CSE 190: Programming Assignment 4: BeerMind

Fall 2018

Instructions

Due Friday, November 30th at 11:59PM:

1. It is highly recommended that you start this assignment ASAP, so that you have sufficient time to clarify any doubts regarding implementing the expected RNN architecture in PyTorch, debug your code, and experiment with good number of models, as well. The dataset we are working with is very large and expect it to take a few hours to train a model.

2. Please submit in your assignment via Gradescope. The report should be in NIPS format or another "top" conference format (e.g. IEEE CVPR, ICML, ICLR) - we expect you to write at a high academic-level (to the best of your ability).

3. Also, submit your code on Gradescope. Keep your code clean with explanatory comments (additionally as you may want to reference or reuse it in the future).

4. The Part I of this assignment is to be done individually so as in order to help you better understand the topics and help you prepare for the final.

5. In Part II, we want you to get the most out of this assignment, so we are requiring you to work on this assignment in ***teams of (maximum) size 2 only***. We also encourage you to shuffle teams. You can try doing this individually, but it is not recommended to do so.

6. We encourage you to start this assignment early, as the dataset is quite large and may take several minutes for each epoch of training. When you consider running different experiments, the simulations will take time. ***So, start early and start often!***

7. We will not tolerate plagiarism of any kind. Your submitted code will be validated for plagiarism. You can use any resources such as papers, blogs, posts etc. **as long as you cite them**!

Part I

Understanding RNN and CNN Basics

This portion of the assignment is to build your intuition and understanding the basics of recurrent networks - namely how RNNs are unrolled through time and the features that are learned - by manually simulating these. We also revisit CNNs to introduce you to a variant of convolution.

1. Consider the single hidden layer RNN in figure 1.

   Unroll this network 2 steps in time, starting at time step t and properly label the inputs and outputs going in and coming out of this network at each step, tagged by time step (e.g., $A(t), A(t+1)$, etc.). Clearly label the weights (i.e., use the following labels: w_{X_1A}, w_{AA}, w_{AB}, etc.)!

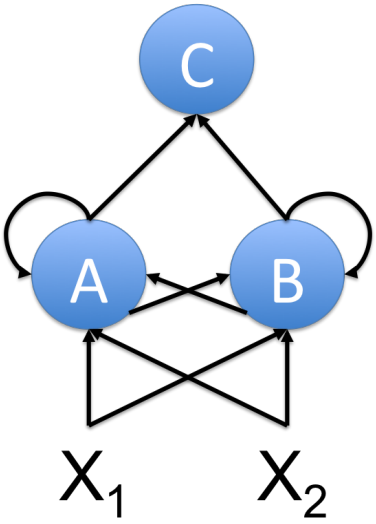

Figure 1: Recurrent Neural Network

2. Given a sequence of length 10, i.e. it has 10 time steps, where input at each time step is of 300 dimensions (a 1×300 vector). Suppose you build an RNN with 2 hidden layers h_1, h_2 (assuming there's no feedback from the second hidden layer to the first hidden layer), such that each hidden layer is of 100 dimensions, and is fully-connected to itself, i.e. hidden state at each layer would be a 1×100 vector. On top of it, you have an output layer which has 24 output neurons.

   (a) For any one particular time step, what will be the dimensions for:

      i. $W_{h_1h_1}$: Hidden state weight matrix for hidden layer 1
      ii. W_{ih_1} : Input weight matrix for hidden layer 1
      iii. b_{h_1} : Bias matrix for hidden layer 1
      iv. $W_{h_2h_2}$: Hidden state weight matrix for hidden layer 2
      v. $W_{h_1h_2}$: Input weight matrix for hidden layer 2, which takes input from hidden layer 1
      vi. b_{h_2} : Bias matrix for hidden layer 2
      vii. W_{h_2o} : Weight matrix for hidden layer 2 - output layer
      viii. b_o : Bias matrix for output layer

   (b) Now considering the entire sequence of length 10, what is the total number of parameters in this model, for all the time steps?

