Recurrent Neural Networks and Long Short-Term Memory Cells

Edward Myers and Stijn Collart

EECE 5644: Introduction to Machine Learning and Pattern Recognition

Abstract

In this paper, we examine *Recurrent Neural Networks* (RNNs), including the motivation for their creation, the types of data they excel at modeling, and how they differ from traditional deep learning networks. We then examine undesired behavior of RNNs, including the "vanishing gradient" problem, whereby only recent sequences within the data are "remembered" by the network. We then examine the use of *Long Short-Term Memory Cells* (LSTMs) that attempt to solve this problem, including their mathematical basis and overall structure. We further mention techniques to improve RNNs, including selective attention. Finally, we use the TensorFlow machine learning framework to train and implement RNNs with LSTMs and show our results trained on text datasets to generate novel sequences with similar styles to the training data.

Nomenclature

RNN	Recurrent Neural Network
LSTM	Long Short Term Memory Cell
t	Current data index
n	Length of data sequence x
θ	Parameters of network
$J_t(\theta)$	Loss function for parameter *theta* at index t
$g(\cdot)$	Non-linear activation function; typically σ, $tanh$, or ReLU
x_t	Input data at time t
y_t	Output data at time t
s_t	Node state at time t
b	Bias term
U,V,W	Weight matrices
η	Learning rate parameter

I. Introduction

Neural networks are becoming increasingly popular for classification and prediction. However, they are severely lacking for sequences of data, such as auditory speech, word prediction and generation, language translation, and any time-varying data sequence. In this paper, we first explore the basic structure and mathematical basis behind neural nets, and more specifically recurrent neural net architectures for deep learning on sequences of data. We talk about how recurrent neural nets are suited for these tasks, and work by passing its own state back into itself as an input for the next sequence index. We go over the methodology on how to train these networks and how to calculate their loss functions. We also talk about some of the limitations inherent in RNNs, and two methods for how to solve them: LSTMs and selective attention.

II. Related Work

Neural Networks

There is a great deal of research available that goes over every aspect of neural networks and deep learning architectures. Some important papers in the field introduced the first perceptron,[2] the creation of practical deep neural networks,[3] and more efficient methods for training deep neural networks.[4][5]

RNNs and LSTMs

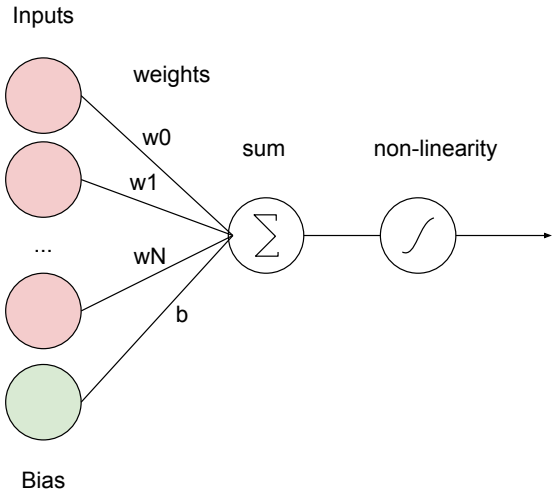
Recurrent neural networks have also been around for a fair amount of time. They were originally defined in 1986 by Michael I. Jordan,[6] and have been gaining popularity in recent years. One paper describes the vanishing gradient by R. Pascanu.[7] Many others use techniques such as LSTMs to perform amazing tasks on data sequences including Alex Graves' handwriting model,[1] Google's WaveNet,[10] and others.

