Assignment 4: Transformer Language Modeling

Due Date: March 30th (Thursday) 11:59 CST

Academic Honesty: Pl[illegible] syllabus for information about collaboration in this course. While you may discuss [illegible]her students, **all work you submit must be your own!** You should not use AI helpe[illegible]ework.

Goals The primary g[illegible]t is to give you hands-on experience implementing a Transformer language model[illegible]hese neural models work and building one from scratch will help you understand n[illegible]ng, but also systems for many other applications that build upon it!

Dataset and Code

Please use up-to-date versions of Python and PyTorch for this assignment. See Assignment 2 for installation instructions for PyTorch.

Data The dataset for this paper is the `text8`[1] collection. This is a dataset taken from the first 100M characters of Wikipedia. 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 (*20* becomes *two zero*). A larger version of this benchmark (90M training characters, 5M dev, 5M test) was used in Mikolov et al. (2012). We will be splitting these into sequences of length 20 for Part 1.

Framework code The framework code you are given consists of several files. We will describe these in the following sections. `utils.py` should be familiar to you by now. `letter_counting.py` contains the driver for Part 1, which imports `transformer.py`. `lm.py` contains the driver for Part 2 and imports `transformer_lm.py`.

Part 1: Building a "Transformer" Encoder (50 points)

In this first part, 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 before, 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:

input	i		l	i	k	e		m	o	v	i	e	s		a		l	o	t	
label	0	0	0	1	0	0	1	0	0	0	2	1	0	2	0	2	1	1	0	2

We also present a modified version of this task that counts both occurrences of letters before *and after* in the sequence: Note that every letter of the same type always receives the same label no matter where it is in

[1] Original site: `http://mattmahoney.net/dc`

input	i		l	i	k	e		m	o	v	i	e	s		a		l	o	t	
label	2	2	1	2	0	1	2	0	1	0	2	1	0	2	0	2	1	1	0	2

the sentence in this version. Adding the `--task BEFOREAFTER` flag will run this second version; default is the first version.

`lettercounting`[illegible]`tercounting-dev.txt` both contain character strings of length 20. **You can** [illegible]**del will always see 20 characters as input.** Different from Assignment 2, you nee[illegible]t each position in the sequence.

Getting started Run:

`python letter_co`[illegible]`k BEFOREAFTER`

This loads the data for this part, 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%.)

Q0 (not graded) Implement Transformer and TransformerLayer for the BEFOREAFTER version of the task. 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 Part 1 solutions **should not** use `nn.TransformerEncoder`, `nn.TransformerDecoder`, or any other off-the-shelf self-attention layers. You should only use Linear, softmax, and standard nonlinearities to implement Transformers from scratch.

TransformerLayer This layer should follow the format discussed in class: (1) self-attention (single-headed is fine; you can use either backward-only or bidirectional attention); (2) residual connection; (3) Linear layer, nonlinearity, and Linear layer; (4) final residual connection. With a shallow network like this, you likely don't need layer normalization, which is a bit more complicated to implement. You don't have to scale your attention either. 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.

Transformer Building the Transformer will involve: (1) adding positional encodings to the input (see the `PositionalEncoding` class; but we recommend leaving these out for now) (2) using one or more of your TransformerLayers; (3) using Linear and softmax layers to make the prediction. Different from Assignment 2, you are simultaneously 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 follows previous assignments. A skeleton is provided in `train_classifier`. We have already formed input/output tensors inside `LetterCountingExample`, so you can use these as your inputs and outputs. Whatever training code you used for Assignment 2 should likely work here too, with the major change being the need to make simultaneous predictions at all timesteps and accumulate losses over all of them simultaneously. NLLLoss can help with computing a "bulk" loss over the entire sequence.

Without positional encodings, your model may struggle a bit, but you should be able to get at least 85% accuracy with a single-layer Transformer in a few epochs of training. The attention masks should also show some evidence of the model attending to the characters in context.

Q1 (40 points, autogr[illegible]ur Transformer classifier with positional encodings and address the main task: ide[illegible] letters of the same type **preceding** that letter. Run this with `python letter_co`[illegible] arguments. Without positional encodings, the model simply sees a bag of character[illegible]h letters occurring later or earlier in the sentence (although loss will still decrease a[illegible]be learned).

We provide a `Posi`[illegible]odule that you can use: this initializes a `nn.Embedding` layer, embeds the *index*[illegible] adds these to the actual character embeddings.[2] If the input sequence is `the`, then [illegible] first token would be $\mathrm{embed}_{\mathrm{char}}(\mathrm{t}) + \mathrm{embed}_{\mathrm{pos}}(0)$, and the embedding of the second token would be $\mathrm{embed}_{\mathrm{char}}(\mathrm{h}) + \mathrm{embed}_{\mathrm{pos}}(1)$.

Your final implementation should get **over 95% accuracy** on this task. **Our reference implementation achieves over 98% accuracy in 5-10 epochs of training taking 20 seconds each 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.** You will fail this hidden test if your model uses anything hardcoded about these labels (or if you try to cheat and just return the correct answer that you computed by directly counting letters yourself), but any implementation that works for this problem will work for the hidden test.

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.

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!

Q2 (5 points) Look at the attention masks produced. Include at least one attention chart in your writeup. Describe in 1-3 sentences what you see here, including what it looks like the model is doing and whether this matches your expectation for how it should work.

Q3 (5 points) Try using more Transformer layers (3-4). Do all of the attention masks fit the pattern you expect? Describe in 1-3 sentences what you see in the "less clear" attention masks.

[2] The drawback of learning absolute position embedding is that your Transformer cannot generalize 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.

