Analysis of the Learning Process of a Recurrent Neural Network on the Last k-Bit Parity Function

Austin Wang
Adviser: Xiuyuan Cheng

May 4, 2017

1 Abstract

This study analyzes how simple recurrent neural networks (RNN) approach learning the last k-bit parity function. In the first experiment, the difficulty that an RNN has while learning different numbers of time dependencies is measured, and it is found that k is a potential lower bound for the number of hidden nodes necessary to quickly learn the function. In the second experiment, a principal component analysis on the hidden state vectors suggests that the RNN is memorizing sequences to achieve perfect test accuracy rather than truly learning the parity function.

2 Introduction

The past few years have seen a remarkable explosion of renewed interest in data science and machine learning, ignited particularly by advances in technology, increased computing power, and a larger supply of data than has ever existed before. The rapid adoption of data science techniques in industry and the versatility of the field in multiple areas have allowed it to achieve enormous practical success. However, the underlying theory explaining this success has been lagging behind.

One of the most popular models currently being used is the artificial neural network, a composition of linear and nonlinear functions meant to approximate functions, often on labeled datasets. A particular class of neural networks is the recurrent neural network (RNN), which was designed to handle sequential data. RNNs have been proven to be extremely powerful in natural language processing tasks such as speech recognition, image captioning, and essay generation. However, while the effectiveness of RNNs is evident by their strong performance, much research remains to be done on exactly how RNNs learn. The process of determining which pieces of past information to hold on to and how to represent this "memory" is a complicated one, but understanding it could provide great

insight into why exactly they work so well.

To tackle this problem, we look at the last k-bit parity function, which takes a binary string and returns 0 if the number of 1's in the last k elements is even and 1 if the number of 1's is odd. This function is easy to understand and easy to implement, but not easy to learn. Changing just one entry in the string completely reverses the output. Changing an even number of entries in the string produces no noticeable effect on the output even though the input string was greatly altered. The hope is that by measuring the difficulty that traditional RNNs face in learning such a function and analyzing their memory state vectors, we can learn something about how they approach sequential problem solving.

3 Background

In the following sections, we present a brief overview of the structure of feedforward and recurrent neural networks, and describe some of the issues that occur when training the latter.

3.1 Feedforward Neural Networks

A feedforward neural network, also known as a multilayer perceptron, is a composition of functions that takes an input vector $\mathbf{x}$ and attempts to reproduce the associated output $\mathbf{y}$. Typically, we have a data matrix $\mathbf{X}$, whose rows correspond to the number of the observation in the dataset and whose columns correspond to the features for that observation, i.e. the predictors. For this data matrix, we have an associated $\mathbf{y}$, whose i^{th} entry represents the response for the i^{th} observation (row) in $\mathbf{X}$. A neural network is designed to learn parameters such that the predicted values $\hat{\mathbf{y}}$ are as close as possible to the true responses $\mathbf{y}$.

In general, feedforward neural networks all have a similar structure consisting of layers, which in turn are made up of nodes. There are three different types of layers: an input layer, hidden layers, and an output layer. The input layer, as expected, is the place where the input vector is fed in. Each node in the input layer holds one of the feature values for the given observation, so the number of nodes is equal to the number of the observation's features. The input layer values are then fed into the hidden layer, which is made up of nodes that are interconnected with the input nodes. Each of these connections has an associated weight attached to it — these weights are the parameters that the neural network is designed to learn. Within a hidden node, the weighted sum of the connections is calculated and then some nonlinear activation function is applied to this sum (usually the rectified linear activation function, the sigmoid function, or hyperbolic tangent function). The result becomes one of the inputs in the next hidden layer (if there is one) or the output layer. Finally, in the output layer, one more weighted sum of the prior connections is taken and fed

through an activation function to yield the predictions. The number of nodes in the output layer, therefore, should be equal to the dimension of the response for the given input vector.

A diagram of a feedforward neural network with one hidden layer is shown below:

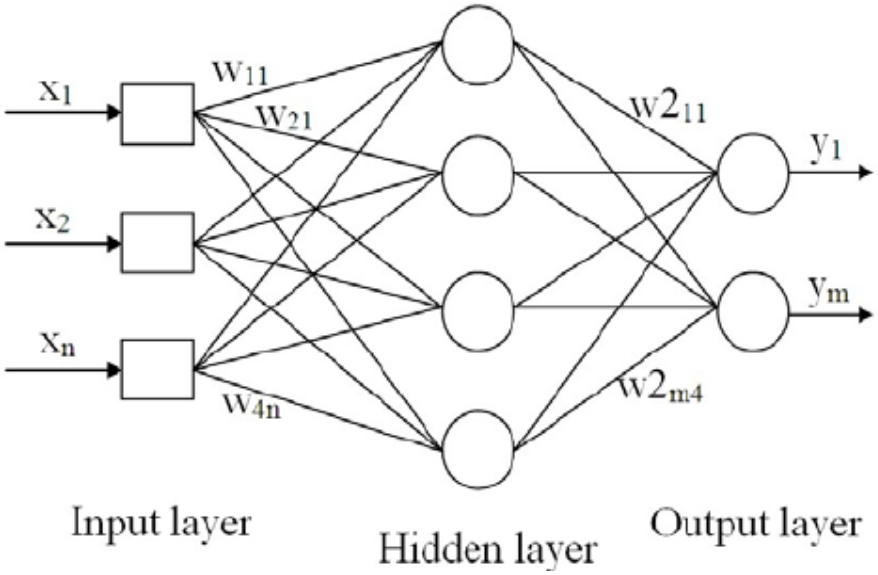

Figure 1: Feedforward neural network architecture (image reprinted from [1])

In summary, a neural network takes an input vector, applies some linear mapping using a learned weight matrix, feeds the result into an activation function, and uses the resulting vector as the input to the next layer until an output is produced. The weights are typically learned by initially setting them to be random, making a forward pass through the network, and updating them using gradient descent by backpropagating the error between the predicted value and the actual response for the given input.

