Assignment3: Recurrent Neural Networks

Koç University COMP441-541

Due on December 19th, 2022 (23:59:59)

Introduction

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 and building one from scratch will help you understand not just language modeling, but also systems for many other applications such as machine translation. 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 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. *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.

Part1: 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.

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 *Frequency-BasedClassifier* 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 (40 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.

Important Note: Your final model should get at least 75% accuracy. 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.

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

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*. Their weights should be randomly initialized when you construct them, though you are also free to experiment with different initializers such as the Glorot initializer (*nn.init.xavier uniform*). You should take the output of the RNN (the last hidden state) and use it for binary classification with a softmax layer. You can add one or more feedforward layers before the softmax layer if you want. 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 *ffnn_example.py* and 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 make a 3-tensor with first dimension length of 1. GRUs are similar but only have one hidden state.

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.

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.cConsider 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!

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 is 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.The *RNNLanguageModel* 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 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.

Q2 (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 prediction. 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 must pass the sanity and normalization checks, get a perplexity value less than or equal to 7. 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.

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 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 In that part, your model is evaluated on 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 $P(w_i|w_1, ..., w_{i-1})$. 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.

Deliverables and Submission

You will submit both your code and writeup to Blackboard.

Written Submission You should upload to Blackboard a PDF of your answers to the questions, you can use the latex template that you've used in previous assignment. Your submission for this assignment is evaluated primarily on the basis of code and execution. Your writeup should simply document what you did and report the results you saw. If you are unable to get the code fully working, please write up what you did as thoroughly as possible so we can assign you appropriate partial credit.

The only file you will be submitting is *models.py*, which should be submitted as an individual file upload. If you wish to implement other classes or functions, put them in *models.py* even if this might not be the ideal coding practice.

Late Policy

You may use up to 7 grace days over the course of the semester for the practicals you will take. You will only use up to 3 grace days per assignment.

Academic Integrity

All work on assignments must be done individually unless stated otherwise. Turning in someone else's work, in whole or in part, as your own will be considered as a violation of academic integrity. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.[2]

References

[1] Mikolov, Tomas, Ilya Sutskever, Anoop Deoras, Hai Son Le, Stefan Kombrink and Jan Honza Černocký. 2012. Subword Language Modeling with Neural Networks.

[2] Adapted from CS378: Natural Language Processing Course