CS378 Assignment 4: Character Language Modeling with RNNs

Due date: Tuesday, April 9 at 5:00pm CST

Academic Honesty: Reminder that assignments should be completed independently by each student. See the syllabus for more detailed discussion of academic honesty. Limit any discussion of assignments with other students to clarification of the requirements or definitions of the problems, or to understanding the existing code or general course material. Never discuss issues directly relevant to problem solutions. Finally, you may not publish solutions to these assignments or consult solutions that exist in the wild. **While you are free to look up existing implementations of PyTorch models to understand the API and its usage, your implementation should fundamentally be your own, not based off of a preexisting PyTorch RNNLM codebase.**

Goals The primary goal with this assignment is to give you hands-on experience implementing a neural network language model using recurrent neural networks. Understanding how these neural models work will help you understand not just language modeling, but also systems for many other applications such as machine translation.

This assignment is also more "from scratch" than past ones. This will help you prepare for the more open-ended final project.

Dataset and Code

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

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, so we will not discuss it.

`lm_classifier.py` contains the driver for Part 1. It calls `train_rnn_classifier`, which learns an RNN classifier model on the classification data. `lm.py` contains the driver for Part 2, and calls `train_lm` on the raw text data. `models.py` contains skeletons in which you will implement these models and their training procedures.

Part 1: RNNs for Classification (40 points)

In this first part, you will do a simplified version of the language modeling task: binary classification of fixed-length sequences to predict whether the given sequence is followed by a consonant or a vowel. You will implement the entire training and evaluation loop for this model.

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

Data `train-vowel-examples.txt` and `train-consonant-examples.txt` each contain 5000 strings of length 20, and `dev-vowel-examples.txt` and `dev-consonant-examples.txt` each contain 500. The task is to predict whether the first letter following each string is a vowel or a consonant. The consonant file (for both train and test) contains examples where the next letter (in the original text, not shown) was a consonant, and analogously for the vowel file.

Getting started Run:

```
python lm_classifier.py
```

This loads the data for this part, learns a `FrequencyBasedClassifier` on the data, and evaluates it. This classifier gets 71.4% accuracy, where random guessing gets you 50%. `lm_classifier`.py contains the driver code, and the top of `models.py` contains the skeletal implementation for this classifier.

Q1 (30 points) Implement an RNN classifier to classify segments as being followed by consonants or vowels. This will require defining a PyTorch module to do this classification, implementing training of that module in `train_rnn_classifier`, and finally completing the definition of `RNNClassifier` appropriately to use this module for classification.

Your final model should get **at least 75% accuracy and train in less than 10 minutes**.[2] **In your report, you should: (1) Describe your model and implementation. (2) Report accuracy results and timing information.** Even if you are not able to get this part fully working, **write up and document as much as you can so we can give appropriate partial credit.**

Network structure The inputs to your network will be sequences of character indices. You should first embed these using a `nn.Embedding` layer and then feed the resulting tensor into an RNN. Two effective types of RNNs to use are `nn.GRU` and `nn.LSTM`. Finally, you should take the output of the RNN (the last hidden state) and use it for classification (optionally put it through feedforward layers and then a softmax layer). This classification part is very similar to what you did in Assignment 2. You can make your own `nn.Module` that wraps the embedding layer, RNN, and classification layer.

Code structure Once you have your own module implemented, the training and eval loop that wraps them will look roughly the same as in Assignment 2. First, you need a function to go from the raw string to a PyTorch tensor of indices. Then loop through those examples, zero your gradients, pick up an example, compute the loss, run backpropagation, and update parameters with your optimizer. You should implement this training in `train_rnn_classifier`.

Using RNNs LSTMs and GRUs can be a bit trickier to use than feedforward architectures. First, these expect input tensors of dimension [sequence length, batch size, input size]. You can use the `batch_first` argument to switch whether the sequence length dimension or batch dimension occurs first. If you're not using batching, you'll want to pad your sentence with a trivial 1 dimension for the batch. `unsqueeze` allows you to add trivial dimensions of size 1, and `squeeze` lets you remove these.

Second, an LSTM takes as input a pair of tensors representing the state, h and c. Each is of size [num layers * num directions, batch size, hidden size]. To start with, you probably want a 1-layer RNN just running in the forward direction, so once again you should use `unsqueeze` to pad things out. GRUs are similar but only have one hidden state.

[2]Our reference implementation can get 78.2% in 6 minutes of training with an unoptimized, unbatched implementation.

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. For these and other commands, read the error messages you get out and check the documentation.

General tips As always, make sure you can overfit a very small training set as an initial test. If not, you probably have a bug. Then scale up to train on more data and check the development performance of your model.

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 RNN may work better than you think!

Q2 (10 points) What happens if you limit the amount of context the model uses? Try a few different values for the number of context characters used (up to a max of 20, which is the default in the dataset). What trends do you observe?

Part 2: Implementing a Language Model (60 points)

In this second part, you will implement an RNN language model. This should build heavily off of what you did for Part 1, 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 and test set (not given to you) are each 500 characters taken from elsewhere in the collection.

Getting started

```
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. As a probability, perplexity can be interpreted as a measure of the "branching factor."

Q3 (60 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 classification. Your network should take indexed characters as input, embed them, put them through an RNN, and make predictions from the final layer outputs.

Your final model should get **a perplexity value less than or equal to 7 and train in less than 20 minutes.**[3] **In your report, you should: (1) Describe your model and implementation. (2) Report accuracy results and any relevant time information.** As with Part 1, even if you are not able to get this part fully working, **write up and document as much as you can so we can give appropriate partial credit.**

[3]Our reference implementation gets a perplexity of 5.57 in about 6 minutes of training. However, this is an unoptimized, unbatched implementation and you can likely do better.

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 (not quite right), "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 state. These may only make minor differences, though.

Start of sequence In general, the beginning of any sequence is represented to the language model by a special start-of-sequence token. This means that the inputs and outputs of a language model are slightly different, since an LM will never output the start-of-sequence character but may need to read it as input. **For simplicity, we are going to overload space and use that as the start-of-sequence character.**

Evaluation Unlike past homework where you are evaluated on correctness of predictions, in this case your model is evaluated on perplexity and likelihood, which rely on particular values that you report. **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 this as closely as you can.

Q4 (10 points BONUS) Implement batching in your RNNLM. See how fast you can get it to train! Describe your optimizations in your writeup. Bonus points will be awarded at the instructors' discretion based on speed and performance of the final model.

Deliverables and Submission

Your submission will be evaluated on several axes:

1. Writeup: correctness of answers, clarity of explanations, etc.
2. Execution: your code should train and evaluate within the time limits without crashing and give the required accuracy values

Submission You should submit the following files to Canvas **as a flat file upload (no zip or tgz)**:

1. A PDF or text file of your answers to the questions
2. `models.py`. **Do not modify or upload any other source files.**

Make sure that the following commands work before you submit:

```
python lm_classifier.py --model RNN
```

```
python lm.py --model RNN
```

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

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*.