Machine learning Engineer Nanodegree

CAPSTONE PROJECT REPORT

Naren Doraiswamy

December 1st, 2017

Definition

Project Overview:

Ever wondered how google always almost predicts the question you ask in its search engine? Fascinating , isn't it? The same question boggled me from a long time and after spending quite a lot of time on the internet researching and reading through awful lot of materials and then finally taking

the machine learning nanodegree course, i could learn the finer nuances about machine learning and also deep learning. My capstone project will be based on a machine translation model using recurrent neural networks which can be applied to speech recognition The applications of language models are two-fold: First, it allows us to score arbitrary sentences based on how likely they are to occur in the real world. This gives us a measure of grammatical and semantic correctness. Such models are typically used as part of Machine Translation systems. Secondly, a language model allows us to generate new text (I think that's the much cooler application). Training a language model on Shakespeare novels allows us to generate Shakespeare-like text.

The goal of statistical language modeling that is being built is to predict the next word in textual data in given context, thus we are dealing with sequential data prediction problem when constructing language models. Still, many attempts to obtain such statistical models involve approaches that are very specific for language domain - for example, assumption that natural language sentences can be described by parse trees, or that we need to consider morphology of words, syntax and semantics. This project is partly inspired by andrew karpathy's post (http://karpathy.github.io/2015/05/21/rnn-effectiveness/) and research paperhttp://www.fit.vutbr.cz/research/groups/speech/publi/2011/mikolov_icassp2011_5528.pdf

Problem Statement:

The basic aim here is to predict the word that will be inputed next by training the model on some training data. Sequential data prediction is considered by many as a key problem in machine learning and artificial intelligence. Developing a machine translation model that predicts the next word that will be typed is one of the intelligence problem that can be solved and it takes the next step towards developing smarter systems Even the most widely used and general models, based on n-gram statistics, assume that language consists of sequences of atomic symbols - words - that form sentences, and where the end of sentence symbol plays important and very special role. The dataset that will be used here for this project will be text from fiction novel. The input data is fragmented into sentences and then words by using the NLTK module and a vocabulary list of most common words is created. A word to vec dictionary is created and then used as inputs for the further computations. Different datasets can be used for training the model for different purposes. Generation of TV show scripts has become one of the most fascinating and advanced sequential data prediction problems that has been solved by intelligence.The higher abstract library , Keras with tensorflow in the backend will be used for our project.In the end, as we input a single character, my algorithm must be able to output or predict another character which would eventually build into a meaningful sentence.

Metrics:

Since the output layer is a fully connected layer which implies the probabilities of the next character to be outputted, we will use the softmax classifier along with the categorical cross entropy for loss generation during the training. So we will one hot encode our targets(y), and compare them with the generated probabilities of the output characters(y_) and use the formulae y*ln(y_). Since we are using the LSTM's instead of the vanilla RNN's , the problem of vanishing gradients is taken care of, but the exploding gradients problem still persists. Hence we will use the gradient clipping method which will have a threshold where anything beyond this value will clip it to the threshold. The LSTM is trained with mini-batch Stochastic Gradient Descent and I have used Adam optimizer(per-parameter adaptive learning rate methods) to stablilize the updates.Also RMSProp can also be used though adam is more adaptive in nature.

The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing.(https://arxiv.org/abs/1412.6980). The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.
Adam realizes the benefits of both AdaGrad and RMSProp.Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance).Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages.The initial value of the moving averages and beta1 and beta2 values close to 1.0 (recommended) result in a bias of moment estimates towards zero. This bias is overcome by first calculating the biased estimates before then calculating bias-corrected estimates.

Analysis

DATA EXPLORATION:

```
In [4]: view_sentence_range = (0, 10)
        print('Dataset Stats')
        print('Roughly the number of unique words: {}'.format(len({word: None for word in text.split()})))
        paras = text.split('\n\n')
        print('Number of paragraphs: {}'.format(len(paras)))


        sentences = [sentence for para in paras for sentence in para.split('\n')]
        print('Number of lines: {}'.format(len(sentences)))
        word_count_sentence = [len(sentence.split()) for sentence in sentences]
        print('Average number of words in each line: {}'.format(np.average(word_count_sentence)))

        print()
        print('The sentences {} to {}:'.format(*view_sentence_range))
        print('\n'.join(text.split('\n')[view_sentence_range[0]:view_sentence_range[1]]))
```

```
Dataset Stats
Roughly the number of unique words: 9699
Number of paragraphs: 1040
Number of lines: 8038
Average number of words in each line: 12.522144812142324

The sentences 0 to 10:



the golden bird

a certain king had a beautiful garden, and in the garden stood a tree
which bore golden apples. these apples were always counted, and about
the time when they began to grow ripe it was found that every night one
of them was gone. the king became very angry at this, and ordered the
```

