Word Prediction using Recurrent Neural Networks

Brian Burns
Udacity Machine Learning Engineer Nanodegree
February 15, 2017

Contents

Definition

Project Overview

Word prediction, or *language modeling*, is the task of predicting the most likely words following the preceding text. It has many applications, such as suggesting the next word as text is entered, as an aid in resolving ambiguity in speech and handwriting recognition, and in machine translation.

The generation of a likely word given prior words goes back to Claude Shannon's work on information theory (Shannon 1948) based in part on Markov models introduced by Andrei Markov (Markov 1913) - counts of encountered word sequences are used to estimate the conditional probability of seeing a word given the prior words. These so called n-grams formed the basis of commercial word prediction software in the 1980's, eventually supplemented with similar syntax and part of speech predictions (Carlberger 1997).

More recently, distributed representations of words have been used (so called *word embeddings*), in which each word is represented by a vector in some low-dimensional vector space (e.g. as a vector of 100 floating point numbers) - these can better handle data sparsity and allow more of the context to affect the prediction, as they can represent similar words as being close together in vector space. Hence, for instance, a cat can be predicted to 'sleep', even if the model was only trained on a dog sleeping, due to the similarity of the words 'dog' and 'cat'.

Feed-forward networks (FNNs) with a fixed amount of context were initially used (Bengio 2003), then Recurrent Neural Networks (RNNs) as training them became more feasible (Mikolov 2012). An RNN can make predictions based on arbitrary amounts of context because it effectively stores the history of everything it has seen in a hidden layer.

The neural network learns word vectors as it is trained, though pre-trained word embeddings are available, such as word2vec (Mikolov 2013) and GloVe (Pennington 2014) - these can be used in place of the input layer to save on training time. They have only become available recently due to the computational complexity of calculating them for large vocabularies (e.g. 400,000 words for GloVe).

The rest of the neural network learns the language model - it effectively learns a function that maps a sequence of input word vectors to the probability of the next word in the sequence, over the whole vocabulary.

The problem is a supervised learning task, and any text can be used to train and evaluate the models - we'll be using a million words from books digitized by the Gutenberg Project (Gutenberg 2016) for evaluation. Others use larger corpora, e.g. Google's billion word corpus (Chelba 2013). Depending on the problem domain, different corpora might be more appropriate, e.g. training on a chat/texting corpus would be more applicable for a phone text entry application.

Problem Statement

Given a sequence of words, predict the most likely following word.

We'll use an RNN to make the prediction - different architectures will be trained on a training set, tuned on a validation set, and tested on a hold-out test set - the results will be compared against a baseline n-gram model - it is anticipated that an RNN with proper parameters will be more accurate than the n-gram model.

Metrics

The metric used to evaluate the performance of the models will be *accuracy* - the average number of predictions where the highest probability next word matches the actual next word.

Analysis

Data Exploration

The training and testing data are obtained from ten books from Project Gutenberg, totalling roughly one million words (or 1.3 million tokens, which include punctuation marks like commas, periods, etc.) -

Text	Words	Letters / Word	Words / Sentence	Unique Words	Grade Level
1851 Nathaniel Hawthorne The House of the Seven Gables (G77)	96,217	5.2	21.2	22,214	12
1862 Victor Hugo Les Miserables (G135)	516,244	5.2	14.6	82,177	10
1865 Lewis Carroll Alice in Wonderland (G28885)	26,758	4.6	16.4	6,346	9
1883 Robert Louis Stevenson Treasure Island (G120)	62,826	4.7	16.9	13,894	8
1898 Henry James The Turn of the Screw (G209)	38,663	4.9	15.4	9,417	8
1899 Joseph Conrad Heart of Darkness (G219)	34,833	5.0	14.5	9,871	9
1905 M R James Ghost Stories of an Antiquary (G8486)	42,338	4.9	19.6	10,882	9

Text	Words	Letters / Word	Words / Sentence	Unique Words	Grade Level
1907 Arthur Machen The Hill of Dreams (G13969)	60,528	5.0	25.7	14,406	10
1908 Kenneth Graham The Wind in the Willows (G289)	54,160	4.9	16.8	13,102	9
1919 P G Woodhouse My Man Jeeves (G8164)	46,947	4.8	10.1	10,917	8
1920 M R James A Thin Ghost and Others (G20387)	29,311	4.7	21.3	7,767	8

The Gutenberg text number is listed in parentheses, and the texts can be found online - e.g. Alice in Wonderland can be found at http://www.gutenberg.org/etext/28885. The grade level is calculated using the Coleman-Liau Index (Coleman 1975), which is based on the average word and sentence lengths.