III. Method

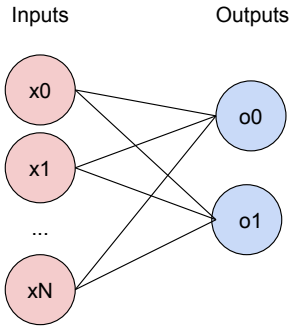
Neural Networks

Perceptrons can be mathematically described by multiplying an input vector by a weight matrix and adding a bias. This is then summed and put into a nonlinear function[a] $g(\cdot)$. This can be found in Eq. (1) and Figure 1a. Because of the nonlinear activation function $g(\cdot)$, neural networks are able to draw non-linear decision boundaries, expanding the potential number of classification tasks they can perform.

$$Y = g(XW + b) \tag{1}$$



(a) MISO Perceptron



(b) MIMO Perceptron

Figure 1: Perceptrons

Perceptrons can also have multiple inputs and outputs, as shown in Figure 1b. Note that while in the figure the summing junction and non-linearity is not shown, it is implied in the diagram.

In a deep neural network, multiple layers of perceptrons are combined, as in Figure 2. Deep neural networks are becoming increasingly popular, as they have the capability to create increasingly complex models and excel at certain tasks. In order to determine what weights work best for a given task, neural networks use the technique of *backpropagation*. In order to do this, first a loss function is defined, $J(\theta)$ with parameter θ, which represents the parameters (e.g. weights) in the network. The gradient of $J(\theta)$ is

[a] Which sometimes also acts to bound the data, such as the sigmoid (0,1) and the hyperbolic tangent function (-1,1)

calculated, and θ is changed to be in the opposite of this direction, in order to minimize loss. This equation can be seen in Eq. (2).

$$\theta := \theta - \eta \frac{\partial J(\theta)}{\partial \theta} \tag{2}$$

The parameter η in Eq. (2) represents the learning rate of the function, which is a scalar that modifies the gradient of the loss function. It effectively decides how quickly the network can modify its weights (change its beliefs). Later in the paper we will discuss the effects of various learning rates on the models.

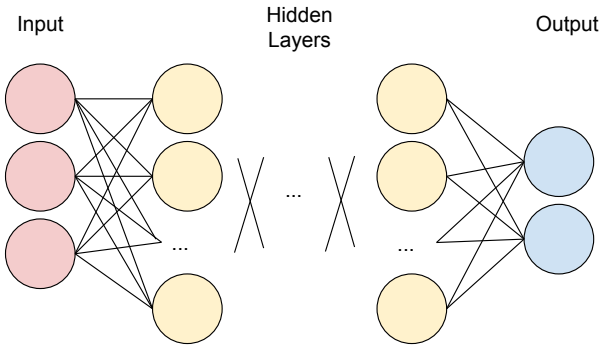


Figure 2: A general example of a neural network

Recurrent Neural Networks

Recurrent neural networks build off the concepts used in deep neural networks. In essence, a recurrent neural network is a deep neural network, but expanded so as to work better with time series data. A traditional deep neural network ran on time series data wouldn't have any input from previous data sequences. In an RNN, a node's last state and last output are fed back into itself for use in predicting the next output. In this way, each node has another input, which is used to model time-series dependencies. A diagram of this can be seen in Figure 3a, and it's state output described in Eq. (3).

$$s_1 = g(Wx_0 + Us_0) \tag{3}$$

Another way of representing the basic RNN structure is by "unfolding" one node, to better show it across each sequence step in the input data. This unfolded representation is visualized in Figure 3b. Next, how do we train a RNN? The same way as a normal neural network, with backpropagation. In this case, our loss function is defined as the sum of the loss for each sequence output, as in Eq. (4).

