Recurrent Neural Networks on the Parity Problem

Thesis by
Aditya Gudibanda

In Partial Fulfillment of the Requirements for the
degree of
Bachelor of Science in Computer Science and Mathematics

YALE UNIVERSITY
New Haven, Connecticut

2017
Defended May 4
ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Dr. Uri Shaham, as well as my mother, Dr. Vidya Rao F.R.C.A., and my brother, Varun Gudibanda, a math major at Carnegie Mellon University. I would also like to thank Dr. James Aspnes in his role as the DUS of the Yale Computer Science Department, and Yale University, for allowing this research to take place.
In this thesis, we present the results of applying a Recurrent Neural Network (RNN) to the parity problem. Given a bit sequence, the parity task is to output whether the number of ones is even or odd. This is equivalent to taking the XOR of all the bits. This is a highly nonlinear problem that is equally sensitive to all elements of the input sequence, and it is a problem that generates great interest in understanding the way that neural networks can learn these nonlinear dependencies. Due to these dependencies, a regular feedforward neural network is unable to learn the parity function without memorizing the outputs for all elements of the input space. We present the failure of a feedforward to learn the parity function by demonstrating its failure to generalize to sequences of longer length, as well as its inability to achieve greater than a 50% accuracy even on the training set, when the input space is too large. Next, we demonstrate the successful training of RNNs on the parity function. We analyze several runs of the RNN with hyperparameter optimization, and we also analyze runs when the RNN does and does not converge, to analyze the differences. We report on interesting behavior during the training of the RNN, and suggest explanations for why this behavior is observed. We then identify some further explanations for how the RNN is learning the parity function by observing the differences between failed and successful runs. Future work will seek to analyze how RNN’s learn different sequences, such as the Fibonacci sequence, arithmetic sequences, geometric sequences, and more advanced sequences such as the digits of \( \pi \).
TABLE OF CONTENTS

Acknowledgements .............................................................. iii
Abstract ................................................................................... iv
Table of Contents ................................................................. v
List of Illustrations ............................................................... vi
Chapter I: Introduction ............................................................ 1
  1.1 Previous Work ................................................................. 1
Chapter II: Feedforward Neural Networks .................................. 2
Chapter III: Recurrent Neural Networks ....................................... 5
  3.1 Architecture and Hyperparameter Optimization ..................... 5
  3.2 Testing Generalization ....................................................... 6
Chapter IV: Hidden State Analysis .............................................. 10
  4.1 Visualization of Hidden States ........................................... 10
Chapter V: Discussion and Future Work ..................................... 15
Bibliography ............................................................................ 18
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Feedforward Neural Network Epoch by Epoch Accuracy</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Feedforward Neural Network Epoch by Epoch Loss</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>Successful RNN Epoch by Epoch Accuracy</td>
<td>6</td>
</tr>
<tr>
<td>3.2</td>
<td>Successful RNN Epoch by Epoch Loss</td>
<td>7</td>
</tr>
<tr>
<td>3.3</td>
<td>Failed RNN Epoch by Epoch Accuracy</td>
<td>8</td>
</tr>
<tr>
<td>3.4</td>
<td>Failed RNN Epoch by Epoch Loss</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Average cell state value across epochs in successful RNN run.</td>
<td>11</td>
</tr>
<tr>
<td>4.2</td>
<td>Average hidden state value across epochs in successful RNN run.</td>
<td>12</td>
</tr>
<tr>
<td>4.3</td>
<td>Average cell state value across epochs in failed RNN run.</td>
<td>13</td>
</tr>
<tr>
<td>4.4</td>
<td>Average hidden state value across epochs in failed RNN run.</td>
<td>14</td>
</tr>
<tr>
<td>5.1</td>
<td>A sample hidden state across epochs of a node in the successful RNN run.</td>
<td>16</td>
</tr>
</tbody>
</table>
INTRODUCTION

We begin with an introduction to the parity problem, also known as the XOR problem. Formally, given $x \in \{0, 1\}^n$, the parity is $p(x) = \sum_{i=1}^{n} x_i \pmod{2}$, where $x_i$ is the $i$th element of the sequence $x$. In the setting of machine learning, we train neural networks with the sequences $x$ as input, and with the parities $p(x)$ as the outputs.