Some sample text:

> "Speak English!" said the Eaglet. "I don't know the meaning of half those long words, and, what's more, I don't believe you do either!" - *Alice in Wonderland* (shortest words)
>
> I went up and passed the time of day. "Well, well, well, what?" I said. "Why, Mr. Wooster! How do you do?" - *My Man Jeeves* (shortest sentences)
>
> From the eminence of the lane, skirting the brow of a hill, he looked down into deep valleys and dingles, and beyond, across the trees, to remoter country, wild bare hills and dark wooded lands meeting the grey still sky. - *The Hill of Dreams* (longest sentences)

Exploratory Visualization

For this project we'll use 50, 100, 200, or 300-dimensional word vectors from GloVe (Pennington 2014) - Figure 1 below shows some sample word embeddings projected to 2 dimensions using PCA - note how the adjectives, verbs, and nouns/agents are grouped closely together, indicating their similarity.

Algorithms and Techniques

Until recently, n-grams were considered state of the art in word prediction - neural networks surpassed them in accuracy in 2003, though at the cost of greater training time (Bengio 2003).

An RNN is able to remember arbitrary amounts of context, while n-grams are effectively limited to about 4 words of context (a 5-gram will give 4 words of context) - higher n-grams require increasing amounts of resources in terms of training data and storage space, as the resources required grow exponentially with the amount of context.

An RNN 'unfolds' to a neural network of arbitrary depth for training (called *Backpropagation Through Time (BPTT)*) - see Figure 2. It keeps track of the words it has seen through a hidden

