Homework 4 Part 1

Language Modeling using RNNs

11-785: Introduction to Deep Learning (Fall 2022)

OUT: **November 18th, 2022**
DUE: **December 9th, 2022, 11:59 PM ET**
Early Bonus: **November 26th, 2022, 11:59 PM**

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to help your friends debug
  - You are allowed to look at your friends code
  - You are not allowed to type code for your friend
  - You are not allowed to look at your friends code while typing your solution
  - You are not allowed to copy and paste solutions off the internet
  - You are not allowed to import pre-built or pre-trained models
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.

  We encourage you to meet regularly with your study group to discuss and work on the homework. You will not only learn more, you will also be more efficient that way. However, as noted above, the actual code used to obtain the final submission must be entirely your own.
- **Overview/TL;DR:**
  - **Part 1**: All of the problems in Part 1 will be graded on Autolab. Download the starter code from Autolab.
  - **Single Word prediction**: Complete the function **predict** in class **TestLanguageModel** in the Jupyter Notebook.
  - **Generation of Sequence**: Complete the function **generate** in the class **TestLanguageModel** in the Jupyter Notebook.
  - **Beam Search [Bonus]**: Use beam search on prediction and generation tasks.

Homework Objectives

- **If you complete this homework successfully, you would ideally have learned**
  - Learn how to train Recurrent Neural Network models for generating text.
  - Learn about various techniques to regularize Recurrent Neural Networks like **Locked Dropout**, **Embedding Dropout**, **Weight Decay**, **Weight Tying**, **Activity Regularization**
  - You will understand the workings of Language Modeling and you will also train one to generate next-word as well as entire sequences!

1 Introduction

Key new concepts: Language modeling, text generation, regularization techniques for RNNs

Mandatory implementation: RNN-based Language model, text prediction and text generation using your trained model, Greedy Decoding.

Restrictions: You may not use any data besides that provided as part of this homework; You may not use the validation data for training; You have to use at least greedy decoding (and optionally beam search) for both text prediction and text generation.

1.1 Overview

In Homework 4 Part 1, we will be training a Recurrent Neural Network on the WikiText-2 Language Modeling Dataset.

You will learn how to use a Recurrent Network (ideally an LSTM-based network) to model and generate text. You will also learn about the various techniques we use to regularize recurrent networks and improve their performance.

The below sections will describe the dataset and what your model is expected to do. You will be responsible for organizing your training as you see fit.

Please refer to Regularizing and Optimizing LSTM Language Models for information on how to properly construct, train, and regularize an LSTM language model. You are not expected to implement every method in that paper. Our tests are not overly strict, so you can work your way to a performance that is sufficient to pass Autolab using only a subset of the methods specified in the aforementioned paper.

These "tests" require that you train the model on your own and submit a tarball (handin.tar) containing the code capable of running the model, generating the predictions and plotting the loss curves. Details follow below.

2 Language Modelling

The problem of language modelling can be simply defined as follows: `Given a sequence of` *`tokens`* `(words or character), predict the next one`.

A variant of the problem can be stated as `Given a sequence of` *`tokens`* `(words or character), predict the missing or` *`masked`* `one`. It is referred to as **Masked language modelling**.

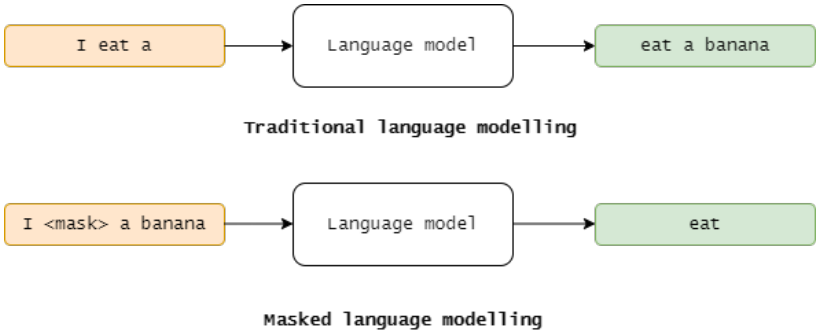

Figure 1: Some variants of language modelling

In traditional language modelling (which is the objective of this homework), a trained language model will learn the likelihood of the occurrence of a word based on the previous words. Therefore, the input of your model is the previous text (see Figure 3 and lecture 15 for more details about language modelling).