$$J(\theta) = \sum_{t=1}^{N} J_t(\theta) \tag{4}$$

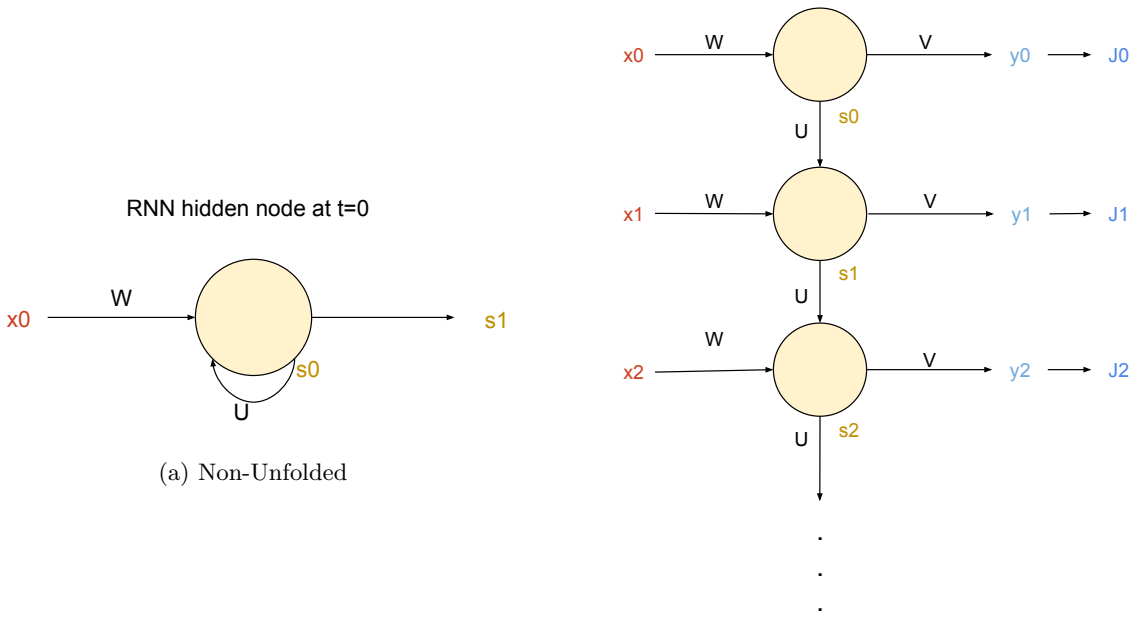
In order to perform backpropagation, we must first find the gradient. From Eq. (4), we can find its basic form, Eq. (5).

$$\frac{\partial J(\theta)}{\partial W} = \sum_{t=1}^{N} \frac{\partial J_t(\theta)}{\partial W} \tag{5}$$

To find the loss for a single time step (for example, the loss at time $t = 2$), we expand it as such

$$\frac{\partial J_2(\theta)}{\partial W} = \frac{\partial J_2}{\partial y_2} \frac{\partial y_2}{\partial s_2} \frac{\partial s_2}{\partial W} \tag{6}$$

Now we need to expand the part with s_2. Because the state at $t = 2$ depends on the state at $t = 1$. Expanding this, and generalizing for any t:



(a) Non-Unfolded

(b) Unfolded

Figure 3: An RNN node

$$\frac{\partial J_2(\theta)}{\partial W} = \sum_{t=1}^{N} \frac{\partial J_t}{\partial y_t} \frac{\partial y_t}{\partial s_t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial W} \tag{7}$$

Now, we can find the gradient and back propagate in order to properly train the model with a labeled data set. However, now we come into the "vanishing gradient" problem. The issue can be illustrated by looking at this component within the gradient for the last time step:

$$\frac{\partial s_t}{\partial s_k} = \frac{\partial s_n}{\partial s_{n-1}} \frac{\partial s_{n-1}}{\partial s_{n-1}} \dots \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0} \tag{8}$$

And each of these terms is less than one, and is specified by the following equation:

$$\frac{\partial s_n}{\partial s_{n-1}} = W^T diag[g'(W s_{j-1} + U x_j)] \tag{9}$$

Because there are so many terms being multiplied, each of which are less than one (due to the activation function used), this makes the contribution of further time steps smaller than that of nearer time steps. In effect, this causes a bias towards short range dependencies within a sequence, as these errors are most prominent in our gradient. There are a few ways to minimize this effect. One is by choosing different activation functions that have higher derivatives (e.g. ReLU). This helps keep the g' term larger. You can also change initialization (e.g., identity instead of normal). This can prevent the W, weight matrix, from shrinking terms too much. However, we are most interested in using gated cells, specifically *Long Short Term Memory* as a way to mitigate the vanishing gradient problem.