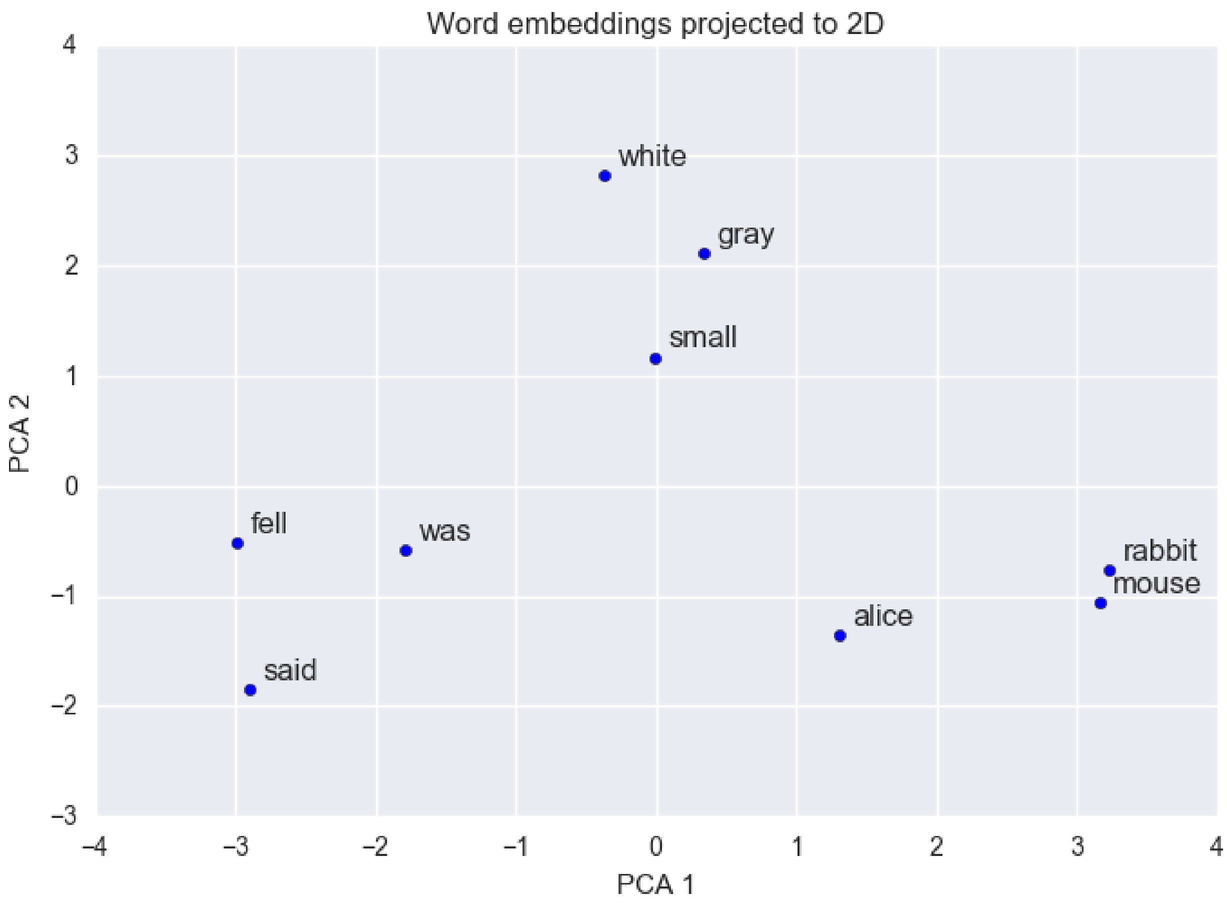

Figure 1: GloVe word embeddings

state at each step in a sequence - this is combined with the current word's representation (by addition) and the sum is then used to make predictions about the next word.

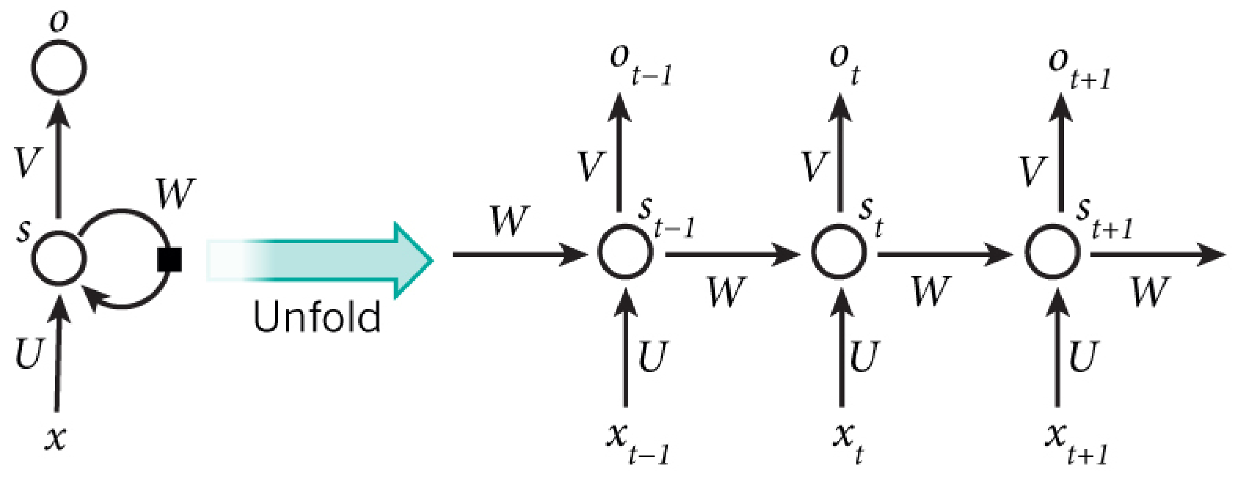

Figure 2: RNN (LeCun 2015)

The matrix U amounts to a table of word embeddings in a vector space of many dimensions - each word in the vocabulary corresponds with a row in the table, and the distance between any two words gives their similarity, once the network is trained. Alternatively, pre-trained word embeddings can be used in an input layer to save on training time.

The matrix W acts as a filter on the internal hidden state, which represents the prior context of arbitrary length.

The matrix V allows each word in the vocabulary to 'vote' on how likely it thinks it will be next, based on the context (current + previous words). The softmax output layer then converts these scores into probabilities, so the most likely next word can be found for a given context.

To learn the parameters for the matrices U, V, and W, we define an error (or loss) function to be the *categorical cross-entropy loss*, and use backpropagation through time to update the weights based on the amount of error between the outputs and the true values.

A *LSTM (Long Short-Term Memory)* RNN (Hochreiter 1997) works similarly, but has a 'memory cell' at the center, which has 'switches' for storing memory, or forgetting memory, and the state of the switches can be learned along with the rest of the parameters. This turns out to be easier to train than plain RNN's, which have problems with vanishing and exploding gradients, which can make them slow and difficult to train.

