The Model - RNN with LSTM Cells

The model chosen to translate English instructions to Unix commands is 2-layered RNN with LSTM Cells for hidden units, a representation of the model is shown below:

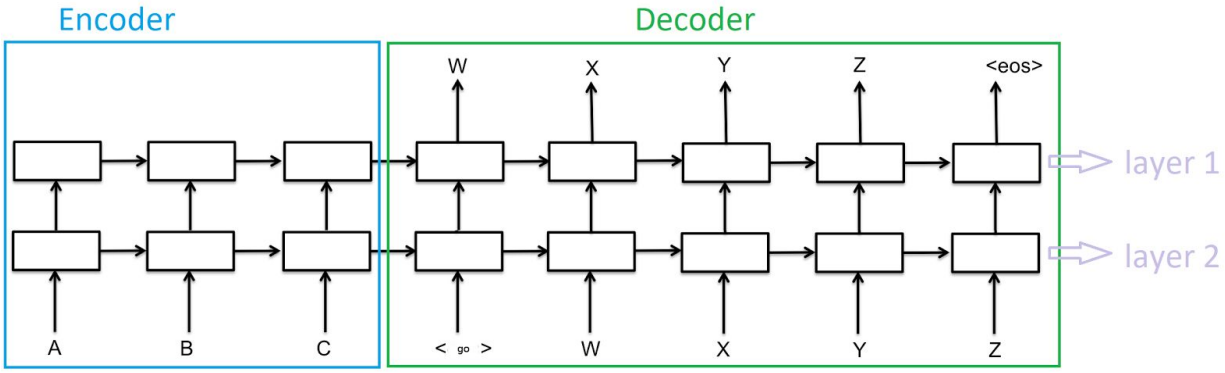


Figure. Model Sketch

Each small rectangle represent a hidden state, aka each LSTM cell. Each English Instruction (input sequence) and each Unix command (output/target sequence) need to be encoded into a **sequence of integers**. In my case, both input sequences and output sequences are **padded,** with special character 'PAD'**,** to length 50, since I don't have any instructions or unix commands that can be tokenized to more than 50 tokens. The padding is done because instructions and commands could both come in various lengths, and it's not very practical to construct a RNN model for every pair of (instruction_length, command_length). With padding, I can just build 1 model to handle the (50, 50) case.

For the input sequences, each value are within the range [0, 40,000) which corresponds to 40,000 different english vocabularies. This comes from an existing assumption that there are 40,000 common english words in machine translation[1]. **Each value in the output sequence of the Decoder is within the range [0, 4,000)** which corresponds to 4,000 unique vocabularies/tokens in the Unix commands of my interest, considering the scope of this project.

Continuing the discussion about the Encoder, a hidden state at some time T will see an input representing a special end-of-line character, signalling the end of this input sequence, and the decoder part of the RNN will begin. For each of encoder and decoder, the weights on the edge between any time t-1 and t is the same. Although the encoder and decoder can share weight, I am following the common practice of using different parameters. The model implemented is also bidirectional, which is not shown in the graph above. This decision is made since bidirectional model is considered to have more modeling power[2]. The output at time t-1 of the decoder will be

[1] TensorFlow, *"Sequence-to-Sequence Models"*
https://www.tensorflow.org/versions/r1.0/tutorials/seq2seq

[2] Leonardo Araujosantos, *"Machine Translation Using Recurrent Neural Networks"*
https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/recurrent_neural_networks/machine-translation-using-rnn.html

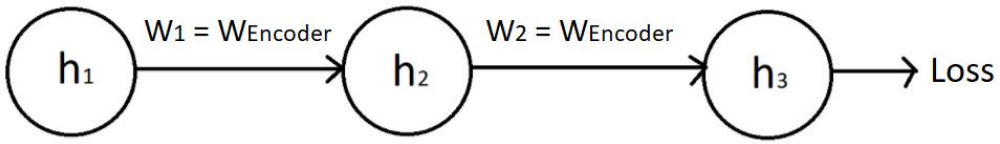
fed in as the input to the decoder at time t, e.g. output W at time=t-1 will become input at time=t, shown in the Decoder part of the picture above.

A hidden state at time = t, h_t, stores weighted combination of the input value at time = t and the information carried forward from the previous hidden state h_{t-1}. Each hidden cell itself is a LSTM which has logistic gates: **write gate, read gate, keep gate**. The write gate is set by the rest of the RNN and when it's set to be one, information/state is stored into it. Similarly, when the read gate is set to be 1, the state stored can come out and influence future states of the RNN. The keep gate will keep the state stored when it's set to 1, and will "forget" when it is set to 0. Choosing LSTM is because it has been proven to be great at remembering information for long periods of time, perform well in machine translation, and back-propagation (the algorithm used to train my RNN) through LSTM cells are also fairly efficient.

The reason behind choosing 2 layers is that I assume there are at least 2 types of information/structures for the RNN to learn in my translation problem. The first layer could learn the order of vocabularies, e.g "come back one level", 'back' should follow 'come'. The second layer, for example, could learn to distinguish between arguments and actions within an instruction: "find somefile.txt" where 'find' is an action and 'somefile.txt' is an argument to the unix command to be produced.