3. **Depthwise separable convolution:** In the regular 2D convolution performed over multiple input channels, the filter is as deep as the input and lets us freely mix channels to generate each element in the output. Depthwise convolutions don't do that - each channel is kept separate - hence the name depthwise. Here's a diagram to help explain how that works (Figure 2):

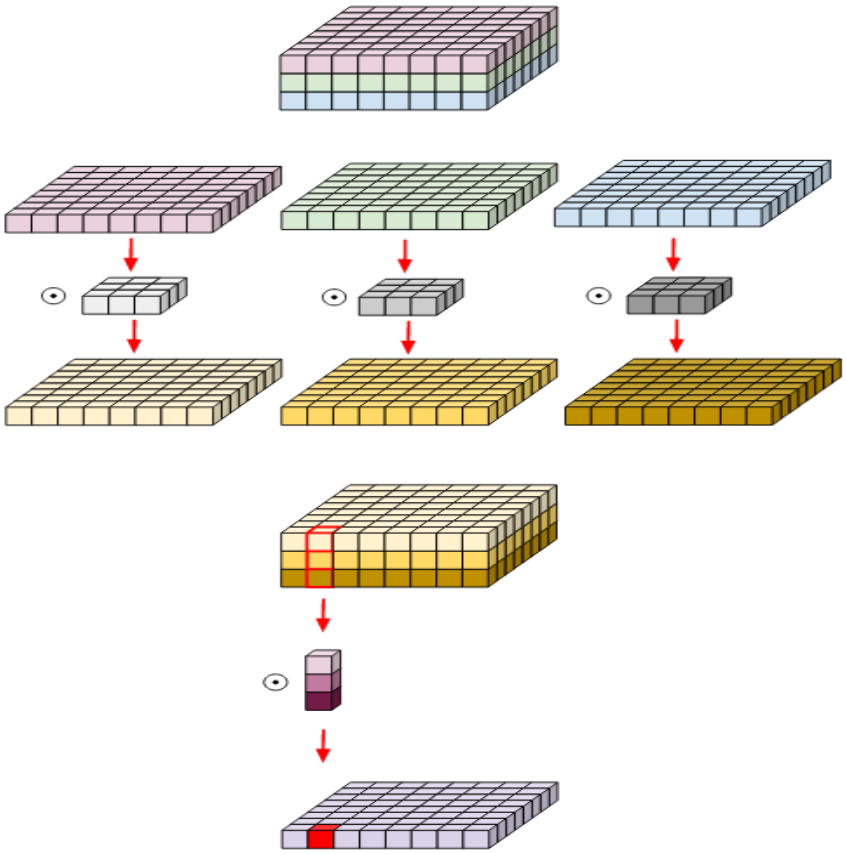

Figure 2: Depthwise Separable Convolution

After completing the depthwise convolution, an additional step is performed: a 1×1 convolution across channels. This step can be repeated multiple times for different output channels. The output channels all take the output of the depthwise step and mix it up with different 1×1 convolutions.

Suppose you are given a 3-channel input with spatial dimensions 256×256. Imagine a convolutional layer with 8 different convolutions, and so 8 channel outputs. Below, we will compare the number of learnable parameters of using standard ("vanilla") convolution and depth-separable convolution. Assume that all filters have a stride of 1 and no padding.

(i) How many learnable parameters are needed for a standard 3×3-size convolution kernel (again, with eight different kernels)?

(ii) How many are needed for the depth-separable convolution, with 3×3-size convolutions, followed by 8 1×1 convolutions?

(iii) For each case, what is the output feature map size?

(iv) For each case, how many floating point operations are needed?

Compare results from the two approaches and reason why it might be a good idea to use depth-separable convolutions instead of vanilla convolutions.

Part II

Beer Review Generation

You will build a recurrent neural network-based language model to generate beer reviews tailored to describe specific categories and star ratings, just as was demonstrated in class. See the demo here if you missed class. We will use the Beer Advocate dataset here, which has a large corpus of reviews that users have assigned to various beers along with their ratings and corresponding beer properties. You can treat each review as a sample. For more information about language modelling using RNNs and LSTMs, refer to Andrej Karpathy's blog.