Of course, language models can be operated on at different levels, such as character level, n-gram level, sentence level and so on. However, in this homework, we recommend using the word level representation. You may try to use character level if you wish.

Additionally, it would be better to use a "fixed length" input. (This "fixed length" input is not the most efficient way to use the data, you could try the method in section 4.1. of Regularizing and Optimizing LSTM Language Models).

Lastly, using packed sequences is not a requirement.

3 Dataset

A pre-processed WikiText-2 Language Modeling Dataset is included in the template tarball and includes:

- `vocab.npy`: a NumPy file containing the words in the vocabulary
- `vocab.csv`: a human-readable CSV file listing the vocabulary
- `wiki.train.npy`: a NumPy file containing training text
- `wiki.valid.npy`: a NumPy file containing validation text

The train and validation files each contain an array of articles. Each article is an array of integers, corresponding to words in the vocabulary. There are 579 articles in the training set.

As an example, the first article in the training set contains 3803 integers. The first 6 integers of the first article are `[1420 13859 3714 7036 1420 1417]`. Looking up these integers in the vocabulary reveals the first line to be: `= Valkyria Chronicles III = <eol>`

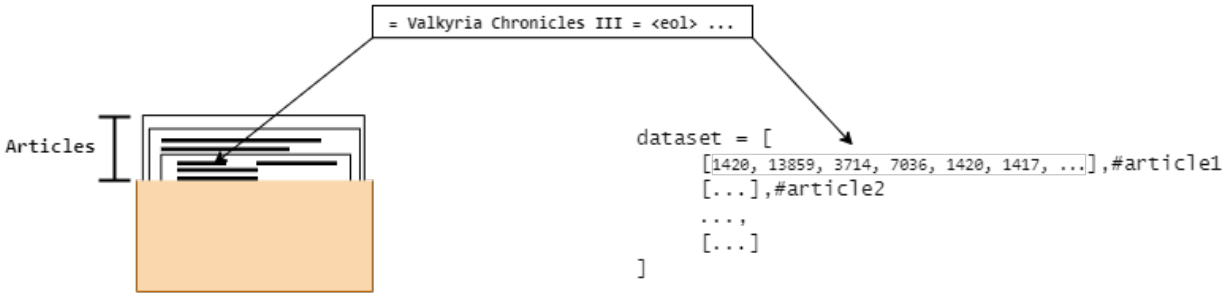

Figure 2: A little visualization of the dataset

`<eol>` (end of line) is a special token added to the vocabulary to specify the end of a given line.

A preview of the raw dataset (Wikitext 2) is also available on **huggingface.co** via the following link https://huggingface.co/datasets/wikitext/viewer/wikitext-2-raw-v1/train.

The dataset for this homework is provided as a collection of articles. So you may concatenate those articles to perform batching as described in the paper. It is advised to shuffle articles between epochs if you take this approach.

You will also receive a vocabulary file containing an array of strings. Each string is a word in the vocabulary. There are 33,278 vocabulary items.

Refer to section 6 for more details about the testing procedure.

3.1 DataLoader

For your model to be able to generate meaningful outputs, it has to learn how to build sentences in the selected language (in this case, English). This is achieved through language modelling (see section 2).

In this section, we show how to make the most out of the dataset. As explained in section 1, your main task is to build a language model that you will use to perform text generation. Thus, your samples should be formatted in a certain way so as to successfully train your language model.

3.1.1 Example

Lets see an example.

Article: *I eat a banana and an apple everyday including today.*

The following may be considered as hyperparameters that you can tune at your convenience.

- sequence_length = 3
- batch_size = 2

Then one batch consists of 2 of pairs of sequences which are pairs of (input sequence, target sequence).