Long Short-Term Memory

As mentioned in the previous section, LSTM cells were designed to mitigate the vanishing gradient problem. They use logical operations in order to selectively remember long term dependencies within a data set. The way an LSTM works is by selectively modifying the cell state, seen in Figures 5 and 4[b] as cascading

[b]See Appendix A for key to Figure 4

downward after passing through each LSTM node (from state s_t to state s_{t+1}). First, there is a element-wise multiplication for "forgetting" information, and then an element-wise addition for "remembering" new information.

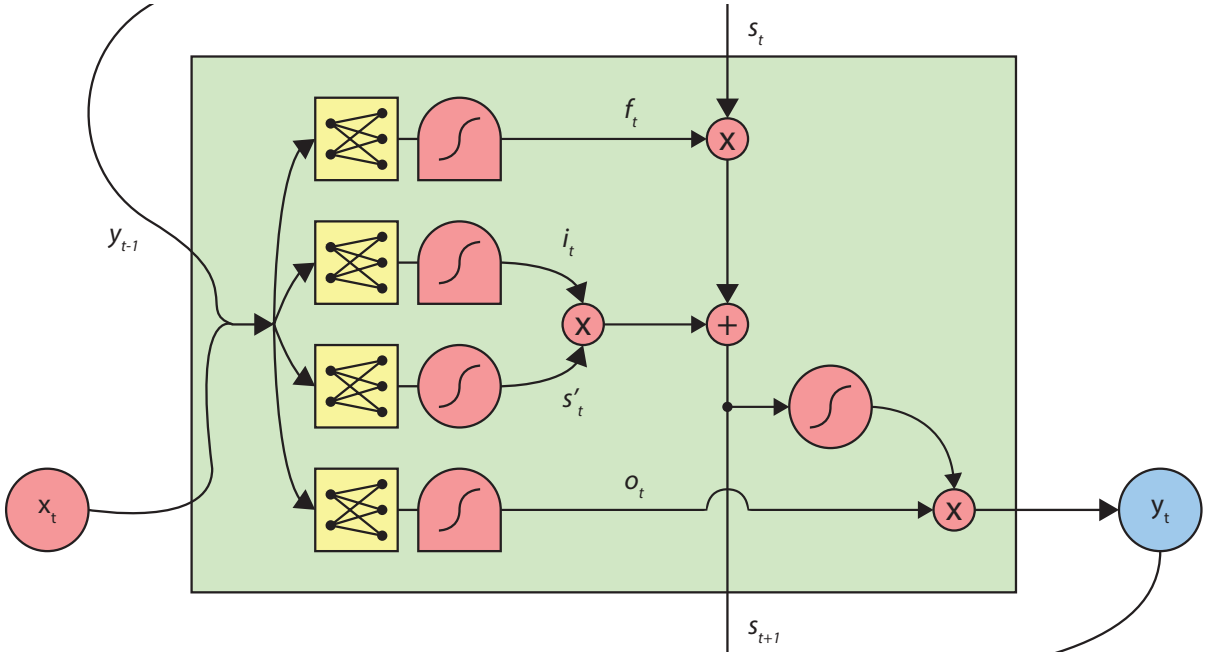


Figure 4: A "close up" view of the inner workings of an LSTM cell

The first step on the state's journey is called the *forget gate layer*, and is represented by f_t (Eq. 10). It is controlled by element-wise multiplication, which is decided by a neural network with sigmoid activation function. This in effect acts as a filter; because the output is between 0 and 1, it acts to selectively block (forget) incoming parts of the state. An example of this would be while training a language model, forgetting the number of subjects referred to once a new subject is introduced.

$$f_t = \sigma(W_f \cdot [y_{t-1}, x_t] + b_f) \tag{10}$$

The next step is to determine what information should be added to the state. To do this, a sigmoid neural net layer determines what data to input (Eq. 11), and a tanh network determines candidate values, c'_t to add to the state (Eq. 12). Finally, these are added to the next state value.