1.1 Previous Work

The application of neural networks on the parity problem is a well studied problem of theoretical interest. The parity function is the least linearly separable problem, in the sense that each input bit has the ability to completely change the output. Previous work has shown that a regular feedforward neural network with one hidden layer and a sigmoid activation function requires $O(n)$ nodes in the hidden layer to learn the parity function. It has been shown that choosing the weights and biases of such a network can be interpreted as solving a system of linear equations [Setiono, 1996]. However, using variants on the traditional neural network architecture, such as allowing for different, non-monotonic activation functions, and by connecting neurons in cascade, drastic improvements have been found, even allowing the parity problem to be solved by a neural network with 2 nodes in its hidden layer [Myron Hohil and Smith, 1999]. However, it is important to note that since feedforward neural networks required a fixed-length input, the number of distinct input patterns, and therefore the complexity of the problem, increases exponentially with the length of the input. This also does not fit humans’ intuitive definition of learning the problem, which involves generalization to larger sequences.

Hence, Recurrent Neural Networks (RNNs) are a natural next tool to solve the parity problem, and is the one that we study in this report. The type of RNN we focus on in this paper is specifically those with Long Short Term Memory (LSTM) cells, which are able to hold data in memory, intuitively making them amenable to the parity problem, which one would think would necessitate the holding of the current parity in memory in order to truly learn the parity function. RNNs have been able to learn the parity function [Nal Kalchbrenner and Graves, 2016] but it is not yet well understood how they are able to do so.
Chapter 2

FEEDFORWARD NEURAL NETWORKS

We first trained and tested a regular feedforward neural network with one hidden layer, and another feedforward neural network with two hidden layers. All layers had 200 hidden nodes. We tested the neural networks on sequences of length 30. We chose the training and testing data by choosing 10000 random numbers without replacement between 0 and $2^{30} - 1$ and converting them to binary. We then split these randomly into training and test sets, with an 80-20 training-test split. This ensured that the training and test sets were disjoint, to ensure that we were testing the network’s ability to generalize.

In the figures below we report the accuracy and loss per epoch. In both plots, the red represents the training data, and the green represents the testing data.

As one can see, the feedforward fails to generalize to the testing data, even after roughly 500 epochs, and even after achieving close to perfect accuracy on the training data. This confirms the known fact that feedforward neural networks are unable to learn the parity function.
Figure 2.1: Feedforward Neural Network Epoch by Epoch Accuracy.
Figure 2.2: Feedforward Neural Network Epoch by Epoch Loss.
We now come to the main section of this paper, which are the results of training RNN’s on the parity function. We used RNNs with LSTM cells, which have an internal memory vector. This intuitively makes RNNs more amenable to the parity problem, because they can store the current parity of the sequence, as the RNN traverses the sequence. Furthermore, since the RNN can take variable length sequences as input, this allows us to not only test the ability of the RNN to generalize to sequences of the same length, but to sequences of greater length as well. This will allow us to see if there are limitations to the number of timesteps that LSTM cells can remember input - for example, it will be interesting to see if, once the RNN has learned the parity function of sequences of length 15, whether it can generalize to sequences of length 30, 40, 50, etc - the extent to which it does so provides an indication of the ability of LSTM cells to perform the desired function while being immune to noise over time. This also corresponds to a sense of how precise the RNN’s "understanding" of the parity function is. These are all interesting questions about RNNs which we study in this paper, and that have not been studied to a significant degree in the existing literature.

3.1 Architecture and Hyperparameter Optimization

There are several architectural variables as hyperparameter optimizations that we tuned throughout the experiments. On the architectural level, the number of nodes per layer, as well as the number of layers, were the two major variables. We tried 100 and 200 nodes, and 1 or 2 layers, and found that 200 nodes of 2 layers worked best.

In addition to that was the existence of dropout, which we fixed at 50%. This did not improve training. In all cases, we used softmax followed by cross entropy as the cost function. Finally, we also varied the use of LSTM vs Gated Recurrent Unit (GRU) cells, and found that LSTM cells improved performance.

For hyperparameter tuning, we adjusted the learning rate, minibatch size, and number of epochs, and settled on a learning rate of 0.01, a minibatch size of 10, and we trained until convergence on the training set with a patience parameter of 8 epochs.
3.2 Testing Generalization

In order to test the generalization of the networks, we considered generalization to the testing data, which as noted before was constructed to be disjoint from the training data and hence a good measure of the ability of the model to generalize to sequences it had not yet seen. It is interesting to note that although almost all of the runs of the RNN resulted to close to perfect accuracy on the training set with enough epochs, and only some of them resulted in similar perfect accuracy on the testing set, indicating that some of the runs failed to generalize.

Below, I show to runs of the RNN, each starting with a different random initialization of the weights. I show the accuracy and loss per epoch for each run. One of the runs generalizes to the test set, and the other does not.