Training a Recurrent model for Review Generation

To generate reviews for beers, we will follow this paper's approach. The input to the RNN will be a fixed input (one-hot encoding) of the beer type and rating that is repeated on every time step. Let's call this feature vector x_{feat}. The dimension of this feature vector is denoted by d_{feat}.

You will combine this feature vector with one-hot encoded character inputs, as well as punctuation (comma, colon, etc.). For this build a simple function that maps characters to one-hot encodings, call it `char2oh` and the other way round, call it `oh2char`. Lets call the one-hot encoded character input as x_{oh}. Each time you predict a character, it will be fed into the next time step for predicting the next character. During training, the actual next character is fed in; during testing, the softmax output is *sampled* to obtain the next character.

Now we can concatenate x_{oh} and x_{feat} and feed it into a recurrent neural network. Then you use a softmax on top of the network to predict the *next* character. The one-hot encoding of this next character, say $x_{\text{oh,next}}$ concatenated with x_{feat} would serve as the input to the *next* time step. (output of current time step is the input of the next time step). This entire process is illustrated in Figure 3.

Notably, reviews exhibit consistent large-scale structure, discussing five attributes of each beer (appearance, smell, taste, mouthfeel, and drinkability) in sequence. BeerAdvocate users exhibit idiosyncratic writing styles, adhering to consistent protocols (hyphenation, colons, line-breaks) for delineating the various aspects of reviews. Naturally, the bulk of the variance between users reviews of a particular beer owes to their own subjective opinions and tastes, something which a successful model must take into account.

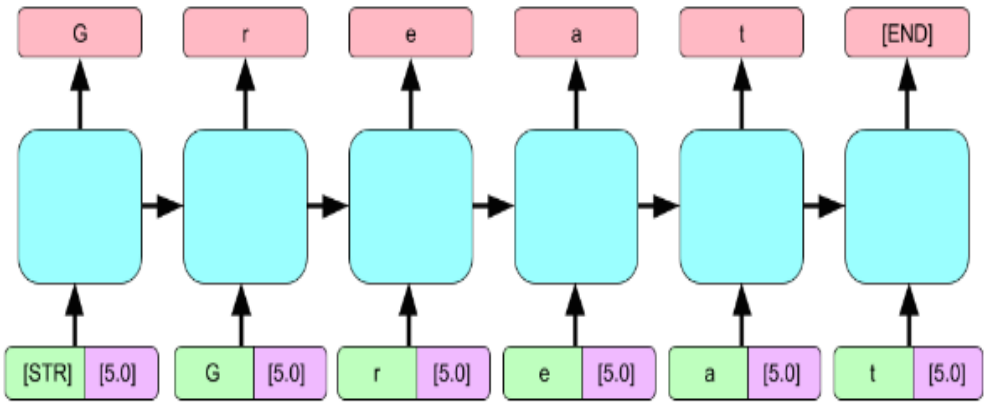

Figure 3: Architecture for review generation using concatenated input RNN

1. **Read and process the data**
   You are given two files for the dataset - **Beeradvocate_Train.csv** and **Beeradvocate_Test.csv**. The *Beeradvocate_ Train.csv* contains metadata for the review, as well as the review itself. Of this metadata, you are required to use just **beer/style** and **review/overall** features for training the model, and later for review generation. These two features will henceforth be referred to as "metadata" or "metadata feature vector" throughout the assignment. Note that **review/overall** is given in the form of a fraction, and you should scale that to be a value between 0 to 5, i.e. if the review/overall for a particular record says "13/20", you should convert that to 3.25. Each input to the RNN will be a concatenation of this metadata feature vector and the one hot encoded vector for character. The *Beeradvocate_ Test.csv* contains only the metadata for reviews. Your task, (should you choose to accept it ;)) is to train a model that generates the beer review character-by-character for the given metadata of test file.