Hornik, Stinchcombe, and White showed the power of this structure in 1989 by proving that feedforward neural networks with a single hidden layer could approximate continuous functions on compact subsets to any degree of accuracy [9]. However, the parity function is not continuous. While this does not mean that a feedforward neural network cannot approximate it, we focus on recurrent neural networks to learn the function, as the structure of recurrent neural networks and the idea of memory align more with how we typically view the parity function.

3.2 Recurrent Neural Networks

A recurrent neural network is a class of neural networks that was designed specifically to handle sequential or temporal data. The main difference between recurrent neural networks and feedforward neural networks is that a recurrent neural network "remembers" past information using a state vector. To understand this more clearly, we present the equations used when evaluating a network designed to perform classification of a sequence (or in our case, whether a sequence is "odd" or "even"):

$$s_t = tanh(W(x_t, s_{t-1}) + b_s)$$
$$o_t = softmax(Us_t + b_o)$$

where (x_t, s_{t-1}) is the concatenation of the vectors x_t and s_{t-1}.

The input is usually some sequence of numbers, where the t^{th} element is treated as the t^{th} time step in the sequence. s_t is the state vector at time t. As one can see from the equation, it depends on the input at time t and the previous state, which in turn depends on the previous input and the the state before that. In this way, the state vector acts as a sort of memory tracker; it provides us information about the network after only seeing part of the full input sequence. o_t is the output at time t. It is the transformation of the state into the desired prediction. In many situations, we only care about the final output, but this setup allows us to optionally track progress as we traverse the sequence.

Recurrent neural networks are often likened to folded feedforward neural networks. A diagram of the unfolded version is shown below:

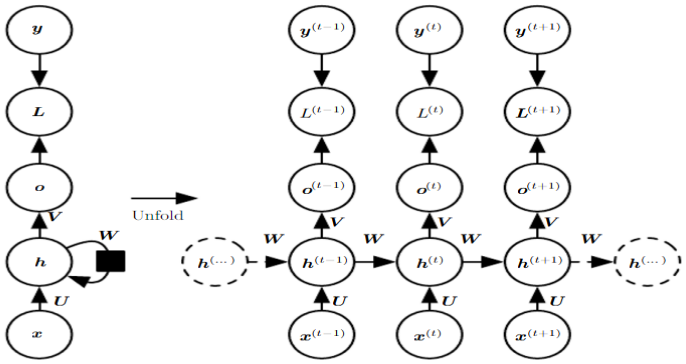

Figure 2: An unfolded RNN, which is very similar in looks to a feedforward neural network. The hidden state in this case is represented by h (image reprinted from [4])

We note that for an input sequence of length k, we simply have k layers in a

feedforward neural network. The only real difference is that instead of feeding in the input all at once at the beginning, we feed in parts of the input at each layer.

Particular RNN architectures have been shown to be able to simulate any Turing Machine [5] [11]. However, how easy it is to find the proper weights to do so is an entirely different matter. To understand the main issues with training a recurrent neural network such as the one presented above, we now look at backpropagating through time and the vanishing gradient problem.

3.3 Backpropagating Through Time and Vanishing Gradient

Recall from our previous equations for the state and output vectors of our RNN that we essentially have four sets of parameters to train: W, b_s, U, and b_u. For the sake of demonstrating backpropapating through time and vanishing gradient problem, we will only calculate the gradients with respect to W.

For a classification task such as learning the parity function, we use the cross entropy loss function given by:

$$L_t = -y_t log(\hat{y}_t)$$

Note that this is the loss at a single time step t. In particular, for a sequence of length n, we will really only care about L_n (in implementation, however, we often provide desired outputs at each time step; for the parity function, these outputs would be the running partial parity of the sequence, and thus we could have an associated loss for each time step). Let us look at the chain rule for calculating the derivative of L_n with respect to W:

$$\frac{\partial L_n}{\partial W} = \frac{\partial L_n}{\partial \hat{y}_n} \frac{\partial \hat{y}_n}{\partial s_n} \frac{\partial s_n}{\partial W}$$
$$\frac{\partial L_n}{\partial W} = \frac{\partial L_n}{\partial \hat{y}_n} \frac{\partial \hat{y}_n}{\partial s_n} \sum_{t=0}^{n} \frac{\partial s_n}{\partial s_t} \frac{\partial s_t}{\partial s_W}$$

where the second step uses the product rule, since s_n depends on the previous states which in turn depend on W.

We also note that:

$$\frac{\partial s_n}{\partial s_t} = \prod_{k=t+1}^{n} \frac{\partial s_k}{\partial s_{k-1}}$$

Normally, we would simply use these partial derivatives to calculate our Jacobian and perform some type of gradient descent algorithm to update the

weights in W. However, we see from the above equations that the derivative of the loss function with respect to W actually depends on all the previous states back to time step 0. Not only is calculating these gradients computationally intensive, but we also run into an issue called the vanishing gradient problem [3].