The screenshot of the jupyter notebook shows the code written for the data exploration purpose. The dataset after being read into the script is split into sentences and then into words and number of words,lines and paragraphs in the dataset are found out. Also the average number of words in each line are found out. With a little bit of digging into dataset, we find out that there are no absurd observations about it.

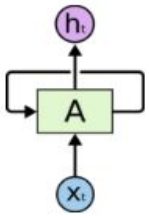
Exploratory Visualization:

There isn't any visualizations in this particular project as the dataset is mainly text data and the only visualization that popped up for me was the word embeddings which is not being used in this project. Though, i would like to build upon this in my future explorations with natural language processing techniques that i will work upon soon.

Algorithms and Techniques:

Like i mentioned before, the main algorithm that i will be using is the recurrent neural networks(RNN), specifically the Long short term memory (LSTM). The traditional neural networks can't keep a tab on the previous inputs that they received. This is a huge problem where the present input depends on the previous input dataset and with no memory of the previous inputs or states, the networks wouldn't be a problem solver at this point.

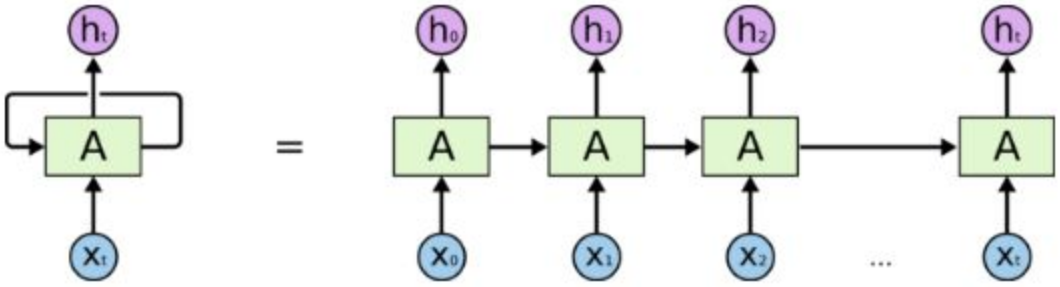
Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.



Recurrent Neural Networks have loops.

In the above diagram, a chunk of neural network, A, looks at some input X_t and outputs a value h_t. A loop allows information to be passed from one step of the network to the next.

These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



An unrolled recurrent neural network.

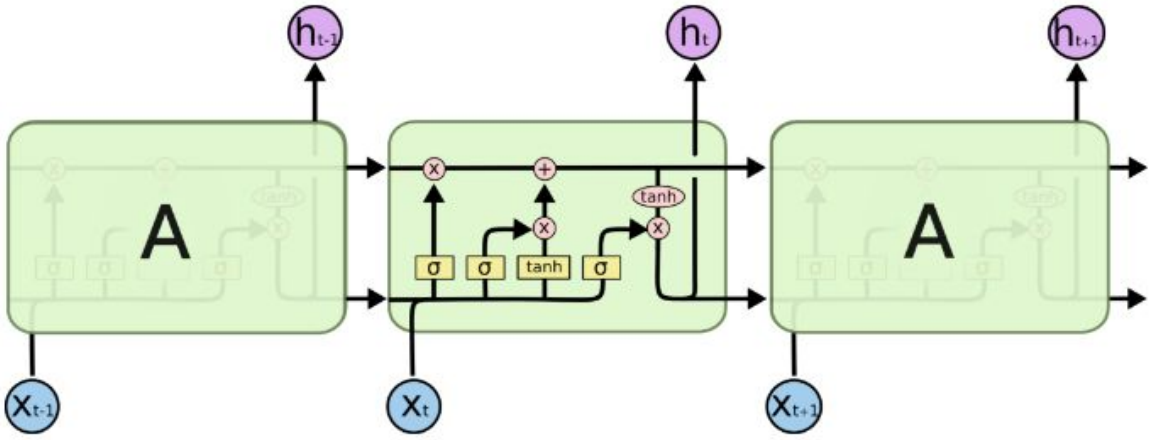
In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning… The list goes on. And that is the reason why i am using the recurrent neural network to solve the problem.

