Learning Text Sequences with Different Encodings

December 14, 2016

Abstract

We wrote an LSTM implementation that parametrizes the number of characters that each input ‘x‘ encodes – the n in $n-gram$ – and measured training efficiency and the perplexity of text generated by a model trained on 1 billion characters from Wikipedia. Results indicate that, even if we embed the one-hot-encoded bigram or trigram vector representations in a 128 element vector, training is more efficient on unigrams; the text that the model generates after 10 minutes on a MacBook Pro is more reasonable when the model is trained on the 27 element one-hot representation of unigrams. Tensorflow code is available at github.com/sshleifer/lstm/blob/master/code/tri_char_rnn.py.

Motivation for the LSTM Architecture

At a high level, the sequence modeling problem is to predict $x[t + 1]$ given $x[0 : t]$. Traditional feedforward neural networks are not well suited for this task because they require evenly lengthed inputs. The original solution to this was recurrent neural networks, which use loops (a neurons output feeds back into previous neurons as input) to allow information from the past to persist. Unfortunately, RNN overwrite the information they are propogating too quickly for many tasks. For example, an RNN might not remember the subject of a sentence after a few words, even though that information is important for predicting the sentences' verb. If we think of an unrolled RNN as a series of t repeating modules, each creating an output h[t] and taking input x[t], then we see that there is a lot of back propagation happening between the first module and the Nth module. LSTM, which are a specific kind of RNN, solve this long-term dependency problem by adding parameters to the repeated module so that there is a vector, the cell state, dedicated to persisting information through the network. The cell states are guarded from overwriting by a forget gate.

RNN send their output back While there has been considerable research showing that the memory of traditional Recurrent NetexLong Short-Term Memory models improve on Recurrent Neural Networks by adding parameters to each module that contain information from many steps ago. Each module has the ability to add or remove informaton from the Cell state, with regulation mediated by sigmoid layers whose outputted probabilities indicate how much of the previous hidden state and input x[t] to let into the cell state.

Encoding Tradeoffs The size of the vector needed to represent the one-hot-encoding of each element in the sequence is 27 ascii characters ^n, where n is the number of characters represented by each x. When these vectors get long, practitioners tend to embed them in a lower dimensional space. In our experiment, we one-hot-encode the n-grams and then embed them in a 128 element vector. So, for bigrams and trigrams, we are forced to learn an embedding and that embedding is lossy, two major disadvantages of moving past unigrams. The benefit of training on larger n-grams is that the full x sequences are shorter, so it may be easier to encode a dependency that is 10 characters away if that is only 5 modules away. To address this we also vary how far backwards errors can back-propagate.

Experiment Results

Overall, smaller n encodings generated fewer misspellings and more reasonable text than larger n models trained for the same amount of time. In the two trigram experiments we ran, we let a trigram model run for 1000 steps, first with 5 unrollings and then with 10 unrollings. Gradients for all models were clipped at 5. The 5 unrollings model took 8 minutes compared with 15 and scored roughly the same perplexity on the 1000 character validation set, but the text generated is not english. One major problem seems to be that the words end up being very long, but still since "tfbyti" was outputted we know we have a trigram with either tf or fb, both of which seem very unlikely.

"istfbytio ane nvennumguasfghellutof s dntrpubic sr t oy atwo deothen o zem ionype imonorgu" - Trigram LSTM trained for 15 minutes.

Bigram models were more succesful, largely because they ran 7x faster than trigrams per batch. The difference here might be the time and loss associated with training the emedding.

The most succesful, however, were unigram based models, which, within five minutes of birth could output more than 50% real english words (albeit shorter ones). With no embedding cost and a lower dimensionality input vectors (27), these models were able to complete 1000 steps in 20 seconds compared to 120 seconds for Bigrams and 15 minutes (900 seconds) for trigrams.

> "and theory boyer the zer ther one nine seven two zero zero peramented strong avaud blows than user rescre hter but election the jaries a consiletist famoust studrum amerapests pebhes to" - Unigram Model after 5 minutes training

If we increase the number of unrollings (how far errors can backpropagate) from 10 each iteration of the model involves a bit more learning and takes a bit more time, so we must compare performance per minute and hope that model improvement is roughly linear. With 3 unrollings, the unigram gets to 7.34

validation perplexity in the first minute. With 10 unrollings, 4.78 and with 20 unrollings, 4.93, thereby suggesting that 10 unrollings is somewhere near the optimum for our architecture if the first minute is a good proxy for the rest of training.

Conclusion and Next Steps

Unfortunately, getting acquainted with Tensorflow and implementing the text-generating model took 15 hours, and an additional 5 hours were spent trying to get a word based model to generate text. For this reason, we have not explored very far past the surface level results: that the unigram model produces more reasonable text than the bigram model which produces more reasonable text than the trigram model. Some future things to try include a) reducing or increasing the dimension of the embedding for the trigram/bigram model, which would allow me to test whether that is really the slow part, b) finding some word-based RNN code on the internet that generates text and see how it compares, c) experimenting with using the unigram model to measure the perplexity of the text generated by the other models, thereby allowing one metric to score each model's performance. d) repeat this set of experiments except let every model train for an hour or two.

References

[http://colah.github.io/posts/2015-08-Understanding-LSTMs/]

[https://github.com/tensorflow/tensorflow]