A *GRU (Gated Recurrent Unit)* RNN (Chung 2014) is similar to an LSTM, but has fewer parameters and is a bit easier to train.

Benchmark

For the benchmark model a simple trigram model will be used - this is a standard approach for next word prediction based on Markov models. A multidimensional array, indexed by vocabulary words, stores counts of occurrences of 3-tuples of words based on the training data. These are then normalized to get a probability distribution, which can be used to predict the most likely words following a sequence.

A more state of the art baseline would be a Kneser-Ney 5-gram model (Kneser 1995), which calculates 1- through 5-grams and interpolates between them, but implementing that algorithm was beyond the scope of this project.

Methodology

Data Preprocessing

The text files contain header and footer information separate from the actual text, including a Gutenberg license, but these are left in place, and read together into a single Python UTF-8 string. The string is split into paragraphs, which are then shuffled randomly - this allows the RNN to learn more context than if the text was split on sentences and shuffled.

The paragraphs are combined, and the text is converted to lowercase to increase the amount of vocabulary that can be learned for a given amount of resources. The text is then tokenized to the top *nvocab* words - the rarer words are **not recorded** - punctuation marks are treated as separate tokens.

The sequence of tokens is then split into training, validation, and test sets - the test set was set to 10,000 tokens, and the validation set to 1% of the training set, which amounts to around 13,000 tokens out of 1,300,000 tokens.

Implementation

The RNN was implemented with Python using Keras (Chollet 2015) with a TensorFlow backend (Abadi 2015) - TensorFlow is a general-purpose neural network toolbox, while Keras is a wrapper around TensorFlow that simplifies its use. Different architectures and parameters were explored, and compared based on their accuracy scores. The most promising model was then used to find the final testing accuracy score.

The basic architecture of the RNN is an embedding input layer, one or more hidden layers, a densely connected output layer, and a softmax layer to convert the outputs to probabilities.

The initial approach used sequences of integer tokens for input and one-hot encoded tokens for comparison with the output - this proved infeasible memory-wise - the training data was only *O(nelements * n)*, roughly 1 million * 10 ~ 100MB, but the labels were *O(nelements * nvocab)*, roughly 1 million * 10,000 ~ 1e11 bytes = 100 GB, too much for a 16GB system.

The next approach used Python generators for the input sequences and output labels - the neural network is able to accept generators instead of arrays through Keras' `fit_generator` method, but in tests with small datasets this proved to be about 10x slower than using the one-hot arrays.

The final approach changed the output loss function from *categorical_crossentropy* to *sparse_categorical_crossentropy* - this allowed the output labels to be specified as arrays of integer tokens, instead of one-hot encoded tokens, which allowed them to fit easily into memory, with *O(nelements)* ~ 10MB.

During preprocessing the texts were tokenized to the top *nvocab* words, then a word embedding matrix *E* was built from the complete GloVe word embedding array (which contains 400,000 words), and set as the initial weights for the neural network's embedding layer.

For the training step, the training sequence of roughly 1.3 million tokens was fed to the network, which was trained for a certain number of epochs - depending on the RNN configuration each epoch took around 30 minutes.

During training Keras output the progress and a generated sequence before/after each epoch - both the training and validation loss and accuracy are shown during training and at the end of the epoch, e.g. -

```
Training model...
Train on 1283291 samples, validate on 12963 samples
Epoch 0 generated text: of lightly malvern gurgling anybody despairing pointing fray 'm amid antechamber 'm fight sooth
Epoch 1/10
1283291/1283291 [==============================] - 1207s - loss: 5.4187 - acc: 0.1518 - val_loss: 5.1304 - val_acc: 0.1665
Epoch 1 generated text: . `` you will have a , and i have not a man . '' she had a little ,
Epoch 2/10
1283291/1283291 [==============================] - 1207s - loss: 5.1610 - acc: 0.1669 - val_loss: 5.0386 - val_acc: 0.1734
Epoch 2 generated text: ; and it is a man , i have not done to say to the . it was a of
Epoch 3/10
1283291/1283291 [==============================] - 1206s - loss: 5.1034 - acc: 0.1708 - val_loss: 5.0139 - val_acc: 0.1780
Epoch 3 generated text: . he knew that she had no one of the of , of which the first man was , ''
```

Neural networks can overfit to the training data if they are trained for too many epochs - this is prevented by an early stopping trigger, which will stop training if the validation accuracy does not improve for a certain number of epochs.