It is well known that the derivative of tanh is given by:

$$\frac{d}{dx}tanh(x) = 1 - tanh^2(x)$$

But the range of tanh is (-1,1), so the range of $\frac{d}{dx}tanh(x)$ is (0,1). Since $s_t = tanh(W(x_t, s_{t-1}) + b_s)$, this means that every time we take the derivative of s_t, we are multiplying by a number whose absolute value is less than 1. As we saw in our previous equations, we take this derivative many times, especially for the contribution of very early time steps. Thus, the derivative of the loss function with respect to W loses much of the contribution from early time steps due to the exponential shrinking of those derivatives. The result is that it is very difficult to train RNNs when they have to capture information toward the beginning of the sequence.

To solve this issue, various alterations of the RNN cell were proposed, the most effective being the LSTM and GRU cells which incorporate "forget" gates and a cell state to prevent many fractional gradients from being multiplied together [8]. LSTM and GRU RNNs have found much success in learning long-term dependencies. However, due to their slightly more complicated structure and increased number of parameters, we will be focusing on "vanilla" RNNs such as the ones previously described for sake of analyzing the learning process.

4 Experiments

With the basics of RNN architecture explained, we now describe our approaches to analyzing how RNNs learn. We performed two main explorations.

4.1 Exploration 1: Learning Capabilities of Minimally Trained Vanilla RNN

As explained earlier, the parity function is particularly difficult to learn. This arises from the fact that the output is dependent on the input values at *every* time step. Many sequential tasks that vanilla RNNs are often used for involve dependencies that are only a couple finite steps in the past. An example would be predicting the word "Spanish" when the input sequence is "Jane went to Spain to practice her". In this example, the key dependencies are on the words "Spain" and "practice" — the fourth and second to last words. The parity function, on the other hand, requires the RNN to learn dependencies on every input in the sequence. As expected, efforts to train a vanilla RNN for this task

were not particularly successful; even after hundreds of epochs and large hidden layer sizes, we were unable to break the 50% misclassification rate. While it is likely possible with meticulous training, the correct number of hidden nodes, the proper learning rate, and proper parameter initialization, we chose to devote our time toward exploring what types of related sequences an RNN could learn with minimal training.

4.1.1 Setup and Data Generation

A problem related to learning the parity function is learning the parity of the last k bits in an input sequence. This solves the issue of having to learn each and every dependency of input sequences with potentially different lengths. An interesting question to ask then is, "For a given k, how easily can a simple RNN with little training learn the function?"

Our input dataset is a random binary sequence of length 1000000. Our output vector is of the same length and contains the parity of the k bits up until that time step. Thus, if y is our output vector, we have that $y_i = parity(x_i, x_{i-1}, ...x_{i-k+1})$. We note that the first $k-1$ entries of y do not have any real meaning, but we defined them by allowing the input sequence to loop back around, i.e. $y_0 = parity(x_0, x_{999999}, ...x_{1000000-k+1})$. This is not a serious issue; we have plenty of correctly labeled data to work with.

```
def gen_data(k,size=1000000):
    assert size >= k
    X = np.array(np.random.choice(2, size=(size,)))
    Y = [0]*size
    for i in range(size):
        sumk = 0
        for j in range(k):
            sumk = sumk + X[i-j]
        if (sumk)%2 == 1:
            Y[i] = 1
        else:
            Y[i] = 0
    return X, np.array(Y)
```

Figure 3: The function used to generate our data.

```
#Example of what data looks like
a, b = gen_data(4,10)
print(a)
print(b)
```

```
[0 1 0 0 0 1 0 0 0 1]
[1 0 0 1 1 1 1 1 1 1]
```

Figure 4: An example of what our set of binary sequences looks like. a is our input sequence and b is the response. For this example, we chose a k value of 4.

This long sequence of data was then divided into stacked batches — we chose a batch size of 200 — and further divided into groups based on the number of

truncated backpropagation steps. The number of truncated backpropagation steps, as expected, determines how many steps back in time we backpropagate a given error. If we did not truncate, calculating all the gradients would be too computationally intensive and also likely ineffective due to the vanishing gradient problem. In addition, since the furthest we need to look back in time is k steps, as long as we choose the number of truncated steps to be greater than k, we will not run into any issues. For our experiments, we generally chose this number to be 20.

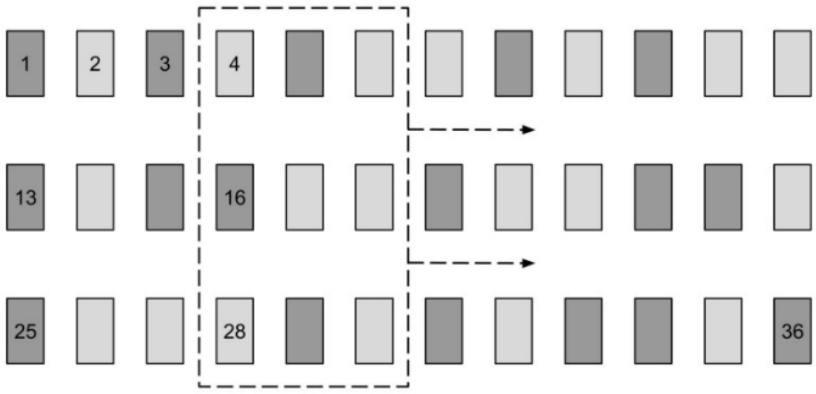

Figure 5: A diagram of how batches were set up. The length of the dotted box represents the batch size and the the width represents how many steps we allow the batch to traverse. In the diagram, both these numbers are 3, but in our implementation the box was 200x20 (image reprinted from [7]
)

An epoch was defined as a pass and weight update through all these batches. Unlike in many cases where one only has access to a single dataset, we can very easily generate new datasets of the same form. Thus, for every new epoch, we generated another 1000000-length binary sequence.

