finicky to get working.

these position embeddings to the character embeddings when building the full Transformer in the next step. To be clear, if the input sequence is the, then the embedding of the first token would be $\mathrm{embed}_{\mathrm{char}}(\mathrm{t}) + \mathrm{embed}_{\mathrm{pos}}(0)$, and the embedding of the second token would be $\mathrm{embed}_{\mathrm{char}}(\mathrm{h}) + \mathrm{embed}_{\mathrm{pos}}(1)$.

Transformer Building the Transformer will involve: (1) adding positional encodings to the input (see the `PositionalEncoding` class); (2) using one or more of your TransformerLayers; and (3) using Linear and softmax layers to make the prediction. You will simultaneously be making predictions over each position in the sequence. Your network should return the log probabilities at the output layer (a 20x3 matrix) as well as the attentions you compute, which are then plotted for you for visualization purposes in `plots/`.

Training A skeleton for training is provided in `train_classifier`. We have already formed input/output tensors inside `LetterCountingExample`, so you can use these as your inputs and outputs. Note that you will need to make simultaneous predictions at all time steps and accumulate losses over all of them simultaneously. NLLLoss can help with computing a "bulk" loss over the entire sequence.

Grading Your final implementation should get **over 90% accuracy** on both the BEFOREAFTER and BEFORE tasks. If your implementation achieves below 90% on either task, you will lose 2 points for each absolute percentage point below 90%. For example, if you get 88% on BEFORE-AFTER and 80% on BEFORE, your homework score will be 76/100. Our reference implementation achieves over 98% accuracy on both tasks in 5-10 epochs of training. It takes about 20 seconds per epoch using 1-2 single-head Transformer layers (there is some variance and it can depend on initialization). Also note that **the autograder trains your model on an additional task as well.**, so do not hardcode anything about these labels (or attempt to cheat by returning the correct answer by directly counting letters yourself). Any Transformer-based implementation that works for BEFORE and BEFOREAFTER will also work for the hidden task.

Debugging Tips As always, make sure you can overfit a very small training set as an initial test, inspecting the loss of the training set at each epoch. You will need your learning rate set carefully to let your model train. Even with a good learning rate, it will take longer to overfit data with this model than with others we've explored! Then scale up to train on more data and check the development performance of your model. Calling `decode` inside the training loop and looking at the attention visualizations can help you reason about what your model is learning and see whether its predictions are becoming more accurate or not.

If everything is stuck around 70%, you may not be successfully training your layers, which can happen if you attempt to initialize layers inside a Python list; these layers will not be "detected" by PyTorch and their weights will not be updated during learning.

Consider using small values for hyperparameters so things train quickly. In particular, with only 27 characters, you can get away with small embedding sizes for these, and small hidden sizes for the Transformer (100 or less) may work better than you think!

Exploration *(optional, you do not need to submit anything for these questions)*

1. Look at the attention masks produced. What is the model doing? Does it match your expectations?

2. Try using more Transformer layers (3-4). Do all of the attention masks fit the pattern you expect?

Deliverables and Submission

You will need to upload **two files** to Gradescope to receive full credit for this assignment: `transformer.py` and the AI disclosure `ai_disclosure.txt`. Make sure that the following command works before you submit:

`python letter_counting.py`

Your AI disclosure should be formatted as specified in the Academic Honesty section below.

Academic Honesty

- We run automatic checks of Python files for plagiarism. Copying code from others is considered a serious case of cheating.

- There is no penalty for using AI assistance on this homework as long as you submit AI disclosure (this includes storing any prompts that you feed to large language models). That said, anyone caught using AI assistance without proper disclosure will receive a zero on the assignment (we have several automatic tools to detect such cases). We're literally allowing you to use it with no limitations, so there is no reason to lie!

- When submitting your solution to Gradescope, please include a text file (ai_disclosure.txt) together with the Python code. The file should contain answers to the following questions:

  1. Did you use any AI assistance to complete this homework? If so, please also specify what AI you used.
  2. If you used a large language model to assist you, please paste all of the prompts that you used below. Add a separate bullet for each prompt, and specify which problem is associated with which prompt.
  3. Free response: For each problem for which you used assistance, describe your overall experience with the AI. How helpful was it? Did it just directly give you a good answer, or did you have to edit it? Was its output ever obviously wrong or irrelevant? Did you use it to get the answer or check your own answer?

Acknowledgment

This homework is adapted from CS388: Natural Language Processing, taught by Greg Durrett at the University of Texas at Austin.

References

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *arXiv*.