Another way to improve the accuracy of the model is to apply *dropout* during training, which randomly sets a fraction of input units to 0 at each update, which helps to prevent overfitting. This reduces the number of parameters needed to learn a fully connected layer, and increases robustness by forcing the model to learn multiple ways to learn the same patterns (Srivatsava 2014).

For evaluation, plots were made of the training and validation loss over the training epochs - see Figure 3 below - this also helps diagnose overfitting, which is indicated by the loss flattening out and starting to increase.

Refinement

The basic RNN algorithm remained the same over the course of the project, but many parameter values were explored.

Results

Model Evaluation and Validation

The RNN architecture has many parameters which can affect the speed of training and the accuracy of the model - the total parameter space has over 2,000,000 combinations to explore (see table below). This does not even including the optimizer parameters, like learning rate or decay.

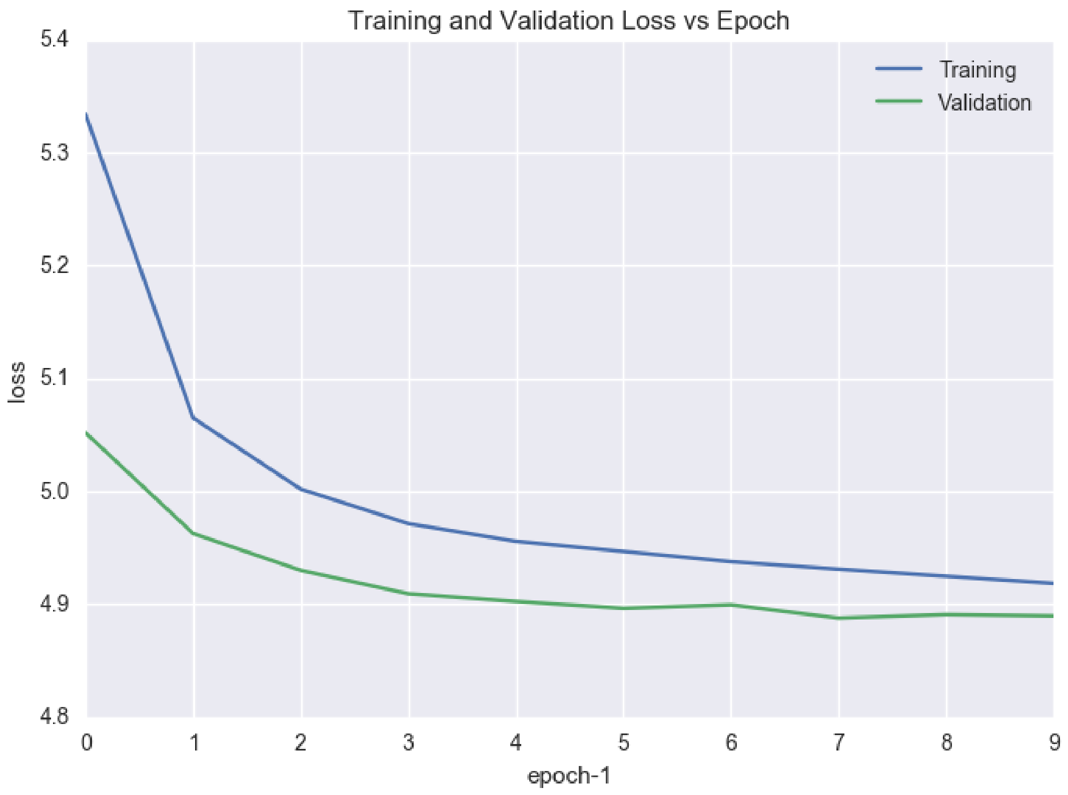

Figure 3: Training and validation loss plot

Parameter	Description	Values	Number to explore	Initial Value	Final Value
nepochs	number of epochs to train model	1-10+	1	3	10
patience	stop after this many epochs of no improvement	1-10	1	3	3
layers	number of RNN layers	1-3	3	1	1
dropout	amount of dropout to apply after each layer	0.0-1.0	5	0	0.1
nvocab	number of vocabulary words to use	5k,10k,20k,40k	4	10k	10k
embedding_dim	dimension of word embedding layer	50,100,200,300	4	50	200
trainable	train the word embedding matrix?	True/False	2	False	True
nhidden	size of the hidden layer(s)	50,100,200,300	4	50	200
n	amount to unfold recurrent network	1-10	5	5	10
rnn_class	type of RNN to use	Simple, LSTM, GRU	3	Simple	GRU
optimizer	optimizing algorithm to use	sgd, rmsprop, adam, etc	7	sgd	adam
initializer	random initializer for matrices	uniform, normal, etc	4	glorot_uniform	uniform
batch_size	number of training sets per batch	16,32,64	3	32	32
total			2,419,200		

With each model taking a few hours to train, only a small subset of the parameter space could be explored. The approach taken was to optimize each parameter individually in turn - such a search may get stuck in a local optimum, but a more complete search would be too expensive.