It is interesting to note that the only difference between these two runs were the random initialization of the weights. Both runs were trained on samples of length 15, a learning rate of 0.01, minibatch size of 10, and two layers of 200 LSTM neurons each. The cost was an end-of-sequence softmax followed by cross entropy, and the optimizer was Adam. These were found to be the optimal hyperparameters.
Figure 3.2: Successful RNN Epoch by Epoch Loss.

during our hyperparamter turning. This indicates that RNNs are very sensitive to the initial weights of the model. Over 13 runs of the RNN, 4 of them generalized to the test set, and 9 of them did not. This indicates that there is a "feasible region" in the weight space, in which the RNN generalizes. It would be interesting to learn properties of this feasible region, such as whether it is convex, or connected, and this is intended to be studied in future work.

Furthermore, it is interesting to see the speed at which the RNN goes from approximately 0.5 accuracy, close to random guessing, to perfect or close to perfect accuracy. The jump happens, quickly, within less than 20 epochs. The jump seems to indicate that the RNN slips into the solution very quickly once it gets close enough. When we compare Figure 3.1 and 3.3, we see that the rise in the training set accuracy is much slower in the failed RNN run in Figure 3.3. An inspection of all runs seems to indicate that runs that fail increase in training accuracy quite slowly, while runs that succeed jump to high accuracy much quicker, once they are about to find the solution.

It is interesting to note that we also tried L2 regularization of the weights of the
LSTM, and this resulted in the models not converging. This indicates that training involves setting some of the weights to values that lead to a worse cost during regularization than without - since regularization penalizes changes to weights that increase the magnitude of the weights too much, this indicates that the RNN needs to increase some of its weights significantly at some point, perhaps to find the region of the weights that can solve the parity function. Adding regularization drastically slowed down the accuracy rate increase on the training set - several trails found that even after 250 epochs, the training accuracy was still less than 0.55, whereas in runs without regularization, successful runs had often solved the parity function, or at least overfit to the training set with almost perfect accuracy by then. Similar results were found with adding Dropout as well, indicating that neurons need to work and train in tandem in order to find the parity function.
Figure 3.4: Failed RNN Epoch by Epoch Loss.
Chapter 4

HIDDEN STATE ANALYSIS

We performed analysis of the hidden states of the network over the entire course of training to gain insight into how the RNN learns the parity function. In this section we present the results and visualization from this analysis.

4.1 Visualization of Hidden States

We show here some samples of hidden states of the RNN per epoch. We note that there are distinct patterns in the different nodes, which is surprising given that our intuition is that since the parity function is symmetric in all the input sequences, we would expect each node to be symmetric, especially since this is a fully connected RNN and hence each node is symmetric with respect to its topology within the network. Hence, one possible interpretation is that different nodes may be keeping track of the parity of different parts of the sequence. Further testing may be able to verify or invalidate this hypothesis. In Figures 4.1-4.4, we show the average hidden and cell state, across all epochs, first for the successful RNN run, and then for the failed RNN run. These figures were generated by taking the average value over all states in the network at each epoch, and plotting the results as a function of the epoch number. It is important to note that the run of the RNN corresponding to Figures 4.1-4.2 corresponds to the same run that is graphed in Figures 3.1-3.2, the successful learning of the parity function, and that the run of the RNN corresponding to Figures 4.3-4.4 corresponds to the same run that is graphed in Figures 3.3-3.4, the failed learning of the parity function.

Looking at Figure 4.1, we see that the average is relatively constant, with a slight upward trend, until around epoch 175, which is exactly when the network’s training and test accuracy jump to around perfect. After that, the average cell state varies widely, which is quite interesting.

Figure 4.2 is even more interesting. The average is amazingly, extremely constant over roughly the first 150 epochs. From epoch 150 to 175, it starts to exhibit more variance, until epoch 175, when it starts to exhibit extreme variance. This seems to indicate that some sort of large, possibly random, shift in the hidden values of some subset of the nodes in the network led to the sudden solving of the parity problem.
Further fluctuations led to refinements in the accuracy. It is interesting to note that in both of these graphs, there is a great deal of fluctuation in epochs 225-250, but if you look in Figure 3.1, both training and testing accuracy are roughly constant at perfect accuracy.

Finally, we come to Figures 4.3 and 4.4, the failed run of the RNN function. We can see that both of these figures are approximately constant for the first 100 epochs. Then, as the fluctuation increase at that point onwards, we can see that this corresponds to the point in the Figures 3.3 and 3.4 where the training and test values start to diverge for both accuracy and loss, just like in the successful run. The difference, however, is that the successful run learned the parity function immediately after the fluctuation began, whereas the failed run took roughly 120 epochs to reach close to perfect accuracy on the training set after the fluctuations began.