4.1.2 Building and Training the Model

To build our recurrent neural network, we used TensorFlow graphs. We converted the 0's and 1's in our input and output data to their one-hot encoded vectors, i.e. 0 was mapped to [1 0] and 1 to [0 1]. This was done so we could treat our prediction task as a classification problem. As a result, we used the cross entropy loss function. Recall from before that the cross entropy loss function is given by:

$$L_t = -y_t log(\hat{y}_t)$$

Since y_t is either [1 0] or [0 1] and $\hat{y}_t$ is a 2-dimensional vector as well of probabilities that sum to 1, the loss at any given time step is simply the negative log of our probability prediction for the correct label.

For each value of k, we minimized this function using the Adagrad optimization algorithm (with a base learning rate of 0.1, but this is of little significance since Adagrad adapts the learning rate for each parameter). Weights were randomly initialized according to a standard normal distribution, and we trained for a fixed number of 10 epochs. In addition, we did not worry about having some type of dropout or overfitting countermeasure — if the RNN successfully learned the function, it would be able to perfectly fit the training data. Thus, the number of hidden nodes was the only parameter we altered when testing how easily functions for different k values could be learned.

After creating the proper TensorFlow placeholders and defining the necessary relationships, our final train network algorithm is as follows (code adapted from [2]):

```
def train_network(num_epochs, num_steps, state_size, k, verbose=True):
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        training_losses = []
        for idx, epoch in enumerate(gen_epochs(num_epochs, num_steps,k)):
            training_loss = 0
            training_state = np.zeros((batch_size, state_size))
            if verbose:
                print("\nEPOCH", idx)
            for step, (X, Y) in enumerate(epoch):
                tr_losses, training_loss_, training_state, _ = \
                    sess.run([losses,
                              total_loss,
                              final_state,
                              train_step],
                                  feed_dict={x:X, y:Y, init_state:training_state})
                training_loss += training_loss_
                step_range = (size/batch_size)/num_steps
                if step % (step_range) == 0 and step > 0:
                    if verbose:
                        print("Average Loss at End of Epoch:",
                              training_loss/step_range)
                    training_losses.append(training_loss/step_range)
                    training_loss = 0
        Wstar = sess.run(W)
        bstar = sess.run(b)
        Ustar = sess.run(U)
        b2star = sess.run(b2)

    return training_losses, Wstar, bstar, Ustar, b2star
```

Figure 6: The TensorFlow function to train our network and perform gradient updates on our weights.

4.1.3 Results

The weight matrices and biases were saved from the training process so that we could predict on test sequences. We tested to see the minimum number of hidden nodes necessary for the RNN described above to successfully learn the

last k-bit parity function. Our test set was 1000 sequences of length 2, 3, ... , 10, 100, and 1000. A success was defined as having the correct prediction on the final k bits, and we say that the RNN learned it perfectly if it could achieve 100% accuracy on each of these different lengths. Predictions were found by choosing the value (0 or 1) with the highest probability, as our predictions were softmax values.

In training, we found that for a set number of hidden nodes, sometimes the RNN could learn the function better than other times. The only clear explanation for this is that the weight initializations play a significant role in whether or not the RNN can learn the function. The following hidden node results mean that we were able to find a weight initialization with the given number of nodes that achieved perfect accuracy on our test sets (not that *every* weight initialization with the given number of nodes achieved perfect accuracy).

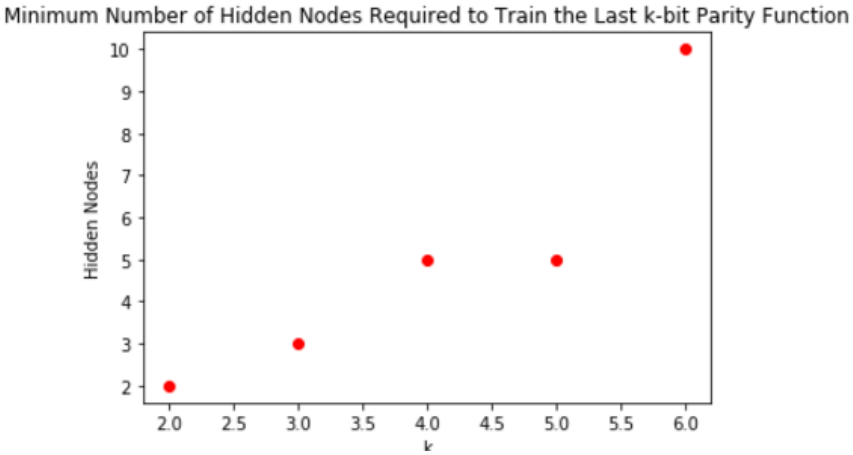

Figure 7: The minimum number of hidden nodes required for us to achieve perfect accuracy on all our training sequences. We note that these are conservative estimates — it is likely possible to achieve the same accuracy with fewer hidden nodes if trained for a greater number of epochs and with perfect proper weight initializations.

For k values in the range [2,6], we were able to successfully train our RNN. A key observation we made was that we always needed at least k hidden nodes to train the last k-bit parity function. Even with several different weight matrix initializations, the lower bound on the number of hidden nodes seemed to be k. However, in our actual experiments, it often took greater than k hidden nodes to train the function.

We failed to find a hidden layer size that could train the 7-bit parity function and beyond. As described before, this is likely due to the vanishing gradient

problem. Capturing longer and longer term dependencies becomes exponentially difficult. In order to successfully train the 7-bit parity function, it is likely that we would need to train for more epochs and choose a smarter weight initialization.