The initial parameters chosen led to a test accuracy of 14.7%. Many experiments were performed to tune the hyperparameters and try to improve the accuracy:

Parameter	Values tried	Best value	Best validation accuracy
initial			0.140
dropout	0,0.1,0.2,0.3	0.1	0.160
nlayers	1,2,3	2	0.167
nhidden	40,50,60	50	0.167
nepochs	3,10	10	0.171
embedding_dim	50,100	100	0.185
nhidden	50,100	100	0.199
nlayers	1,2,3	1	0.200
trainable	true,false	true	0.222
n	5,10	10	0.232
rnn_class	simple, lstm, gru	gru	0.232
optimizer	adam, sgd, rmsprop, adagrad, adadelta	adam	0.232
initializer	glorot_uniform, uniform, normal	uniform	0.234
nvocab	10k,20k	10k	0.234
embed+nhid	100,200	200	0.236

The final parameters chosen led to a test accuracy of 24.4%, an absolute improvement of 9.7% over the initial parameters.

The test accuracy gives an indication of the robustsness of the model in the domain of 19th and early 20th century English novels and stories - for other domains the accuracy would be lower. For instance, using the model for word prediction in a texting app would have lower accuracy due to the lack of modern words, abbreviations, and phrases in the training data.

Justification

The benchmark trigram model achieved a final accuracy of 18.7%, while the best RNN architecture achieved a test accuracy of 24.4%, an absolute improvement of 5.7%, though at the cost of much greater training time (8 seconds vs 8.2 hours).

An accuracy score of 24.4% indicates a fairly useful model - if the user was presented with the best guess for the next word as they are entering text then the correct word would be shown nearly a quarter of the time - showing the 2nd and 3rd best guesses also would increase the usefulness of the model even more.

For training, the benchmark trigram model has a space complexity of O(v^3), with v=number of vocabulary words, though the actual space used will be much less, as not all word combinations are valid. The time complexity for training is roughly O(t), with t=number of training words, as each training step is simply two O(1) dictionary lookups and an addition operation.

The space complexity for predictions is the same, O(v^3), and the time complexity for predicting the next word in a sequence is O(v), since a prediction requires two O(1) dictionary lookups, then iterating over all key values to find the probabilities of the next word in the sequence. This could be improved to O(1) overall by using a priority queue to store the word probabilities instead of a dictionary of word counts, which would add a little time to the training phase.

For training, the neural network model has a space complexity of approximately O(vw+2wh+h^2), which is dominated by the word vector lookup table, which is roughly O(vw), with w=word vector dimensions - this can be much less than the trigram model. The time complexity for training is roughly O(etnv), with e=number of epochs and n=amount of context. Each epoch, t sets of n words are fed through the matrices, which are relatively cheap multiplication operations, but the final step is a softmax computation over the entire vocabulary v, which involves exponentials, so this term dominates the training. The backpropagation step is roughly the same number of computations, so the overall complexity remains the same. This is much larger than the trigram training time complexity of O(t).

For predictions the space complexity is the same, O(vw), and the time complexity is O(nv), which consists mostly of the softmax layer computations.

So overall, the neural network would be quite usable on a phone, for instance, once it was trained - with space requirements on the order of 1 million units and time requirements on the order of 100,000 calculations. For comparison, the trigram model could require 100 million units (more or less, depending on the amount of training data and whether low word counts were eliminated), and 10,000 calculations. In other words, the neural network would be able to fit more information in smaller space, at the cost of increased prediction time.