$$i_t = \sigma(W_i \cdot [y_{t-1}, x_t] + b_i) \tag{11}$$

$$s'_t = tanh(W_s \cdot [y_{t-1}, x_t] + b_s) \tag{12}$$

The state updates step (forgetting and remembering) are represented by Eq. 13.

$$s_t = f_t \cdot s_{t-1} + i_t \cdot s'_t \tag{13}$$

Finally, the output for this sequence number (time step) must be determined. This is calculated by the sigmoid neural network represented by Eq. 14. The new cell state is first put through a *tanh* function, to ensure that it is between (-1,1). Then, the sigmoid neural net "filters" the modified state, determining which aspects of it go into the output. In our language model example, it would choose aspects such as: verb tense, subjects referred to, pronouns, etc.

$$o_t = \sigma(W_y \cdot [y_{t-1}, x_t] + b_o) \tag{14}$$

$$y_t = o_t \cdot tanh(s_t) \tag{15}$$

In this way, LSTM networks are able to intelligently remember and forget long-term dependencies within sequences. Because of the logical operations determined by trained neural nets, relevant information can be completely preserved across many more steps than would be possible by a simpler recurrent neural network without LSTM cells.

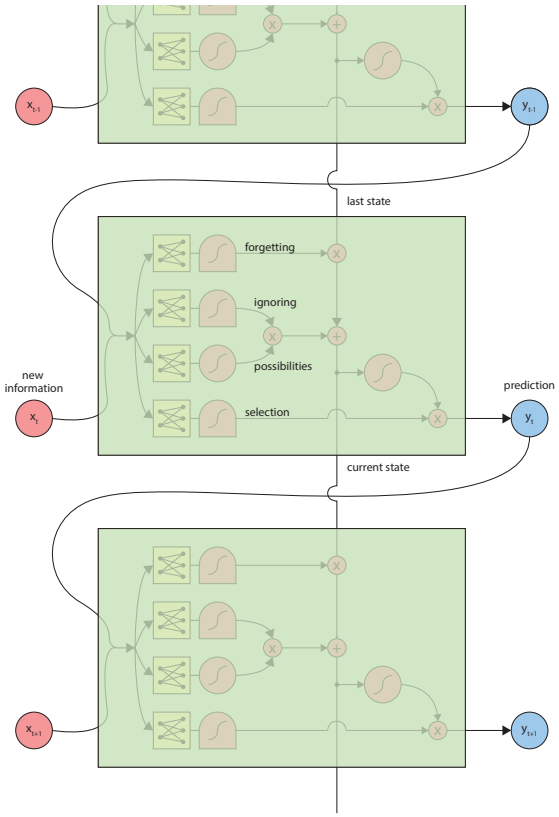


Figure 5: A network of LSTM nodes, showing flow of input data, predictions, and state.

Attention

So what is the next step of LSTM recurrent neural networks? How do we further optimize the network for better results? The answer is attention. In general, attention aims to tell the RNN what subsection of data to look at out of a larger set at a given time. It is outside of the scope of this paper, but in the Graves handwriting model,[1] they used soft windows to determine how many timesteps to use for generating each character while generating handwriting data.

TensorFlow

Tensorflow is a deep learning framework that we extensively employed while developing our code. It allows us to use GPU acceleration, performs differentiation automatically, and allows for code reusablility and extensibility. Many other deep learning frameworks exist, including Caffe, Keras, theano, torch, Pytorch, and dmlc mxnet.

Tensorflow contains a robust library of optimization algorithms, cost functions, and machine learning methodologies, allowing us to quickly process our data in many different ways and explore the results.

IV. Results

We tried a number of different approaches towards generating novel sequences of text. Our first attempt employed cross entropy with a gradient descent optimizer. This method was successful at recreating sentences

and strings from relatively shorts blocks of text (less than 2000 words) from short stories (specifically Aesop's Fables), but performed poorly with larger text files. Examples of shorter recreations are shown below.