The RNN's are great, but they do have a few problems with them like the vanishing and exploding gradient problem with them. Hence, a slight variation in the network architecture is made to remove those irregularities and there are various versions of main RNN networks. The most famous one is the LSTM. Along with them, the Gated recurrent units(GRU), bi-directional recurrent networks are a few others being in use today.

LSTM Networks:

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!.

LSTMs also have this chain like structure like that of RNN's, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



The repeating module in an LSTM contains four interacting layers.

The cell state C_t is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let

through. A value of zero means “let nothing through,” while a value of one means “let everything through!”An LSTM has three of these gates, to protect and control the cell state.The first is called the forget gate, the next is the input gate layer and third is the update gate. The parameter discussion is done in the jupyter notebook.

Benchmark:

The gihub repo is taken as reference/benchmark while i started. I have finished taking up my deep learning nanodegree and have achieved similar results for my project. The implementation set a particular benchmark loss after being trained for a long time,but the trial account on the floydhub limits the GPU usage upto an hour. The loss can be reduced further , if the network is trained for a longer time, and produce a more robust and generative model. The result obtained here are pretty satisfying though and produces similar results like the one considered as the benchmark.

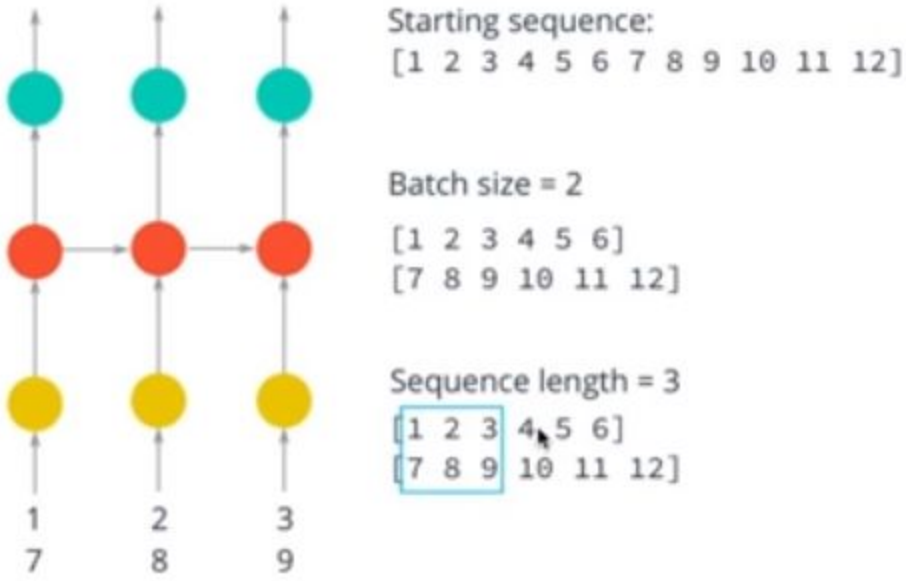
Methodology

Data Preprocessing:

As told in the data exploration section, there are no abnormalities in the dataset. The only modification i have done is reducing the length of the dataset as it took way too long for completing a single epoch during the training process. With this modification and varying other parameters altogether in the model, i have been able to achieve tangible results in the project.Since our input is text and we cannot just input them to our model as it understands only numbers and not text, we'll load the text file and convert it into integers for our network to use. We will create a couple dictionaries to convert the characters to and from integers. Encoding the characters as integers makes it easier to use as input in the network.

```
# convert each unique character to an integer and vice versa
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))
```

The code used to generate the dictionary is shown above. Each character is now converted into an integer which will be fed into the algorithm. Next, we need to standardize the input sequences that we will feed into our model. We will need our sequences to have same lengths. This can be done by predefining a fixed length window that we will use to slide through our text by one step at a time. Each slide will create one sequence in our input. In fact, we can even adjust our sliding step. To make it easy to understand, a pictorial representation is shown below.