Looking at the additional runs that were performed, we note that the same result can be observed, indicating that this is a stable and consistent observation across all runs of the RNN.
Figure 4.2: Average hidden state value across epochs in successful RNN run.
Figure 4.3: Average cell state value across epochs in failed RNN run.
Figure 4.4: Average hidden state value across epochs in failed RNN run.
DISCUSSION AND FUTURE WORK

We have presented in this paper a demonstration of the failure of feed-forward neural networks to learn the parity function, by showing its failure to generalize to a test set. We have furthermore shown the success of an RNN to learn the parity function, and generalize to a test set that was disjoint from the training set.

In this analysis, we discussed interesting behavior during training that the RNN exhibited, which provide interesting insight into how the RNN is learning the parity function. First, we note the fact that the successful RNN runs jump to perfect accuracy in a small number of epochs. This indicates that there is critical region of the weights where, once the RNN weights reaches this region during training, the RNN can converge to the optimal solution very easily. A further analysis of the hidden states during training shows that many of them converge to 0 around the time that the RNN learns the parity function. This seems to be the case for almost all the nodes of the RNN. This would suggest that adding regularization to the weights may speed up training, or increase the likelihood that the RNN will solve the parity function rather than overfit to the training data. However, based on our results in Section 3.2, regularization does not help, indicating that the network requires both freedom to increase particular weights, as well as allow neurons to work together to solve the parity function.

We also have discussed some of the analysis of the differences in runs in which the RNN converged and ones in which it didn't. In runs that failed, the training accuracy improved slowly and steadily, rather than jumping quickly to perfect accuracy. The improvement was slow and steady even when the training accuracy eventually (over)fit to the training data with close to perfect accuracy. This raises the question whether it is possible to predict whether the RNN will succeed, or overfit and fail, based on the rate of increase in the training accuracy in the beginning stages of training.

Then, we performed further visualization and analysis of the hidden states of the RNN during training, epoch by epoch. In Figures 4.1-4.4, we also see that the average of the hidden states and the average of the cell states start off roughly constant and then start to fluctuate around the same time that the training accuracy
starts to increase. This behavior is very interesting, and implies that varying the hidden and cell states is necessary in some way to solve the parity problem. It is also interesting that the average of the cell states starts off as constant in all cases. It would be very interesting to see if this is also the case for when the RNN tries to solve other mathematical sequences, which is another future direction mentioned below. Furthermore, since the fluctuation in the averages occurs before the increase in the training is complete, it is an interesting possibility to consider whether the fluctuations can be used to predict whether the RNN run will succeed or fail, before the "jump" occurs.

Next, we looked at an example state to show the behavior of most of the states in the neural network. Specifically, most of the node values, for both cell state and hidden state, converged to around zero after the "divergence" point between the training and testing values. Such an example state is displayed in Figure 5.1. The reader can examine the behavior of other nodes, of which there are too many to display all in this paper, in the provided link to the code repository. It is interesting to consider whether this implies that the RNN is pruning away nodes that are not useful to
computing the parity function. It does this even without a penalty for regularization, which implies that this is necessary for solving the parity function, or at least for refining its solution after it has reached the jump point. Since most of the nodes are converging to around zero for epochs after the "divergence" point, it is interesting to see the increase in fluctuations in the average value after the divergence point. If the fluctuations in individual nodes were truly random, we would expect them to be averaged out to approximately zero. However, since the average of the nodes actually increases in fluctuation, this indicates that the nodes are moving together. It is important at this point to tie in our analysis of adding regularization to the RNN, which we found to greatly reduce the performance of the RNN. This corroborates our observations that large, possibly coordinated fluctuations in the weights of the network are necessary for solving the parity problem and refining the solution, because regularization severely penalizes such fluctuations in the weights.

There are several exciting directions for this work to take in the future. Further analysis of the feasible region of the initialization of weights in order for the RNN to learn the parity function would also be a fruitful future direction. It was also interesting to note that during training, there was a phase shift during which the training and test accuracy jumped from around 0.5 random guessing to almost perfect accuracy within about 10 epochs. It would be very interesting to see what is happening to the weights around that point in training. Finally, we would like to train the RNN on different mathematical sequences, such as arithmetic sequences, geometric sequences, Fibonacci sequences, and the digits of $\pi$. These further steps would provide even further insight into how RNNs are able to learn mathematical sequences, and hopefully provide more insight into RNNs themselves.
