DS-GA-1008 Assignment 4: Team Longcat

Long Sha

Part 1: Character-level RNN language model

For this part of the assignment, I turned the word-level recurrent neural network (RNN) language model into a character-level RNN language model. The RNN language model embeds long-short term memory (LSTM) units into a recurrent neural network structure (Mikolov 2012). Conventional approaches, such as applying convolutional neural networks on language processing, lack the ability to consider language context. Context is crucial for the current task in predicting words, since the next word in a sentence depends does not depend only on a fixed set, but on various sets of words in a sentence we term memory. Therefore, a neural network with the ability to encode memory of a context is suitable for our task. RNN is a neural network structure that takes past model output as input, and contains input dependent hidden states. These hidden states are analogous to the memory we need for language processing. However, taking a sentence as input, these hidden states from conventional RNNs only cover a short segment of the input, or we call short-term memory. To successfully encode language context, we need both short- and long-term memory to cover contexts that span various timescales in a language text. LSTM is a form of RNN units that has the property for both long-term and short-term memory processing.

In the current task, I turned a well-trained word-level language model into a character-level language model. The basic model structure remained the same. First, the model contains LSTM units that take input from current character, previous hidden state, and previous long-term memory cell. Each LSTM unit outputs the next hidden state and the next long-term memory cell. This LSTM unit is then embedded in an unfolded 2-layer recurrent neural network. Each RNN unit takes the current word, next word, and the previous state of the network (which includes both hidden state and long-term memory cell from LSTM) as input. The network then feeds the inputs into a 2-layer recurrent neural network, and outputs the error, predictions for log probabilities, and current state of model. The same network is then cloned based on the length of the sequence of characters segmented for training.

The training set contains 5017.5k characters, and the validation set contains 393k characters. The class of potential character output contains 50 unique classes that cover English alphabet, Arabic numbers and common symbols.

For the specific model I trained, I used the following parameters for the architecture of the model. As input, the sequence length is 10, with a batch size of 20. I adopted 2-layer RNN, with 500 units per layer. The weights were initialized uniformly between -0.1 to 0.1. I applied 0% drop-out on the on the non-recurrent connections of the network

(Wojciech et al., 2015). I used a learning rate of 1, and after 14 training epochs, I reduced the learning rate by a factor of 1.15 for every epoch. I used norm-clipping method for gradient regularization at a norm of 5.5. For every training step, the gradient of error is set constant to 1. This constantly trains the network with a fixed gradient error.

After training the model for 5 hours with 20 epochs, I achieved an average perplexity of 230 on the training set, and 354 on the validation set. With longer epochs, this approach would further reduce the perplexity.

I also tested other parameter settings for the training. Specifically, I varied the size of RNN and length of sequence. Increasing the size of RNN will prolong the training time, and requires more important roles of regularization using dropout during training. Interestingly, I tested the model with the same set of parameters as reported above, and applied RNN with 1000 units per layer, with Drop-out at 0.5. My results indicate that 500 units without Drop-out could perform better than RNN with 1000 units, with drop-out. I suspect that for character generation, models with large set of parameters is much harder to train. The task for generating character, with lower dimensionality of input and output, might be less complex than generating words, and therefore calls for neither a need for large dimensionality for hidden state nor for regularization. However, beyond the hyperparameters I tested, maybe more appropriate regularization methods in Drop-out and gradient norm clipping are required for a model with large set of parameters to work.

In addition, with no Drop-out, RNN with 500 units out-performs RNN with 200 units, indicating a benefit from increasing hidden states. This indicates that models with appropriate number of hidden states larger than 200 units and smaller than 1000 units perform better in the given task.

My manipulation with the sequence length indicates that a shorter sequence length could benefit model performance. By shortening the sequence length to 10 from 50, I could achieve ~ 100 less perplexity. I suspect that a normal length of word, which is usually close to 10, constraints the character sequence better than a length of 50.

Appendix: Answers to the questions

Q2 In the LSTM code, look at the function lstm(i, prev_c, prev_h). Relate these inputs to the equations in the paper 4. (i = . . . , prev_c = . . . , prev_h = . . .).

Variable ***i*** specifies the input vector for word/character. It is fed into a look up table from the input vector dimension to size of RNN network.
prev_c specifies the long term memory cell from last step
prev_h specifies the RNN hidden state from the last step.

Q3 What does the function create_network() return? Unrolled or not?
This function returns an nngraph gmodule, with a 2-layered RNN network embedded with LSTM unit. This is the unrolled version of the network. It is unrolled since the

function runs a for-loop and inserts ***next_c*** and ***next_h*** into the ***next_s*** sequentially, which is the network state in each layer.

Q4 What are model.s, model.ds, model.start_s? When is model.start_s reset to 0?
model.s records the current state of output for all words in the sequence.
model.ds records the gradient of model parameters w.r.t. output for all words in the sequence.
model.start_s records the state of output starting from the current evaluation in the sequence.

model.start_s is reset to 0 when function reset_state is called. reset_state is called at the beginning of training, validation, or testing. It is called when the current sequence of words has terminated and we are going to start a new sequence of words for model evaluation.

Q5 What form of gradient normalization is used?
The code applied gradient clipping for gradient normalization. The given code shrinks the norm of the gradient once the norm exceeds 5.

The code also applies regularization method with Dropout.

Q6 What optimization method is used?
The model is trained using gradient descent method. The gradient of error for each step is fixed at 1.

Q7 How do you deal with the extra output in the backward pass?
Assign dpred to tensor of zeros. During back-propagation in an nngraph gmodule, we only aim to calculate the gradient of parameters w.r.t. the gradient of the error, and we can safely force the gradient of other output to zeros.

Q1: See code:
nngraph_handin.lua

Word-level prediction: See code:
main_word.lua

Citation:

Mikolov, Tomas, Karafi´at, Martin, Burget, Lukas, Cernock`y, Jan, and Khudanpur, Sanjeev. Recurrent neural network based language model. In INTERSPEECH, pp. 1045–1048, 2010.

Zaremba, W., Sutskever, I., & Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.