The starting sequence is the input integers from the dictionary and that is converted into sequence length based on the number of batches required which is a parameter to be validated.

Implementation:

All the basic details have already been given in the previous section as to which algorithm and metrics are being used and how they are going to be implemented. For building the neural network, we will use the keras framework with tensorflow in the backend. We will convert our text dataset into a dictionary of integers and then fed into the neural networks after batching

them. We will create a batch of 1000 and sequence steps of 50 and after every layer we will have a dropout layer which will drop out 20% of the connections randomly without any bias to generalize the model well .For the final output layer we create the dense layer with categorical cross entropy as the loss function and adam optimizer. The weights after every epoch is saved in hdf5 format and later used during the sampling process. The model is trained for 5 epochs and the loss drops from 2.5848 to 1.8872 after training for 5 full epochs.

Now that the training is done, the text generation is remaining which is our final part. For this we will sample every single word and the trained model will predict what the next character is. The trained weights are loaded into the model and a random integer is selected which is indeed a character and the model predicts a few characters. The character with the best probability will be outputted. The outputted character will be taken as the the next character for input and the prediction is made again. The process iterates until the range is reached and we now have our computer generated text based on the trained model of a fiction novel.

Refinement:

For the parameters used , the batch size of 500 has been used which could have been reduced to 64 or 128 for finer results but the lack of gpu free hours and enormous amount of time needed to train a single epoch with these smaller batch sizes led me to use higher values for them. But for even more refined results , we can use smaller batch size and smaller sequence lengths. Also the training epochs used can be higher for more accurate results.

Results

Model Evaluation and Validation:

The results obtained from the model is put up below in the screenshot along with the sampling code. The model is able to generate quite a few correct words though the sentence is not really formed that well. The most frequently used words like the, he , she is learnt very well by the model the relations between the subjects too is well portrayed in the results and hence the model can be trained with the optimum parameters like mentioned previously to actually come up with something really good.

```
'''Load the trained weights'''
weights = "weight_saving_at_4_and_ 1.8872.hdf5"
#weights = "weight_saving_at_003_and_ 1.8215.hdf5"
model.load_weights(weights)
model.compile(loss='categorical_crossentropy', optimizer='adam')
start = np.random.randint(0, len(X)-1)
pattern = X_[start]
print (''.join([indices_char[value] for value in pattern]))
# generate characters
for i in range(150):
    x = np.reshape(pattern, (1, len(pattern), 1))
    x = x / float(nb_chars)
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    result = indices_char[index]
    seq_in = [indices_char[value] for value in pattern]
    sys.stdout.write(result)
    pattern.append(index)
    pattern = pattern[1:len(pattern)]
print ("\nEnd.")
```

```
re thou hangest!’

and the head answered
 to the woung was to the wood and said: ‘i will soon toon the wound be a gord to the wood and said, ‘i wil
l soon to the wound be a gord to the wood an
End.
```

Justification:

I have worked on a similar project in my deep learning nanodegree on generating tv scripts and results obtained here are pretty satisfying while comparing both the results. The parameters used in that project were pretty optimised for the benchmark while the parameters used here weren’t really exactly the most optimised ones to get the optimum results but they have actually performed better than expected

Conclusion:

As told before in the data visualisation part, there are no visualizations in the particular project though we can create a word embeddings models in the future work. Now that we have seen the results can be really impressive with the RNN’s, especially with the LSTM's tweak. So after every epoch is completed, the checkpoint file which holds the weights at that particular instant will be saved in to the repository. Using every file , we see an improvement in the prediction of the words. initially the word spaces In the first checkpoint iteration we see that the network just learns the word spaces and then later on it tries to generate sentences with vague virtuoso and then in the next iteration it learns some most used words like the he , she, its etc.. and now more english like text appears… I have used the checkpoint file from the 5 epoch and these weights are used. and if it is trained for a longer duration with lesser batch size it will be able to generate The punctuations, quotations, names etc in a very seemingly natural way... So basically the model first discovers the general word space and then starts to rapidly learn the actual words. And if trained longer , the longer words can be generated too along with the

theme of the training set. But we need enormous amount of data and very good computational efficiency to achieve that.