The Training Algorithm - Back Propagation

The algorithm used to train my RNN is Back Propagation, it is chosen because it is the best algorithm available to me. The Tensorflow code specifically uses Truncated Backpropagation to only consider propagation up to a specific number of steps for tractable computation. Bearing similarities to feedforward nets, back propagation on RNN is quite straight forward, a simplified example is shown below (removed some parts from model):



To constraint $W_1 = W_2 = W_{Encoder}$:
We need $\Delta W_1 = \Delta W_2$ for every update to the weights
So We compute $\frac{\partial Loss}{\partial W2} = \frac{\partial Loss}{\partial h3}\frac{\partial h3}{\partial W2}$, and $\frac{\partial Loss}{\partial W1} = \frac{\partial Loss}{\partial h3}\frac{\partial h3}{\partial h2}\frac{\partial h2}{\partial W1}$
And our $\Delta W_1 = \Delta W_2 = \eta \frac{1}{2}\left(\frac{\partial Loss}{\partial W2} + \frac{\partial Loss}{\partial W1}\right)$, where η is our learning rate
$W_1 \leftarrow W_1 - \Delta W_1$
$W_2 \leftarrow W_2 - \Delta W_2$

Initial Experimental Results

The training set has 5030 pairs of instruction and command, while the validation set has 298 pairs. The types of instructions are: finding a file/folder, listing what's in a directory or catting a file, go to certain directory, go certain levels up directory tree, and go back to previous folder. For details on what the instructions are, please see data_gen.py.

The validation set contains only instructions that have structures previously **unseen** in the training set to measure how good the model is at learning the underlying structures of this translation problem. Please refer to comments in data_gen.py for details on what unseen structures mean. A simple explanation is that given only "bring somefile.txt" and "give me somefile.txt" in training set, I want to test whether the model will learn that "bring me somefile.txt", a new structure, means the same thing and should map to the same command. I also wanted to see if the model learned how to distinguish argument (somefile.txt) from action (bring) by letting the validation set use file names never appeared in the training set.

Besides measuring the metrics defined in the proposal, the loss function, sampled softmax loss, should serve as a good general indicator of how the model is performing as it approximates how many parts/tokens of a translated command matches with the target command.

Figure. Training Loss & Validation Loss vs. Steps - 2-layered RNN with LSTM cells

Here are the measurements of the metric so far: **`python translate.py --decode`**
Metric 1: Overall parts matched per command in percentage: 65.85% in Training Set
Metric 2: Percentage of correct ordered command: 84.47% in Training Set
Metric 3(new): Number of complete matches: 1030 out of 5030 in Training Set, 20.48%

Metric 1: Overall parts matched per command in percentage: 48.34% in Validation Set
Metric 2: Percentage of correct ordered command: 88.59% in Validation Set
Metric 3(new): Number of complete matches: 15 out of 298 in Validation Set, 5.03%

Some preliminary examination of the result revealed that most translation errors occurred with not being able to correctly translate the ‘argument’, e.g. “can you access picture.jpeg please” → “if [[-d cpp.c]] ; then ls cpp.c ; else cat cpp.c ; fi;”, which shows that the argument “picture.jpeg” is incorrectly translated to “cpp.c”. And this provided some insights into what some of the future effort should focus on. **`python translate.py --encode --self_test`**

Though not ideal, this performance is much better than that of a Linear Regression Model, which I used as baseline. The encoding of Instructions and Commands remained the same, however in the baseline, we only train the W and b from Y = W*X + b, where X is the encoded instruction sequence and Y is the command sequence. From the loss below we can see that the model isn’t improving much as training goes on. The measured metrics for this baseline Linear Regression Model is also worse in general.

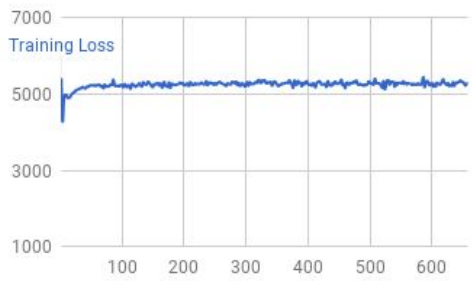


Figure. L2 Training Loss vs Training Epoch - Linear Regression Model

`python baseline.py`

M1: Overall parts matched per command in percentage: 9.78% in Training Set

M2: Percentage of correct ordered command: 95.81% in Training Set (Meaningless when M1 rate is so low)

M3: Number of complete matches: 8 out of 5030 in Training Set, which is 0.16%

Future Effort

As mentioned previously, there are still problems with the current model. Firstly, the model is often unable to correctly translate the “argument” portion, e.g. somefile.txt, of the English instructions to Unix commands. I suspect this has something to do with how I tokenize the English instructions and the commands. Currently, both the instructions and the commands are tokenized by white spaces and a set of special characters: *([.,!?\"':;)(])* This might not be the best way to tokenize since it makes intuitive sense to treat “somefile.txt” as a whole rather than tokenizing at “.” and produce [‘somefile’, ‘.’, ‘txt’]. However, for a command like “cd ../../..” tokenizing at the ‘.’ could help the model generalize its understanding about what ‘.’ means in the commands, so there needs to be more experiments. A second problem is whenever there is **unseen vocabulary** in the arguments in the instructions, the model cannot bring the argument to the outputted command, since every unseen vocabulary will be encoded to _UNK = 4. A possible way to improve is to have some heuristic function to guess whether the unseen vocabulary, e.g. ‘radom_file_03’, is an argument, then the model could encode it to some special integer value and make the output sequence also has that value, which then can be mapped back to the string of the unseen vocabulary. I also have explore using GRU cells instead of LSTM, but so far didn’t see significant performance difference.

All in all, there are many exploration, testing, and adjustments awaiting in the future.