A batch would have 2 training samples with sequence length 3. The first batch looks like this:

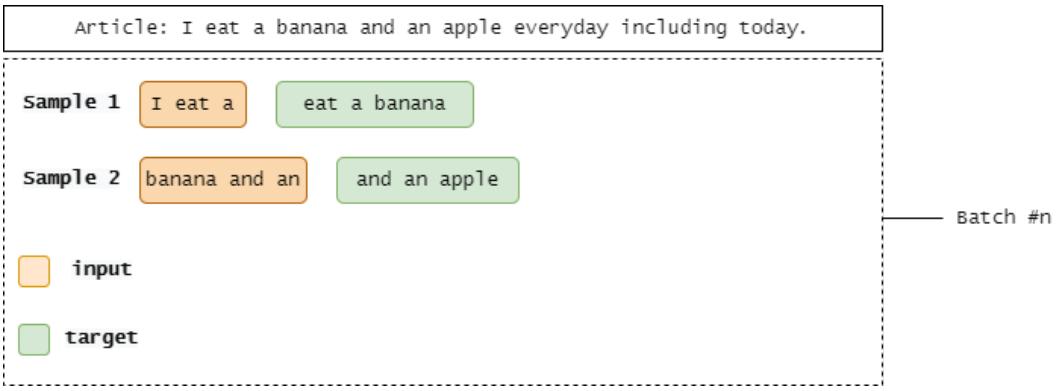

Figure 3: An example of batching and data loading

Pytorch's `DataLoader` class allows you to format your samples the way you desire. To do so, you will have to overwrite the `__iter__` method (See starter notebook).

The `__iter__` method should:

1. Randomly shuffle all the articles from the WikiText-2 dataset.
2. Concatenate all text.
3. Group the sequences into batches.
4. Run a loop that returns a tuple of `(input, label)` on every iteration with `yield`. (look at iterators in python if this sounds unfamiliar)

4 Problems

4.1 Prediction of a Single Word (50 points)

Complete the function `prediction` in class `TestLanguageModel` in the Jupyter Notebook.

This function takes as input a batch of sequences, shaped `[batch size, sequence length]`. This function should use your trained model and perform a forward pass.

Return the scores for the next word after the provided sequence for each sequence, which means you should return the score for the last timestep from the output of the model. The returned array should be in shape of `[batch size, vocabulary size]` (float).

These input sequences will be drawn from the unseen test data. Your model will be evaluated based on the score it assigns to the actual next word in the test data. Note that scores should be raw linear output values. Do not apply softmax activation to the scores you return. **Required performance for this task is a negative log likelihood of 5.0**. The value reported by autolab for the test dataset should be similar to the validation NLL you calculate on the validation dataset. If these values differ greatly, you likely have a bug in your code.

4.2 Generation of a Sequence (50 points)

Complete the function `generate` in the class `TestLanguageModel` in the Jupyter Notebook.

As before, this function takes as input a batch of sequences, shaped `[batch_size, sequence_length]`.

Instead of only scoring the next word, this function should generate an entire sequence of words. The length of the sequence you should generate is provided in the **n_words** parameter. The returned shape should be `[batch_size, n_words]`. This function requires sampling the output at one time-step and incorporate that in the input at the next time-step. You can call the function `predict` in class `TestLanguageModel` to get the score of prediction. Please refer to Recitation 8 for additional details on how to perform this operation. Your predicted sequences will be passed through tests that we have curated, and the NLL of your outputs will be calculated.

If your outputs make sense, they will have a reasonable NLL. If your outputs do not reasonably follow the given outputs, the NLL will be poor. **Required performance for this task is a negative log likelihood of 3.0**.

5 Training and regularization Techniques

You are free to structure the training and engineering of your model as you see fit. Follow the protocols in Regularizing and Optimizing LSTM Language Models as closely as you are able to, in order to guarantee maximal performance.

Make sure your implementation is consistent with the batching strategy you choose (batch first or batch last). You may be required to transpose the outputs of your prediction and/or generation models.

The following regularization techniques will be sufficient to achieve performance to pass on Autolab. Refer to the paper for additional details and please ask for clarification on Piazza. It may not be necessary to utilize all of the below techniques.