Below are the graphs depicting the training process for each of the $k \in [2, 6]$. In general, the training loss decreases after the first epoch — one pass through the data is enough for the RNN to begin to recognize the pattern. However, in the case of $k = 6$, we find that it takes a couple of epochs for the RNN to start learning. This seems to be the case as k grows larger and larger.

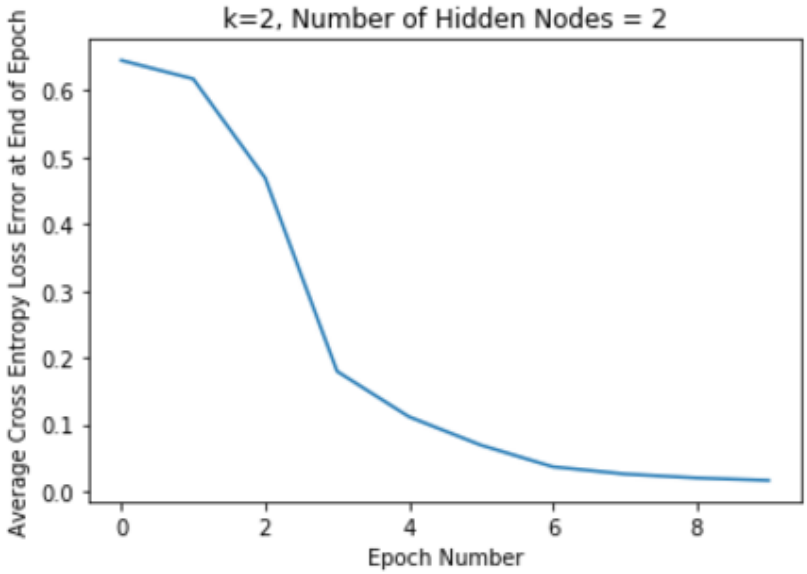

Figure 8:

It is interesting to note that with exactly k hidden nodes, there were some weight initializations that *nearly* reached 100% accuracy in 10 epochs of training (see Figure 13). From a look at the training error across epochs, it appears that had we trained for a couple more epochs, we may have been able to achieve 100% accuracy on all the different sequence lengths. This further suggests that k nodes is the lower bound for learning the last k-bit parity function — using more than k hidden nodes just allows us to learn the function more quickly and is less sensitive to the choice of weight initialization.

The fact that we nearly achieved 100% accuracy but were just shy is also interesting. This is surprising because we would expect the RNN to either know the

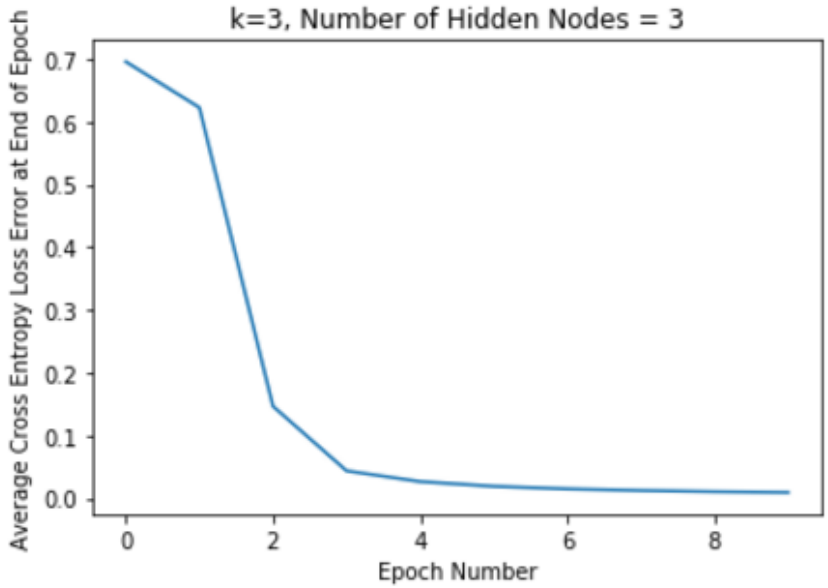

Figure 9:

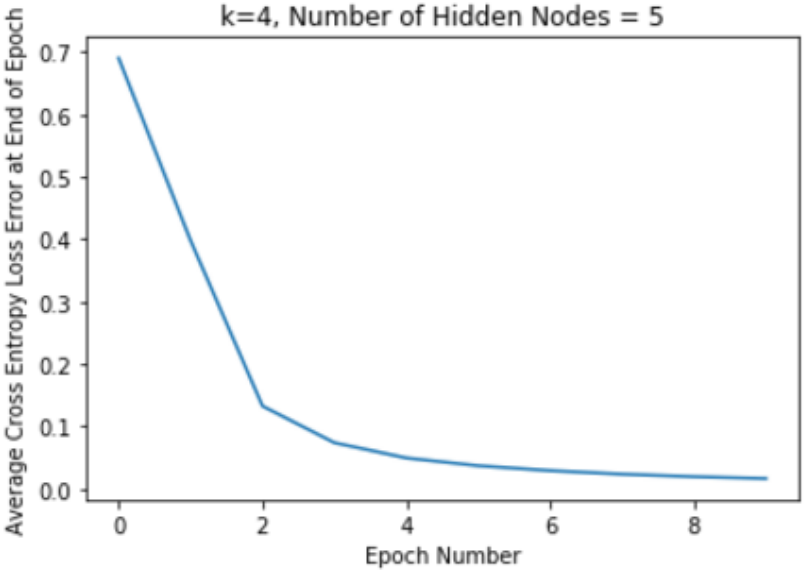

Figure 10:

