AU22 CSE 5525 Homework #4: Character Language Modeling with RNNs and Transformer

Deadline: **11:59PM on 12/05/2022**

Academic Integrity

You are free to discuss the homework assignments with other students and study course materials together towards solutions. **However, all of the code and report you write must be your own!** If you do discuss or study together with others (except via posts/comments on Teams), please list their names at the top of your written submission. Unless there is academic misconduct detected, this won't be counted against your credit.

Plagiarism will be automatically detected on Carmen, e.g., by comparing your HW with all resources, as well as manually checked by TA. If the instructor or TA suspects that a student has committed academic misconduct in this course, they are obligated by university rules to report their suspicions to the Committee on Academic Misconduct. Please see more in Statements for CSE5525 on Carmen pages.

Goals

In this project, you'll implement a character neural language model with two sequential models, Recurrent Neural Networks (RNNs) and the Transformer model [2]. A simple code framework for data loading and evaluation of language models is provided. You will have to figure out how to implement and train the language models with both RNNs and Transformer.

The goal is to be familiar with the Transformer architecture by implementing every component and practicing the basics of language modeling (data, training, evaluation, etc). You should also realize the importance of hyperparameter tuning (with your experience in implementing and training Seq2seq models). You will continue your journey of PyTorch and you can call existing functions in Pytorch, as long as the required steps are shown.

Timeline & Credit

You will have around 3 weeks to work on this programming assignment. Late submissions will NOT be accepted and you also need to work on the final project for the rest of the semester. So, start early and plan ahead! We currently use a 100-point scale for this homework (Part 1: 40 points; part 2: 40 points; 1-2 page report: 20 points), but it will take 15% of your final grade.

Questions?

Please create a post on MS Teams to get timely help from other students, the TA, and the instructor. Remember that participation takes 5% of your final grade. You can show your participation by actively answering others' questions and online class participation! **Besides, everyone can benefit from checking what has been asked previously.** Feel free to join the TA or the instructor's office hours as well (Time and Zoom links can be found on Carmen).

Dataset and Code

Please use Python 3.5+ and PyTorch 1.0+ for this project.

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 [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.

`lm.py` contains the driver for this homework and calls training functions from `train.py` on the raw text data. `models.py` contains skeletons in which you will implement these models.

Part 1: Implementing an RNN Language Model

In the first part, you will implement an RNN language model. This should build heavily off of what you did for HW#3, though new ingredients will be necessary, particularly during training.

Data For this part, we use the first 100,000 characters of `text8` as the training set. The development set is 500 characters taken from elsewhere in the collection.

Getting started

```
python lm.py
```

This loads the data and 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 `RNNLanguageModel` class you are given in `models.py` has two methods: `get_next_char_log_probs` and `get_log_prob_sequence`. The first takes a context and returns the log probability distribution over the next characters given that context as 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. You can implement the second just using the first, but that's computationally wasteful; you can instead just run a single pass through the RNN and return the aggregated log probability of the sequence.

Q1 (40 points) Implement an RNN language model. 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 `RNNLanguageModel` appropriately to use this module for prediction. Your network should take a chunk of indexed characters as input, embed them, put them through an RNN, and make predictions from the final layer outputs.

Note that you should make next character predictions *simultaneously* at every position in the sequence you feed in. Here, you're making a set of predictions for each sample. You'll want to use the `outputs` field in PyTorch, which returns the outputs (same as the hidden states in an LSTM) at each position in the RNN. You should be very familiar with RNNs after doing HW#3.

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.**[2]

Chunking the data Unlike classification, language modeling can be viewed as a task where the same network is predicting words at many positions. Your network should process a chunk of characters at a time, simultaneously predicting the next character at each index in the chunk. You'll have to decide how you want to chunk the data. Given a chunk, you can either initialize the RNN state with a zero vector, "burn in" the RNN by running on a few characters before you begin predicting, or carry over the end state of the RNN to the next chunk. These may only make minor differences, though.