- Apply weight decay
- Apply locked dropout between LSTM layers
- Apply embedding dropout
- Tie the weights of the embedding and the output layer
- Activity regularization

- Temporal activity regularization

5.1 Locked Dropout

In standard dropout, a new binary dropout mask is sampled every time when the dropout function is called. For example, the input x0 to an LSTM at timestep t = 0 receiving a different dropout mask than the input x1 fed to the same LSTM at t = 1.

A variant of this, locked dropout, also called variational dropout (Gal Ghahramani, 2016), samples a binary dropout mask only once upon the first call. It then repeatedly uses that locked dropout mask for all inputs and outputs of the LSTM within the forward and backward pass.

A thing to notice is that each sample within the mini-batch should use a unique dropout mask. It will ensure the diversity in the elements dropped out. Please also think about on which dimension you should apply the mask.

5.2 Embedding dropout

Embedding dropout (Gal Ghahramani (2016)) is to perform dropout on the embedding matrix at a word level. The dropout is broadcasted across all the word vector's embedding. The remaining non-dropped-out word embeddings are scaled by $\frac{1}{1-p_e}$ where p_e is the probability of embedding dropout. Embedding dropout is performed for a full forward and backward pass, which means all occurrences of a specific word will disappear within that pass.

5.3 Weight tying

Weight tying (Inan et al., 2016; Press Wolf, 2016) shares the weights between the embedding and softmax layer, substantially reducing the total parameter count in the model. The technique has theoretical motivation (Inan et al., 2016) and prevents the model from having to learn a one-to-one correspondence between the input and output, resulting in substantial improvements to the standard LSTM language model.

5.4 Activation Regularization (AR)

Activation Regularization is regularization performed on activation functions. Different from L2-Regularization, which is performed on weights, AR penalizes activation functions that are significantly larger than 0. It is defined as :

$$\alpha L_2(m \odot h_t)$$

where m is the dropout mask, L_2 is L2 norm, h_t is the output of the RNN at timestep t, and α is a scaling coefficient.

5.5 Temporal Activation Regularization (TAR)

Temporal Activation Regularization is a kind of slowness regularization for RNNs that penalizes the model from producing large changes in the hidden state. It is defined as:

$$\beta L_2(h_t - h_{t+1})$$

where m is the dropout mask, L_2 is L2 norm, h_t is the output of the RNN at timestep t, and β is a scaling coefficient.

The AR and TAR term should only be applied to the output of the final RNN layer as opposed to being applied to all layers.

6 Testing

In the handout you will find a template Jupyter notebook that also contains tests that you run locally on the dev set to see how your network is performing as you train. In other words, the template contains a test that will run your model and print the generated text. If that generated text seems like English, then the test on Autolab will likely pass.

6.1 Autograder Submission

`hw4/training.ipynb` is a Jupyter Notebook of the template provided to you.

Within the Jupyter Notebook, there are `TODO` sections that you need to complete.

Every time you run training, the notebook creates a new experiment folder under `experiments/` with a `run_id` (which is CPU clock time for uniqueness). All of your model weights and predictions will be saved there.

The notebook trains the model, prints the Negative Log Likelihood (NLL) on the dev set and creates the generation and prediction files on the test dataset, per epoch.

Your solutions will be autograded by Autolab. In order to submit your solution, create a tar file containing your code. The root of the tar file should have a directory named hw4 containing your module code. You can use the following command from the handout directory to generate the required submission tar.

```
make runid=<your run id> epoch=<epoch number>
```

You can find the run ID in your Jupyter notebook (its just the CPU time when you ran your experiment). You can choose the best epoch using `epoch_number`. You will then have to submit the file **handin.tar** to autolab to get your score for the assessment.

Important: Your model will likely take around 3-6 epochs, to achieve **a validation NLL below 5.0** (On the prediction task). The Autolab tests require a performance of 5.4 (or less).
Performance reported in the paper is 4.18, so you have room for error. However, with a good set of hyper-parameters and some well chosen regularization techniques (mentioned above), you can get an NLL below 5.0 in no more than 2 epochs.

Warning: The classes provided for training your model are given to help you organize your training code. You shouldn't need to change the rest of the notebook, as these classes should run the training, save models/predictions and also generate plots. If you do choose to diverge from our given code (maybe implement early stopping for example), be careful.

Good luck !