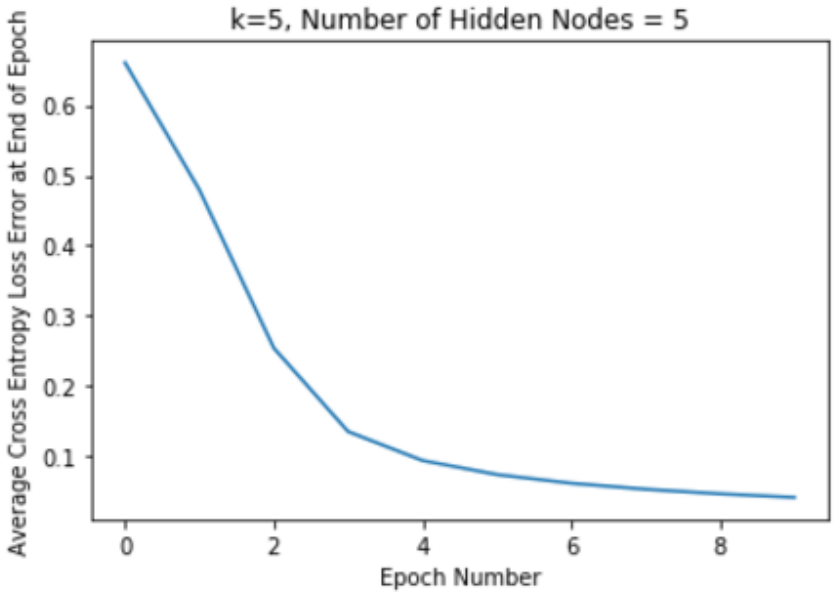

Figure 11:

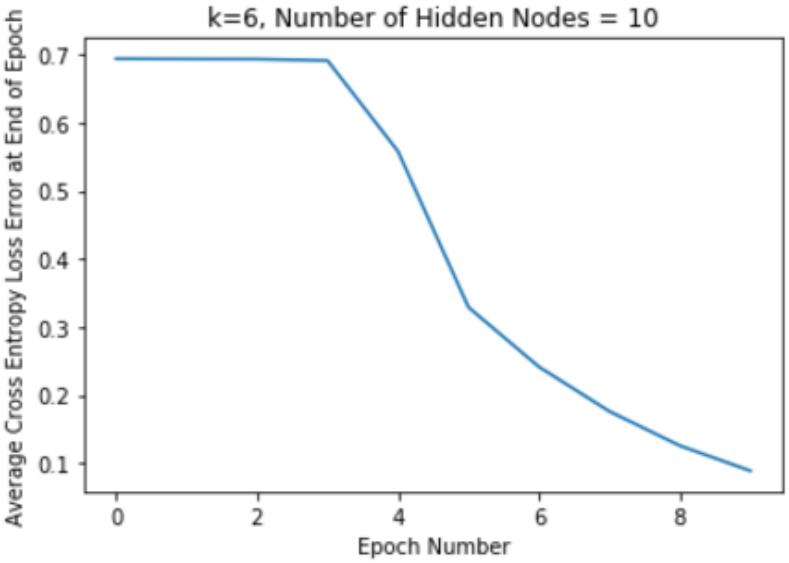

Figure 12:

```
Accuracy for Sequence of Length 4: 1.000000
Accuracy for Sequence of Length 5: 0.891000
Accuracy for Sequence of Length 6: 0.932000
Accuracy for Sequence of Length 7: 0.909000
Accuracy for Sequence of Length 8: 0.947000
Accuracy for Sequence of Length 9: 0.932000
Accuracy for Sequence of Length 10: 0.930000
Accuracy for Sequence of Length 100: 0.936000
Accuracy for Sequence of Length 1000: 0.942000
```

Figure 13: Accuracy for the last 4-bit parity function with 4 hidden nodes. For each sequence length, accuracies were calculated on 1000 different sequences with that length.

function or not; if it learned the proper function, we would have perfect accuracy, and if it did not, we would have 50% accuracy. An accuracy of around 99% indicates that it is learning a function that is *similar* to the last k-bit parity function, but not the actual one. This raises the question of whether or not even the RNNs that achieved 100% accuracy were actually learning the function we expected. Perhaps instead of a function of exactly k variables, the RNN learned one of more than k variables that happened to mimic the same output.

The above experiments helped quantify how difficult it is for vanilla RNNs to learn different types of parity functions. We learned that weight initialization is very important for convergence, but less important if the number of nodes in the hidden layer is large enough. Increasing this number generally sped up the learning process and was necessary as k increased. The size of the hidden layer is exactly equal to the dimension of our state vectors (our "memory"); this suggests that analyzing these vectors could be useful in understanding this learning process. This idea brings us to our second exploration.

4.2 Exploration 2: Analysis of State Vectors

One way we might expect the RNN to learn the last k-bit parity function would require that the state contain information about the current parity of the k previous bits and the oldest of those k bits. If that were the case, then it is clear how to calculate the next parity in the sequence from the previous state and next input: simply flip the previous parity if the next input is different from the oldest of the k bits. Determining if the state contained that information is a more difficult matter — as the state vector dimension (number of hidden nodes) could vary depending on how we trained our network, it is not clear which coordinates represent what and how to interpret the vectors. Regardless,

our first step was to track the hidden states in a test sequence.