Conclusion

Free-Form Visualization

A plot of accuracy with increasing amount of training material is shown below - it appears likely that both the n-gram and RNN model would keep improving with more training data -

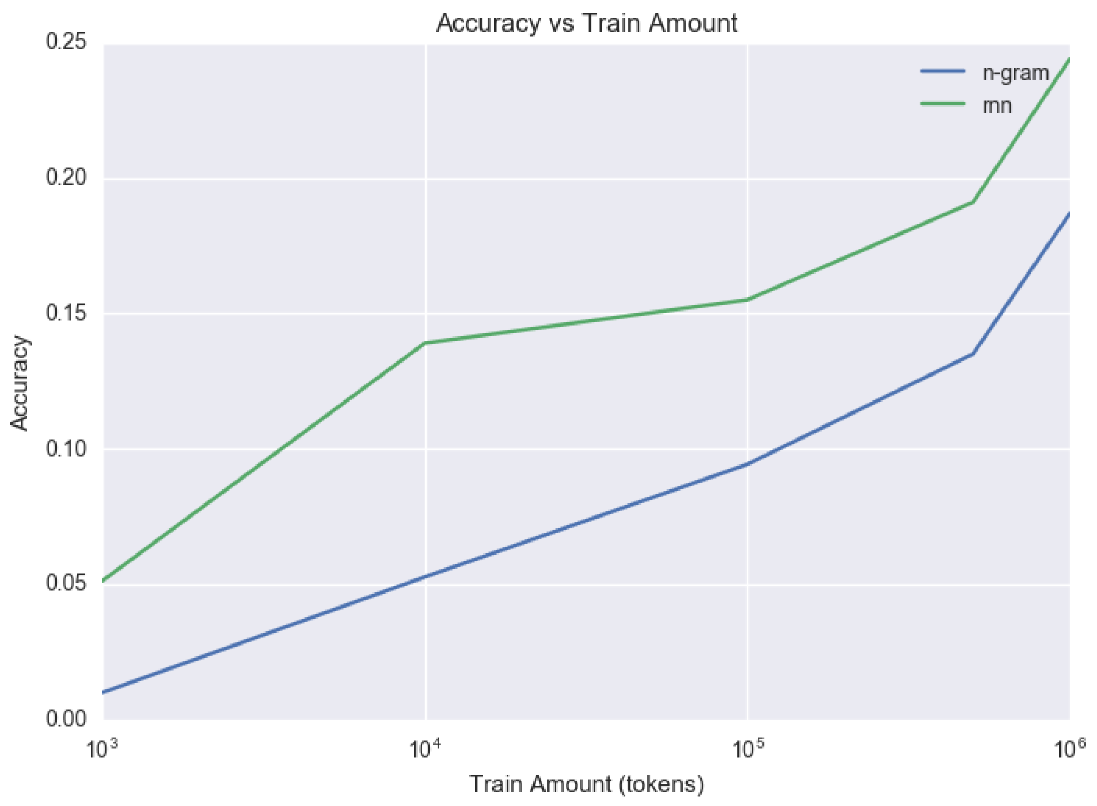

Figure 4: Accuracy vs Train Amount

Some examples of text generated by the different models:

Trigram

- strangely he must have felt in that direction , like house ” built in the book which the
- saw all three made the most crushed of men , each on the of this hesitating nature .
- henslow , he has never been so , ” he replied : – he says , ’no money , but on
- surprised at this point they reached the first place , because one knows whom , in that one
- killed or conquerors . the rascal sprang from the vague mystery of the treasure of abbot thomas ’s

RNN

- and then the old woman had been a little girl . the old women of women . the first
- “ i have no other words . he had a good deal , ” said the doctor , and
- from the window . the door opened . the door closed , and he was the door . “ i

- said he , “ i am not , and i could not have a good thing . ”
- he added , “ i know what you are doing , ” said the captain , “ you

In general, the RNN is more grammatically correct - though neither make a lot of sense - and is able to handle opening and closing quotation marks across phrases of several words, along with appropriate opening and closing phrases like “said the captain”.

Reflection

The RNN is more accurate than the trigram model because it can retain more history than just the previous 2 tokens, and using word vectors instead of discrete tokens for each word allows it to generalize and make predictions from similar words, not just exact matches.

The RNN takes longer to train (e.g. 8 hours vs 8 seconds, about 4000x longer), but would easily be able to fit into memory for applications like phone text entry - it also performs word predictions a bit slower than the trigram, but not enough to be noticeable to an end-user.

The most interesting aspect of this project to me was seeing how well the RNN learned to use quotation marks correctly, and that its accuracy looked likely to just keep improving with more training data - it makes me wonder what the limits of a simple model like this are - would it just keep learning more complex sequences and produce even more realistic sounding text with more training and larger matrices? And what improvements could be made to the model to make it even more accurate, and faster?