   A helpful thing to do would be to prepend each review in the training set with a $< SOS >$ (start-of-sentence) character in the beginning and an append $< EOS >$ (end-of-sentence) character at the end. This way, while generating reviews, when your model sees $< SOS >$, it will start generating reviews until it gets the character $< EOS >$, which your model should learn to generate at the end of each review since this is what it encounters in training samples.

   You need to read the CSV files into a **Pandas** DataFrame first. See this Pandas tutorial for help on how to read files into a DataFrame and to manage/manipulate them. Make use of the **load_data** function to read the files and return DataFrames.

   The **process_train_data** function takes in the training DataFrame and returns the data in a supervised fashion. The input consists of metadata feature vector concatenated with one hot encoded representation of the current character. Output is the one hot encoded representation of the next character. You have to do this for each character of each review. So lets say you have N reviews in total, where each review is m characters long. Let the final input after concatenation of metadata feature vector and one hot encoded representation be a d dimensional vector. In this case, your input for processed training data will be of size $N \times m \times d$, and labels will be of size $N \times m \times v$, where v is the length of one hot encoded vector.
   NOTE: You should append the $< SOS >$ and $< EOS >$ characters before processing them to form feature vectors.

   The **process_test_data** function takes in the test DataFrame and returns the dataset in input vector form. This input vector will be a concatenation of metadata feature vector and the $< SOS >$ character representation in one hot encoded form. So if you have N points in test data, this function will return a $N \times 1 \times d$ matrix for test data.

   The **train_valid_split** function splits the given data into training and validation datasets. You can do this in any manner you want - 80-20 split, 70-30 split, k-fold cross validation split, etc.

2. **Training procedure**
   You will train in a mini-batch fashion. For each mini-batch of size n, there will be reviews of varying lengths. The PyTorch implementation of recurrent networks requires all inputs of a mini-batch to be of same length. To meet this condition, you will find the maximum length review in the mini-batch. Pad the remaining reviews with $< EOS >$ character until all lengths are the same as max length review. The function **pad_data** should be used for this, and it will be called in the **train** function.

   A few things to keep in mind:

   - During training, before passing each mini-batch, initialize the initial hidden states to 0 tensors, denoting that you are not using any pre-stored information for hidden states and are starting with a clean slate for each mini-batch.
   - There might be out of memory issues while working on a GPU, so after each mini-batch, delete any variables (such as initial hidden states of 0) that you create in the training loop for each mini-batch.

3. **Experiment with model architecture**
   You are free to use RNNs, LSTMs or GRUs for your final architecture which you will use to generate reviews for test data. As a part of this experimentation, you have to build an LSTM based architecture, and a GRU based architecture (such that they both have same number of layers, dropout and hidden state dimensions), and then compare them based on convergence time and performance on validation data.

   You will need to use `nltk.translate.bleu_score` module in python in `nltk` to get quantitative performance of your model on validation dataset. The documentation to use this module can be found here. For each review in validation set, you will first generate your own review using the **test** function, and then use the generated reviews and ground truth reviews to compute the BLEU score on validation data (more details in the code provided). You don't need to do this during training for early stopping, but you will need this to select your best model out of all the models you try to finally generate the reviews that you will submit.

   NOTE: If you want, you can also experiment with bi-directional LSTMs and GRUs. You can refer to these links: 1, 2 to understand what they are. They are very easy to implement in PyTorch (see LSTM documentation in PyTorch). Generally for language tasks, bidirectional networks work better than vanilla networks.

4. **Generate reviews for given input conditions**
   Use the **test** function for this task. Given the metadata feature vector for each input, concatenate it to the one hot representation for $< SOS >$ character. Pass this as an input to the RNN to obtain the output. The output (as during training) will be a softmax (with temperature). The difference from training here is that you *sample* from the softmax distribution to get the next character. If you just use the maximally activated output, it will not work well. Use the one hot encoded representation of this character to concatenate to the metadata feature vector and pass that to the second time step of RNN. This continues, till either your model generates the $< EOS >$ character, or the generated review reaches <fill_max_char_here> characters. A good model should be able to generate a decent length review with the $< EOS >$ character at the end.