4.2.1 Opening the RNN Memory

Having saved our trained parameters W, b_s, U, and b_u, we were able to take a test sequence of given length for the specified k value and output all of the hidden states at each time step using the tanh state equation described previously. As an example, we consider the following input-output sequence of the last 3-bit parity function:

```
Input Sequence:  [1, 1, 1, 0, 1, 0, 0, 1, 0, 1]
Output Sequence: [0, 1, 1, 0, 0, 1, 1, 1, 1, 0]
```

Figure 14: Example input-output sequence for last 3-bit parity function.

```
[-0.11715563, -0.9854357 ,  0.99644792]
[-0.99991188,  0.5137709 ,  0.99997688]
[-0.97145324, -0.99544927,  0.99999553]
[-0.99939609,  0.9965672 ,  0.19261538]
[ 0.90227473, -0.99996598,  0.99978456]
[-0.9999779 ,  0.99999987, -0.92666429]
[ 0.9999805 , -0.99791173, -0.99971481]
[ 0.83122567, -0.90086769, -0.15694718]
[-0.86836227,  0.99991726, -0.99964757]
[ 0.99997664, -0.99999984,  0.92190722]
```

Figure 15: The 10 hidden state vectors for the input example. Moving down column i describes how coordinate i of the hidden state changes as we progress through the sequence.

We might expect that if the parity changes at time step t, then each of the state vector coordinates would also change significantly at time step t. This was not the case; from time step 3 to 4 (using 0 as our starting time), we see from our example output that the 3-bit parity remains 0. However, coordinate 0 of the hidden state moves from -0.999 to 0.902, coordinate 1 move from 0.997 to -0.999, and coordinate 2 moves from 0.193 to 0.999. From this, it seems unlikely that the state vector is holding information solely about the parity.

Because of the fact that the state vector dimensionality could vary depending on how we trained, we decided to perform a dimensionality reduction to $\mathbb{R}^2$ to attempt to visualize the differences between the state vectors. We did this by

retrieving the principal components from the singular value decomposition of the centered hidden state matrix. As the first k-1 steps are not properly labeled, we excluded them and only looked at the data after them. The following are the 2-dimensional principal components of the state vectors, along with a visual representation of them:

```
t = 2  [ 0.56845852 -1.51652572]
t = 3  [-1.27263362 -0.63042587]
t = 4  [ 1.5564931  -0.34593728]
t = 5  [-1.58748549  0.22875764]
t = 6  [ 1.04919608  1.2500561 ]
t = 7  [ 1.11727328  0.51158315]
t = 8  [-1.53859806  0.36702032]
t = 9  [ 1.58615968 -0.22510732]
```

Figure 16: The 2-dimensional principal components, from time step 2 to time step 9.

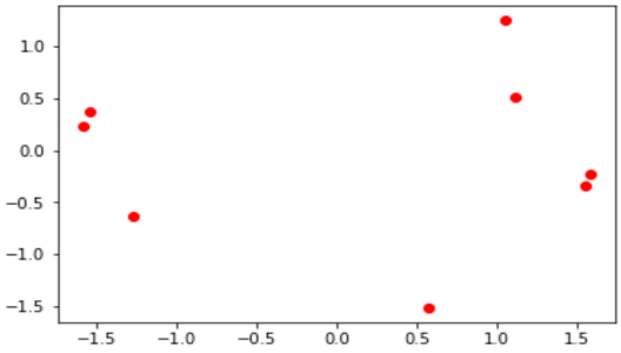

Figure 17: Coordinate 1 vs. Coordinate 0 of the hidden state principal components from time step 2 to time step 9.

We see now that some of the state vectors are quite similar — for example, the vectors at time step 4 and 9 and the vectors at 5 and 8. Let us now go back and analyze the input and output sequence to see if there are any similarities at those time steps.

Interestingly, we are easily able to observe the similarities between steps 4 and 9 and between steps 5 and 8; they share the same last k-bit input sequence. This is evidence that the state vector may not represent anything specifically about the parity, but rather it simply represents a specific past sequence.

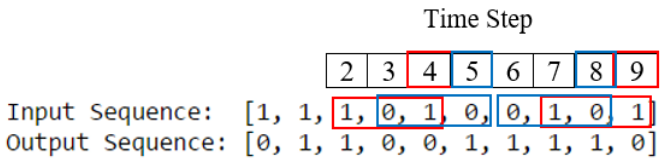

Figure 18: Similarities between Steps 4 and 9 and between Steps 5 and 8.

4.2.2 Results

To test if it is actually the case that the state vectors just represent the different permutations of the last k-bits in the input sequence, we extended our sequence to 1000 time steps for k=3. The following graph shows the different clusters:

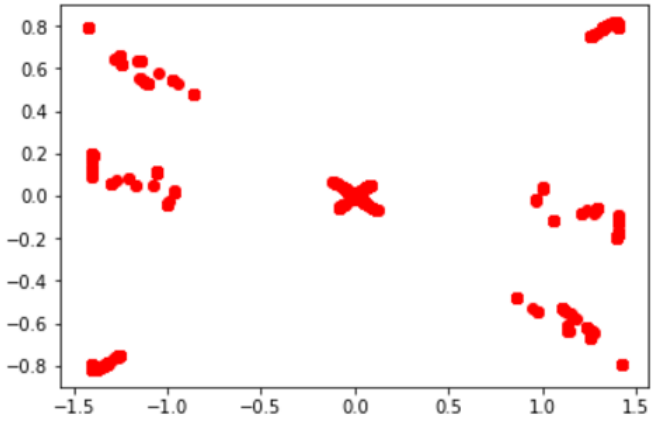

Figure 19: k = 3. Pictured are the two principal components for the hidden states over 1000 time steps.

As expected, we find 8 total clusters (6 on the edges and 2 overlapped in the middle) — 2^k clusters for the 2^k different permutations of the last k-bits. Thus we would expect 16 clusters if we change our k to 4 (see Figure 20).