Improvement

The simplest way to improve the accuracy of the model might be to train it with more data, as seems apparent from the graph showing increasing accuracy with training data - e.g. you could feed it 10 million tokens, though this would require 10 times the training time (e.g. 80 hours). One could try adding more and more training data, up to a billion tokens, time allowing (that would require 8,000 hours, or nearly a year with this setup), and see if the test accuracy continued to improve, or leveled off at some point.

It would also be possible to improve the accuracy slightly by training it longer - the accuracy was still improving a small bit even after 10 epochs, though at some point the model would start overfitting and the accuracy would begin going down.

It’s possible that the training could be sped up also - the softmax output layer is the most computationally expensive part of the training, and there are some ways of approximating it, such as hierarchical softmax or differentiated softmax (Ruder 2016). This would make it possible to improve the accuracy and perform more experiments with the architecture, so would be worthwhile to try to incorporate.

There are other aspects of the architecture which I haven’t explored, such as stateful or stacked RNNs, which might improve the accuracy further.

References

(Abadi 2015) Abadi, Martin, et al. "TensorFlow: Large-scale machine learning on heterogeneous systems." Software available from tensorflow.org, 2015.

(Bengio 2003) Bengio, Yoshua, et al. "A neural probabilistic language model." Journal of Machine Learning Research, Feb 2003.

(Bird 2009) Bird, Steven, Edward Loper and Ewan Klein, "Natural Language Processing with Python (NLTK Book)." O'Reilly Media Inc., 2009. http://nltk.org/book

(Carlberger 1997) Carlberger, Alice, et al. "Profet, a new generation of word prediction: An evaluation study." Proceedings, ACL Workshop on Natural Language Processing for Communication Aids, 1997.

(Chelba 2013) Chelba, Ciprian, et al. "One billion word benchmark for measuring progress in statistical language modeling." arXiv preprint arXiv:1312.3005, 2013.

(Chollet 2015) Chollet, Francois, "Keras." https://github.com/fchollet/keras, 2015

(Chung 2014) Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." (2014) ______________link

(Coleman 1975) Coleman, Meri; and Liau, T. L., "A computer readability formula designed for machine scoring." Journal of Applied Psychology, Vol. 60, pp. 283 - 284, 1975.

(Gutenberg 2016) Project Gutenberg. (n.d.). Retrieved December 16, 2016, from www.gutenberg.org.

(Hochreiter 1997) Hochreiter, Sepp, "Long Short-Term Memory." Neural Computation 9(8), 1997.

(Kneser 1995) Kneser, R. & Ney, H. "Improved backing-off for m-gram language modeling." Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Detroit, MI, volume 1, pp. 181-184. May 1995.

(LeCun 2015) LeCun, Bengio, Hinton, "Deep learning." Nature 521, 436 - 444 (28 May 2015) doi:10.1038/nature14539

(Markov 1913) Markov, Andrei, "An example of statistical investigation of the text Eugene Onegin concerning the connection of samples in chains." Bulletin of the Imperial Academy of Sciences of St. Petersburg, Vol 7 No 3, 1913. English translation by Nitussov, Alexander et al., Science in Context, Vol 19 No 4, 2006

(Mikolov 2012) Mikolov, Tomas, PhD thesis, "Statistical Language Models Based on Neural Networks." http://www.fit.vutbr.cz/~imikolov/rnnlm/thesis.pdf, 2012

(Mikolov 2013) Mikolov, Tomas; et al. "Efficient Estimation of Word Representations in Vector Space." arXiv:1301.3781, 2013.

(Pennington 2014) Pennington, Jeffrey et al.. "GloVe: Global Vectors for Word Representation." http://nlp.stanford.edu/projects/glove/, 2014

(Rosenblatt 1957) Rosenblatt, F. "The perceptron, a perceiving and recognizing automaton." Project Para. Cornell Aeronautical Laboratory, 1957.

(Ruder 2016) Ruder, Sebastian. "On word embeddings - Part 2: Approximating the Softmax." http://sebastianruder.com/word-embeddings-softmax, 2016.

(Shannon 1948) Shannon, Claude, “A Mathematical Theory of Communication.” The Bell System Technical Journal, Vol. 27, July 1948.

(Srivatsava 2014) Srivatsava, Nitish, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting.” Journal of Machine Learning Research 15, 2014.