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

[2] Our reference implementation gets a perplexity of 6.2 in about 2 minutes of training or 5.44 in about 5 minutes of training. However, this is an unoptimized, unbatched implementation and you can likely do better.

Start of sequence In general, the beginning of any sequence is represented in the language model by a special start-of-sequence token. **For simplicity, we are going to overload space and use that as the start-of-sequence character.** That is, space will both be the first token fed to the encoder as well as a token that is predicted in the output space. While this reduces the number of parameters in the model compared to having a separate start-of-sequence token, it doesn't impact the performance much.

Evaluation Unlike past assignments where you are evaluated on the correctness of predictions, in this case, your model is evaluated on the perplexity and likelihood, which rely on the probabilities that your model returns. **Your model should be a correct implementation of a language model.** That is, it should be 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 previous assignments, 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. This is very important for the second part of this homework, where you always need to consider the batch dimension in the Transformer model.

Part 2: Implementing a Transformer Language Model

The Transformer is essentially an encoder-decoder model, but for traditional language modeling, we only need its decoder part, which will autoregressively predict the next token given the context. The Transformer decoder is composed of several blocks (just call them Transformer blocks), where each block contains self-attention, layer normalization, residual connections, and MLP. Most layers are straightforward to be implemented, except that the self-attention requires causal masking to prevent the model from seeing the future tokens (i.e., masking the upper diagonal part of the attention matrix). Check our lecture slides and online tutorials[3] for more details of Transformer.

Q2 (40 points) Implement a Transformer language model. It is recommended to build up the Transformer decoder model from bottom to up, first implement the self-attention module, then implement the Transformer block and finally build up the Transformer model composed of multiple blocks. We provide hints inside the code. There are two main purposes of implementing the Transformer from scratch, one is the be familiar with tensor operations and transformations in PyTorch with many matrix multiplications inside the Transformer, and the other is to truly understand the Transformer model, which has become the backbone model of modern NLP nowadays.

Note that there are several hyperparameters to tune the Transformer, such as the embedding size, hidden dimension, number of layers, max input length, etc. Tuning them as well as the learning parameters, like epochs, and learning rates, to get the best performance you can have for Transformer LMs. You may experience worse performance than RNN LMs sometimes, which is normal because training Transformer LMs requires more data and computation. Given your limited running time (e.g., around 5-10 minutes of training) and computation resource (no need to use GPUs), there is no hard performance threshold for this one as long as you can justify you've implemented Transformer LMs correctly.

Q3 (20 points) Report the performance of your RNN and Transformer language models in a 1-2 page report. Explain the general process of training a language model and explain the differences between RNN LM and Transformer LM. Show your efforts on tuning hyperparameters to get better results, especially for the Transformer LM.

Bonus (30 points) Once you've trained your language model, there are two main things it can do, 1) assigning probabilities to any legitimate sequences, 2) generating or sampling sequences from it. For the bonus points, implement a generation function `generate()`[4] inside either `RNNLanguageModel` or

[3] `https://peterbloem.nl/blog/transformers`; `http://nlp.seas.harvard.edu/2018/04/03/attention.html`

[4] Check a similar API from Huggingface: `https://huggingface.co/docs/transformers/v4.24.0/en/main_classes/text_generation`

`TransformerLanguageModel` to generate a continuation given a prompt/context (could be an empty token). Basically, you need to autoregressively call `get_next_char_log_probs()` and select a token from the distribution at each step to be fed into the next step. You can implement whichever decoding method you've learned in the course, greedy decoding or beam search. Show 2-3 interesting generated samples in your report.

Deliverables and Submission

You will upload your code in `models.py`, `train.py` and the report on Carmen.

Make sure that the following commands work before you submit and you pass the sanity and normalization checks for `lm.py`:

```
python lm.py --model RNN

python lm.py --model Transformer
```

These commands should run without error and train within the allotted time limits.

0.1 Acknowledgment

This homework assignment is based on NLP courses by Dr. Greg Durrett at UT Austin.

References

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

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.