Initially, we found that our models would typically tend to either underfit and produce nonsense, or overfit and directly reproduce sentences from the training material. To try and get around this issue, we tested a wide variety of parameters and optimizers to see what would produce the best results. Our first attempt used a learning rate of 0.001, two hidden layers with 512 nodes, a sequence length of three, and the aforementioned optimizer and cost functions.

We found marked improvement by changing the sequence length from three to five. Doing this means that the network now looked back five words rather than three. Predictably this greatly improved loss and sped up the computation. Next, we tested to see how learning rate affected the computation speed and loss. We began with a rate of 0.001. When increased to 0.01 and 0.1, The initial changes in loss were dramatic; it reduced the number of iterations to increase accuracy by 10% from the thousands to the hundreds. The issue at these learning values (and also somewhat at 0.001) was that the loss would start oscillating violently at certain points. This was likely because the learning rate was not sensitive enough to adjust further. Intuitively, it is easy to initially improve upon random starting values, but the fine tuning of the model requires sensitive changes. Trying this with lower learning rates, such as 0.0005, yielded slightly better results but made the computation take much longer. This research into the effects of learning rates led us to use an optimization algorithm that uses a dynamic learning rate.

We used RMSProp and Adam to explore the effects of the variable learning rate. Both of these algorithms store a decaying average of past squared gradients, while Adam also stores the average of the gradients. Using this information, it alters learning ratings for different weights to ensure that it will not get stuck, or become too aggressive in its changes. In this way it assures that you will not see the large oscillations in loss seen by static gradient descent, while increasing the chances of convergence.

We found suitable results with large blocks of text using a starting learning rate of 0.001, 3 hidden layers with 512 nodes, a sequence length of 15 and using the sequence-to-sequence model with a RMSProp optimization algorithm. These algorithms allowed us to learn far larger blocks of text (millions of characters) and converge on a model in reasonable time (on the order of hours). These models, given their large dictionaries, were capable of producing intelligible sentences when they were not overfitted. Examples of generated sentences are shown below. These examples are drawn from a mixture of models including the entire Lord of the Rings trilogy and a complete list of all Dungeons and Dragons Spells.

Lord of the Rings: "Aragorn"

```
aragorn. 'do not open far away, that
they time for the best journey from the lands of which we came with?
and i knew no days, and i have never moved so, and i would
now



abandon me
```

The above sentence is generated from the word "Aragorn". The first half is nonsense, although it obeys some basic rules of English sentence structure. The second half is a far more readable (and somewhat ominous) novel narrative section.

```
jolly horrible! poor! ' he hewed in the edge of eye of hand,
but it's trying to read it. there's he not so by me: i suspect
i'll say to us, ' said gandalf. 'are you quick?
```

This sentence is very close to being a fully correct English sentence, and contains no elements directly lifted from the books. The formatting is poor but this is likely due to the structure of the text file.

D&D Spells: "Spell"

```
spell ensures privacy for attempt a +1 enhancement bonus on damage. you concentrate.


spell allows like to animate plants, transmute your creatures.

the transmuted spell becomes smarter. the spell grants a +4 enhancement bonus to attack
   ( an animal spell(
```

The above three sentences are all generated D&D spells. While not perfect English, they all have clear novel meaning.

We employed a number of different methods of sentence generation. When looking at the output tensor containing a probability distribution of all possible words, we chose either the most likely, or picked a random word according to the distribution. When using the most likely word you always end up with the same sentences being produced every time, which reduces your chances of finding well constructed novel sentences greatly. When selecting randomly according to the distribution, you can generate long sections that have a higher chance of being unintelligible, but a much higher chance of being novel. All of our included sentences were generated according to the random selection method.

V. Conclusion

In summary, we went over the basis of how deep neural networks, and specifically recurrent neural networks, are created and trained. We discussed how the use of long short-term memory cells can be used to mitigate the vanishing gradient problem and create powerful networks that work on time-series data. We discussed further improvements to be made, including the use of selective attention. Finally, we explored the implementation of these networks using Tensorflow, and how adjusting parameters affected learning and model performance.