   The softmax function with temperature is a variant of the standard softmax function, which is given by 1.

$$softmax(x_i) = \frac{exp(x_i/\tau)}{\sum_j exp(x_j/\tau)} \tag{1}$$

   Generate three reviews (to be included in your report) using three settings of τ: (i) $\tau = 0.01$ (very cold!); (ii) $\tau = 0.4$; (iii) $\tau = 10$ (very hot!). You will be generating a max of **1000 character**-long output in all cases, to make sure it will fit in a figure in your report. Give a qualitative comparison of reviews generated in each of these three cases.

   Besides the report, turn in three .txt files, one corresponding to each temperature. Each entry in the .txt file should be the generated review for each test example, in the same order as test examples. Two reviews should be separated by a newline character. The files that you return should be: **reviews_tau_0.01.txt**, **reviews_tau_04.txt** and **reviews_tau_10.txt**. These files will be used to evaluate BLEU score on the test data against ground truth reviews of test data that we have, and this evaluation will be autograded so make sure to maintain the name and structure of these files exactly as specified, else the autograder will throw an error and you will lose points corresponding to acceptable BLEU score range.

   NOTE: While generating a review, initialize the initial hidden state to 0, i.e. when you pass the $< SOS >$ character as input. But while generating subsequent characters for the same review, do not pass any argument for initial hidden states since you need to use hidden state of previous character generated. BUT, when you move to the next review to generate, you should again pass the initial hidden states as 0. This ensures you start generating each review with a clean slate, while maintaining hidden states while generating characters for the same review.

Your report should include the following:

1. **Abstract**
   This includes a brief summary of the task, and the model that you propose to solve this task. Try to include [brief] quantitative results on validation data in this abstract.

2. **Introduction**
   This includes a description of the task, dataset, and general procedure for training the model and generating reviews. You can refer to parts of this write-up ***in your own words*** for this section.

3. **Models**
   This section discusses each of your models in detail, giving specifications about model architecture, and hyperparameters like L2 regularization penalty, optimizer used, loss function used, dimensions of hidden states, number of layers, etc.

4. **Results**
   This section includes plots for training loss vs epochs and validation loss vs epochs plotted on the same graph for each model that you experiment with. This should also specify the BLEU score on validation data for each model when it converges.

5. **Discussion**
   Here you should discuss the difference in performance between LSTM vs GRU based architecture, and give a qualitative comparison of reviews generated on test data for the three values of temperature provided. These qualitative comparisons should be accompanied by 3 examples per model, where each example contains the metadata feature and review generated by each of the LSTM and GRU based models. Should you choose to experiment with more models and if they turn out to be the best ones you use to generate and submit reviews for test data, you must include 3 examples for those ones too.

6. **References**
   You should list any external references you use for this assignment.

7. **Individual Contributions**
   It is important to mention the individual contribution of each team member for this assignment. Failing to do so will result in severe penalty.

Final notes:

- This assignment is a bit tricky to implement, especially with additional responsibility of initializing and maintaining hidden states during training. Any screw up in this could lead to a poorly trained model (or poor generation of review) even if your underlying model architecture is really good.
- The starter code does not provide sample declarations of RNN layers for you. Now that you have some familiarity with PyTorch, it is up to you to learn to go through the online documentation and implement it correctly, ensuring that you are passing the inputs in expected fashion, and interpreting the outputs correctly as well.
- Feel free to define any additional functions you want, and add any keys in the cfg dictionary if you think it might be necessary.
- It is highly recommended that you start this assignment ASAP, so that you have sufficient time to clarify your doubts regarding PyTorch implementation of RNNs, debug your code and experiment with good number of models as well. The dataset is a huge one and it will take a few hours to train a model.
- The term 'RNNs' has been used in a lot of places in this assignment write-up. This does not mean you have to use vanilla RNNs. RNNs is used in a general sense to denote vanilla RNNs, LSTMs and GRUs.