Part 2: Transformer Decoder for Language Modeling (50 points)

In this second part, you will implement a Transformer language model. This should build heavily off of what you did for Part 1, although for this part you are allowed to use off-the-shelf Transformer components (nn.TransformerEncoder).

For this part, we use [illegible]cters of `text8` as the training set. The development set is 500 characters taken fr[illegible]ection. Your model will need to be able to consume a chunk of characters and make [illegible] character at each position simultaneously. Structurally, this looks exactly like Q1, a[illegible] classes instead of 3.

Getting started Run: [illegible]

```
python lm.py
```

This loads the data, instantiates a `UniformLanguageModel` which assigns each character an equal $\frac{1}{27}$ probability, and evaluates it on the development set. This model achieves a total log probability of -1644, an average log probability (per token) of -3.296, and a perplexity of 27. Note that exponentiating the average log probability gives you $\frac{1}{27}$ in this case, which is the inverse of perplexity.

The `NeuralLanguageModel` class you are given has two methods: `get_next_char_log_probs` and `get_log_prob_sequence`. The first takes a context and returns the log probability distribution for the next character. This output will be a numpy vector of length equal to the vocabulary size. The second takes a whole sequence of characters and a context and returns the log probability of that whole sequence under the model. The output will be a matrix of dimension sequence length by vocabulary size. You can implement the second just using the first, but that's inefficient; you can instead just run a single pass through the Transformer and return the aggregated log probability of the sequence. (Because this looks the same as training, you can use `NLLLoss` for this purpose as well).

Q4 (50 points, autograded) Implement a Transformer language model (transformer_lm.py). This will require: defining a PyTorch module to handle language model prediction, implementing training of that module in `train_lm`, and finally completing the definition of `NeuralLanguageModel` appropriately to use this module for prediction. Your network should take a chunk of indexed characters as input, embed them, put them through a Transformer, and make predictions from the final layer outputs.

Your final model must **pass the sanity and normalization checks, get a perplexity value less than or equal to 7, and train in less than 10 minutes.** Our Transformer reference implementation gets a perplexity of 6.3 in about 6 minutes of training. However, this is an unoptimized, unbatched implementation and you can likely do better.

Network structure You can use a similar input layer (Embedding followed by PositionalEncoding) as in Part 1 to encode the character indices. You can use the PositionalEncoding from Part 1. You can then use your Transformer architecture from Part 1 or you can use a real `nn.TransformerEncoder`,[3] which is made up of TransformerEncoderLayers.

Note that unlike the Transformer encoder you used in part 1, for Part 2 you must be careful to use a **causal mask** for the attention: tokens should not be able to attend to tokens occurring after them in the sentence, or else the model can easily "cheat" (consider that if token n attends to token $n+1$, the model can store the identity of token $n+1$ in the nth position and predict it at the output layer). Fortunately it should be very easy to spot this, as your perplexity will get very close to 1 very quickly and you will fail the sanity check.

[3] `https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoder.html`

You can use the `mask` argument in `TransformerEncoder` and pass in a triangular matrix of zeros / negative infinities to prevent this.

Training on chunks Unlike in Part 1, you are presented with data in a long, continuous stream of characters. Nevertheless, your [illegible]ss a chunk of characters at a time, simultaneously predicting the next character at eac[illegible]ou'll have to decide how you want to chunk the data for both training and inference.

Start of sequence In [illegible] of any sequence is represented to the language model by a special start-of-sequenc[illegible]**, we are going to overload space and use that as the start-of-sequence character**[illegible]chunk of 20 characters, you want to feed space plus the first 19 into the model and p[illegible].

input		i		l	i	k	e		m	o	v	i	e	s		a		l	o	t
label	i		l	i	k	e		m	o	v	i	e	s		a		l	o	t	

Evaluation Unlike past assignments where you are evaluated on correctness of predictions, in this case your model is evaluated on perplexity and likelihood, which rely on the probabilities that your model returns. **Your model must be a "correct" implementation of a language model.** Correct in this case means that it must represent a probability distribution $P(w_i|w_1, \ldots, w_{i-1})$. You should be sure to check that your model's output is indeed a legal probability distribution over the next word.

Batching Batching across multiple sequences can further increase the speed of training. While you do not need to do this to complete the assignment, you may find the speedups helpful. As in Assignment 2, you should be able to do this by increasing the dimension of your tensors by 1, a batch dimension which should be the first dimension of each tensor. The rest of your code should be largely unchanged. Note that you only need to apply batching during training, as the two inference methods you'll implement aren't set up to pass you batched data anyway.

Tensor manipulation `np.asarray` can convert lists into numpy arrays easily. `torch.from_numpy` can convert numpy arrays into PyTorch tensors. `torch.FloatTensor(list)` can convert from lists directly to PyTorch tensors. `.float()` and `.int()` can be used to cast tensors to different types. `unsqueeze` allows you to add trivial dimensions of size 1, and `squeeze` lets you remove these.

Deliverables and Submission

You will upload your code for Part 1 and Part 2 on Gradescope in two separate files (`transformer.py` and `transformer_lm.py`).

Make sure that the following commands work (for Parts 1 and 2, respectively) before you submit and you pass the sanity and normalization checks for `lm.py`:

```
python letter_counting.py
```

```
python lm.py --model NEURAL
```

These commands should run without error and train in the allotted time limits. **The homework was originally developed by Greg Durrett.**

References

Tomas Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocký. 2012. Subword Language Modeling with Neural Networks. In *Online preprint*.

Ashish Vaswani, Noam S[illegible]ob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. [illegible]d. In *arXiv*.