Appendix A

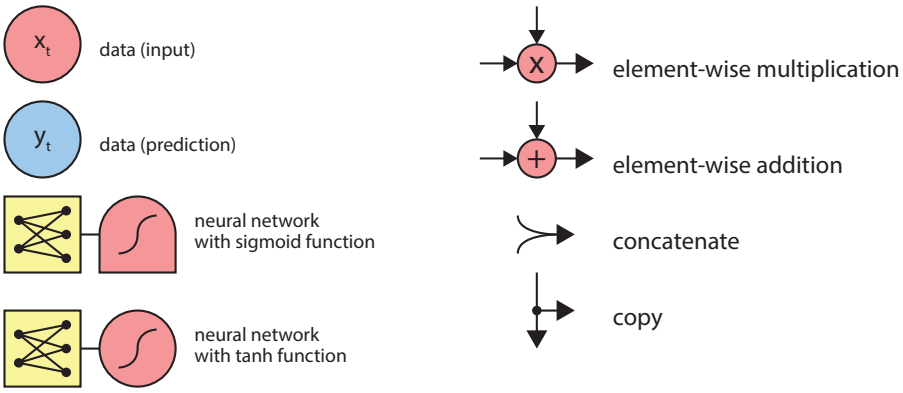


Figure 6: Key: components inside LSTM cell

Appendix B

ShortStoreCode.py

Learns short stories with basic RNN structure and allows creation of sentences, based on edureka tutorials[12][13]

LargeDataTrainer.py

LargeDataTrainer.py - Learns larger data sets with all parameters and structures modifiable, based on GOT chapter generation[14]

LargeDataRun.py

Runs models and generates sentences according to probability distribution, based on GOT chapter generation[14]

Acknowledgments

Brandon Rohrer's Video on conceptually understanding LSTMs, MIT's *LSTMs in Tensorflow video*, and Colah's blog post on understanding LSTM networks[9] were invaluable resources for understanding the basics of how LSTMs and RNNs work. Additionally, Edureka's Tensorflow[12] tutorials were very useful for getting started with TensorFlow. This paper by Wang et. al. was useful to find important contributions in history for NNs and RNNs.[11]

References

[1] A. Graves, "Generating Sequences With Recurrent Neural Networks," *eprint arXiv:1308.0850*

[2] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review*, Vol. 65, No. 6, 1958

[3] B. B. Le Cun, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," *Advances in neural information processing systems*, Citeseer, 1990

[4] G. E. Hinton, S. Osindero, and Y.W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, 18(7):15271554, 2006.

[5] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint* arXiv:1207.0580, 2012.

[6] M. I. Jordan, "Serial order: A parallel distributed processing approach. Advances in psychology," 121:471495, 1986.

[7] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training Recurrent Neural Networks," *eprint arXiv:1211.5063*

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, 9(8):17351780, 1997.

[9] C. Olah, "Understanding LSTM Networks," *Colah's Blog*, August 2015. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[10] A. van den Oord, K. Simonyan, N. Kalchbrenner, S. Dielman, O. Vinyals, A. Senior, H. Zen, A. Graves, K. Kavukcuoglu, "WaveNet: A Generative Model for Raw Audio," *eprint arXiv:1609.03499v2*

[11] H. Wang, B. Raj, "On the Origin of Deep Learning," *eprint arXiv:1702.07800v4*

[12] edurekaIN, YouTube, 05-Jul-2017. [Online]. Available: https://www.youtube.com/watch?v=yX8KuPZCAMo. [Accessed: 10-Apr-2018].

[13] Roatienza, "roatienza/Deep-Learning-Experiments", GitHub. [Online]. Available: https://tinyurl.com/y7y5leul [Accessed: 10-Apr-2018].

[14] Zackthoutt, zackthoutt/got-book-6, GitHub. [Online]. Available: https://github.com/zackthoutt/got-book-6/blob/master/README.md. [Accessed: 15-Apr-2018].