Deep Learning Assignment 4
Recurrent Neural Network

Yijun Xiao
Center for Data Science
New York University
New York, NY 10003
ryjxiao@nyu.edu

1 Introduction

Starting with Wojciech Zaremba's publicly available code [1], we experimented the performances of different structures of recurrent neural networks (RNN) with long short term memory (LSTM) cells on character-level language modeling tasks. Performances were evaluated using perplexity on the Penn Treebank dataset. Our best model achieved a validation perplexity of 228.04.

2 Data

For this assignment, we used the Penn Treebank dataset. Sentences were broken into characters and spaces were replaced with underscores. There are a total of 42,068 sentences in the training set and 3,370 in validation. In terms of number of characters, there are 4,975,414 and 389,672 characters in the training and validation set respectively thus a training validation split of approximately 92.7% / 7.3%.

3 Model

We used LSTM RNN to train the language models. LSTM RNN is a type of recurrent neural network that leverages the concept of long short-term memory. LSTM cells can memorize state information of previous time steps and use it to help make predictions. The LSTM structure we used in this paper is described in Zaremba et al. 2014. The same paper also introduced dropout for RNN's, we investigated the effect in our experiments.

Each LSTM memory cell has an input gate, an output gate, and a forget gate. Input gate adjusts how much weight we would like to give the input module; forget gate governs how much state information from previous time steps we should retain; output gate controls the magnitude of the cell output value.

An LSTM RNN consists of multiple layers of LSTM cells each depends on the output of the previous layer and output of itself at previous step. Input layer is a lookup table which translate character indices to 1D tensors. Prediction layer is a fully connected layer transforming outputs back to the correct size plus a softmax layer.

Table 1: Experiment Results

Params	Baseline	Model 1	Model 2	Model 3	Model 4
batch_size	20	20	20	20	25
seq_length	50	50	50	50	50
layers	2	2	2	3	3
decay	2	2	2	2	1.25
rnn_size	200	500	500	500	1500
dropout	0	0	0.5	0	0.5
init_weight	0.1	0.1	0.1	0.1	0.075
lr	1	1	1	1	0.05
max_epoch	4	4	4	4	9
max_max_epoch	13	13	13	13	36
max_grad_norm	5	5	5	5	9
perplexity	352.65	260.33	680.88	228.04	232.63

4 Experiments

Several architecture and optimization parameters can be adjusted. We experimented the effect of input size, number of layers, as well as dropout.

Baseline model is a two-layer network with input size 200 and sequence length 50. Initial learning rate is 1 and decays to half every 4 epochs. After 13 epochs, validation perplexity achieved 352.65. Our best model used a similar configuration except for input size set to 500 and number of layers set to 3. We tried a larger model with 3 layers, input size 1500, sequence length 50, batch size 25, and dropout 0.5. But the result was not better than its smaller counterpart. Experiment results are concluded in Table 1.

From the results we can conclude dropout is not beneficial for small networks; larger size of input improves network performance; larger number of layers also improves performance; initial learning rate needs to be smaller when `rnn_size` is large and learning rate decay should also be smaller if we need to run more epochs.

We are not able to complete the comparison of the effect of different dropout rates due to time constraint. We expect dropout to work better for large network at certain rate.

5 Answers to the questions

5.1 Q2

`i` is the state value of the previous layer with/without dropout; `prev_c` is the cell value of the same layer at previous time step; `prev_h` is the state value of the same layer at previous time step.

$$\texttt{i} = h_t^{l-1}, \texttt{prev\_c} = c_{t-1}^{l}, \texttt{prev\_h} = h_{t-1}^{l}$$

5.2 Q3

The function `create_network()` returns a cuda version module that represents one slice of the unrolled network. i.e. all layers at one particular time step.

5.3 Q4

`model.s` stores state information, i.e. c_t^l and h_t^l.

`model.ds` stores gradient with respect to the states of layers at the next time step and is used in backpropagation.

`model.start_s` stores state information resulted from last forward step and uses it as the starting state for the next forward propagation step. `model.start_s` is reset to zero when remaining data length is smaller than preset `seq_length`.

5.4 Q5

If norm of gradient of the parameters exceeds a preset value, multiply the gradient by a factor such that the norm of the rescaled gradient equals the preset maximum value.

5.5 Q6

The code uses stochastic gradient descent. For the 1-hour network, batch size is 20; initial learning rate is 1 and decays by a factor of 2 every 4 epochs; optimization is terminated after 13 epochs.

5.6 Q7

The gradient of the extra output being fed to back-propagation should be set to zero, otherwise the gradient of the parameters will be calculated twice.

References

[1] https://github.com/wojzaremba/lstm

[2] Wojciech Zaremba, Ilya Sutskever, Oriol Vinyals. Recurrent Neural Network Regularization. arXiv:1409.2329v4 [cs.NE], 2014.

[3] Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya and Salakhutdinov, Ruslan. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research 15*, 2014.

[4] Graves, Alex. Generating sequences with recurrent neural networks. arXiv:1308.0850, 2013.

[5] Hochreiter, Sepp and Schmidhuber, Jurgen. Long short-term memory. *Neural computation*, 9(8):1735-1780, 1997.