From the graph, we see that changing k to 4 does lead to 16 distinct clusters of the hidden state principal components. Increasing k to 5 saw an increase in clusters as well, but the number became harder to count as the clusters naturally became closer to each other.

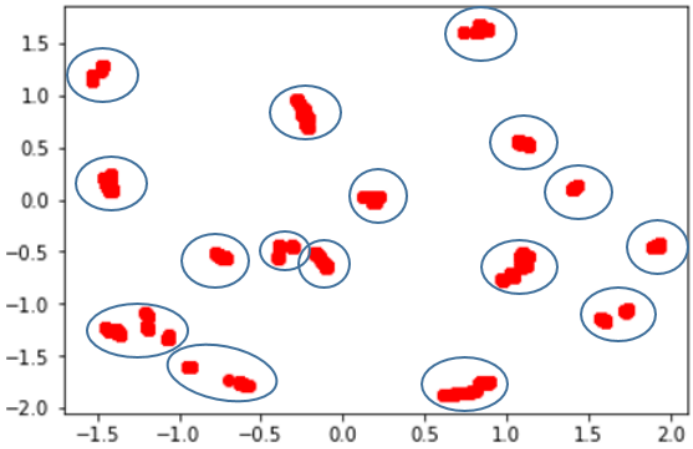

Figure 20: k = 4. Approximately 16 different clusters are visible as expected.

4.2.3 Discussion

From our principal component analysis of the hidden state vectors, we have found strong evidence to suggest that our RNN is not learning anything unique about how to calculate the parity, but is rather memorizing sequences and the associated outputs with them. The state for a past sequence of [1 0 1] is completely different than that for a past sequence of [0 1 1], even though they share the same parity.

It appears that the RNN does learn which inputs are most important — the number of clusters corresponded exactly as we expected for our k value. However, we do notice that our clusters are not always perfect; they tend to spread out a bit. This suggests that the spread within a cluster has to do with the inputs before the most important final k bits. As a matter of fact, for our last 3-bit parity function, when we analyzed the state whose previous inputs were [1 0 1 0], it was similar but slightly different from the state whose previous inputs were [0 0 1 0]. It is likely that our RNN is not truly learning the last k-bit parity function since it retains information prior to those k bits. Rather, it is learning a function of more than k inputs, but one that is more influenced by the last k.

This idea helps explain the strange observation we found earlier in which sometimes the RNN could achieve close to 100% accuracy, but not perfect accuracy. The function that it learned was likely one that still depended slightly on inputs before the final k inputs. With very specific sequences, probably with ones that were not as present in the training sets, these inputs could affect our probability calculations just enough so that after rounding, we would choose the incorrect prediction over the correct one. It also gives a potential explanation for why

increasing the state size made training easier; more hidden nodes mean it is easier to provide distinct representations for the various permutations of the last k bits. Indeed, had we looked at the 3-dimensional principal components instead of the 2-dimensional ones, it is likely that the clusters would be even more distinct. Increasing the dimensionality gives the RNN more freedom to find representations for the input sequences, making training quicker.

5 Conclusion

Our first exploration allowed us to see the vanishing gradient problem in action and help quantify the difficulty of training the last k-bit parity function. We found that we needed at least k hidden nodes to learn the function successfully, and that more nodes helped speed up training and made convergence less dependent on the weight initializations. We also found first signs that even an RNN that classified test sequences perfectly was not truly learning the last k-bit parity function in the way we expected, as sometimes we had accuracies of around 95% when we would expect either 100% or 50%. This was confirmed in our second exploration.

In our second exploration, we opened up the memory of the RNN by actually looking at the state vectors as we make a forward pass through a test sequence. A principal component analysis provided strong evidence for the hypothesis that our RNN was simply memorizing permutations of the last k bits. Our U weight matrix and b_u bias vector take this permutation and map it to the proper k-bit parity; however, the state does not represent the actual parity, just a sequence associated with that parity.

These observations are useful in attempting to generalize this to learning the full parity function on sequences of arbitrary length. Ideally, an RNN could do this by learning a state vector that represents the parity, and that only changes when the new input is a 1. From our experiments, however, this does not seem to be how it is approaching the problem at all. If it truly is just memorizing sequences and associated outputs, then it is not possible for it to learn the parity function for an arbitrary length sequence, as there will always exist a sequence not present in the training set. Therefore, in order to learn such a function, it is likely that we would need to provide some type of restrictions on the learning process so that memorization is not possible.

References

[1] Figure 5. a schematic diagram of a multi-layer perceptron (mlp) neural network., 2013. https://www.researchgate.net/figure/257071174_fig3_Figure-5-A-schematic-diagram-of-a-Multi-Layer-Perceptron-MLP.

[2] Recurrent neural networks in tensorflow i, 2016. http://r2rt.com/recurrent-neural-networks-in-tensorflow-i.html.

[3] Denny Britz. Recurrent neural networks tutorial, part 3 backpropagation through time and vanishing gradients. http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. 2014.

[6] Erik Hallstrom. How to build a recurrent neural network in tensorflow, 2016. https://medium.com/@erikhallstrm/hello-world-rnn-83cd7105b767.

[7] Erik Hallstrom. Schematic of the reshaped data-matrix, 2016. https://medium.com/@erikhallstrm/hello-world-rnn-83cd7105b767.

[8] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural Computation*, 2(5):1735–1780, 1997.

[9] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[10] Michael Nielsen. Neural networks and deep learning, 2015. http://neuralnetworksanddeeplearning.com/chap5.html.

[11] Hava Siegelmann and Eduardo Sontag. On the computational power of neural nets. 1992.