LEARNING NEURAL NETWORKS THAT CAN SORT

A Thesis
Presented to
The Academic Faculty

By

Arnab Dey

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science in the
College of Computing

Georgia Institute of Technology

Dec 2020

LEARNING NEURAL NETWORKS THAT CAN SORT

Thesis committee:

Dr. Richard Vuduc
College of Computing
Georgia Institute of Technology

Dr. Jeffrey Young
College of Computing
Georgia Institute of Technology

ACKNOWLEDGMENTS

I would like to express my appreciation to Professor Vuduc for introducing me to this research project. I have learned a lot from all our stimulating discussions, and I am truly grateful for all of the guidance. I would like to especially acknowledge Srinivas Eswar for mentoring me. Without all of his insightful advice and constant encouragement throughout the past couple of semesters, I could not have completed this work. I would also like to thank Dr. Jeffrey Young for providing valuable insight while I was working on this thesis. Lastly, I would like to convey my gratitude to my friends and family for the support and encouragement that I have received from the very beginning.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This thesis analyzes how neural networks can learn parallel sorting algorithms such as bitonic sorting networks. We discussed how neural networks perform at sorting when given no information or constraints about the allowable operations. We focused on analyzing how the architecture, training data, and length of the array impacted the neural network's performance at sorting. After encountering challenges with using neural networks to sort, we analyzed how neural networks learn the building blocks for sorting (comparator and swapping operators). Once we saw that these basic operations cannot be learned, we framed parallel sorting as a Reinforcement Learning problem. Using Reinforcement Learning, we were able to learn parallel sorting algorithms for sequences of lengths 4 and 8 under certain conditions, specifically limiting the allowable actions. We concluded that using Deep Reinforcement Learning there is potential to learn parallel sorting algorithms without any constraints.

CHAPTER 1
INTRODUCTION AND BACKGROUND

Sorting algorithms are an important topic in computer science that has many theoretical and practical implications. They impact ranges from users experience of applications to the types of problems that we can efficiently solve. There are many well-known sequential sorting algorithms such as Quick sort, Merge sort, and Bubble sort that are widely used, but we are particularly interested in learning new types of parallel sorting algorithms. Our interest in this class of algorithms stems from the runtime benefits of parallel sorting algorithms in comparison to sequential sorting algorithms as well as the similarities between structures for parallel sorting and neural networks.

Our plan to find these new types of algorithms is to use Deep Learning, which is most commonly used to recognize patterns in data - that are not apparent to humans - to solve problems. Before focusing on learning new algorithms, we first need to determine whether we can even learn parallel known algorithms such as bitonic sort. Once establishing whether we can learn these commonly known algorithms, the study plans to shift its focus to learning new parallel algorithms. Through this project, we hope that we show some of the benefits and problems with using Deep Learning methods.

1.1 Parallel Sorting

Computations can be carried out either in sequential order or in parallel. When a computation is done sequentially, the the computer can only complete one task or step at any time. In contrast, parallel computing allows us to work on multiple steps or parts of a computation simultaneously. For example, in a sequential implementation of Merge sort, we break an array (sequence of numbers) into smaller sub-arrays containing either 1 or 2 elements. We then sort each of these smaller sub-arrays and merge them together. By

recursively applying this process of merging the different smaller subsections, we eventually sort the array. With sequential merge sort, we can only work on merging two smaller sub-arrays at one time. However, there is also a parallel implementation of merge sort, where we can merge these different subsections in parallel, allowing us to sort an array even quicker.

We are specifically interested in sorting networks such as Bitonic sort (Figure 1.1), where a circuit-like architecture is used in order to sort an array. Bitonic sort was invented by Ken Batcher in the 1960s, and it involves constantly maintaining bitonic sequences and then merging these networks. A bitonic sequence means that the numbers are initially monotonically (consistently) increasing and then monotonically decreasing [1]. Bitonic sort is ideal for parallelism since each of the bitonic sequences can be maintained and merged independently. At each layer in the bitonic sorting network, we can do $\frac{n}{2}$, where n is the number of elements, compare and swap operations.

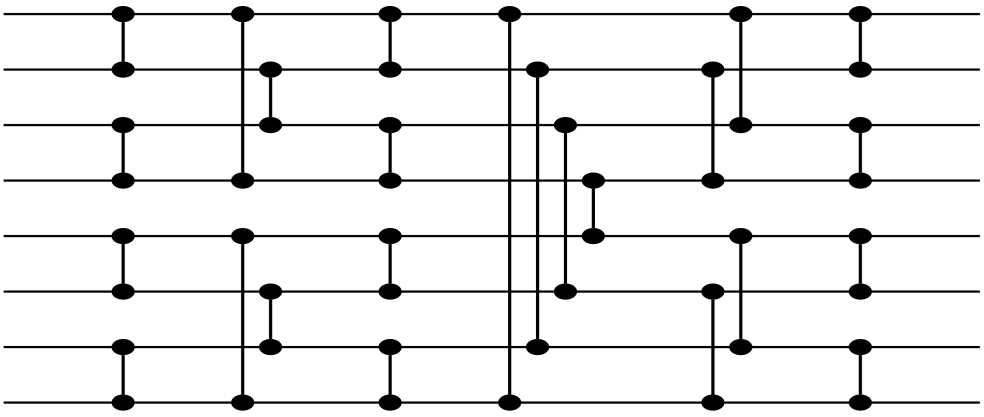


Figure 1.1: Bitonic Sorting Network
Image credit: [2]

While there are many parallel sorting algorithms, our focus is on the sorting networks since they have an architecture that is similar to that of neural networks. In parallel merge sort, for instance, the operations needed for merging the subsections are more complicated to represent: each of the merge operations cannot be represented by a defined sequence of

actions to follow. In certain cases, all of the elements in the one subsection precedes the elements in another subsection while the opposite may be true in other cases.

Example:

$$\underline{\text{Case 1}}:$$

$$\text{Left Subarray} = [1, 2, 3, 4], \text{Right Subarray} = [5, 6, 7, 8]$$

$$\text{Merged Result} = [1, 2, 3, 4, 5, 6, 7, 8]$$

$$\underline{\text{Case 2}}:$$

$$\text{Left Subarray} = [5, 6, 7, 8], \text{Right Subarray} = [1, 2, 3, 4]$$

$$\text{Merged Result} = [1, 2, 3, 4, 5, 6, 7, 8]$$

$$\underline{\text{Case 3}}:$$

$$\text{Left Subarray} = [1, 2, 7, 8], \text{Right Subarray} = [5, 3, 6, 4]$$

$$\text{Merged Result} = [1, 2, 3, 4, 5, 6, 7, 8]$$

The actions needed to replicate this type of merging would require an architecture different from that of neural networks and sorting networks. In comparison, bitonic sort involves a set sequence of elements that needs to be compared and swapped at each step.

1.2 Neural Networks

In general, neural networks (Figure 1.2) are known for finding patterns in data to determine how to transform a set of inputs into outputs. A neural network can be viewed as a functional representation for a mapping between a set of inputs to outputs. Neural network models have recently become extremely popular due to a rise in computing power as well as applications to computer vision and natural language processing. The high accuracy achieved from the use of convolutional neural networks to classify images [3] particularly was a major catalyst to the widespread use of deep learning. Despite these advancements,

researchers have made fewer applications of deep learning to more theoretical problems in computer science such as creating efficient sorting algorithms.

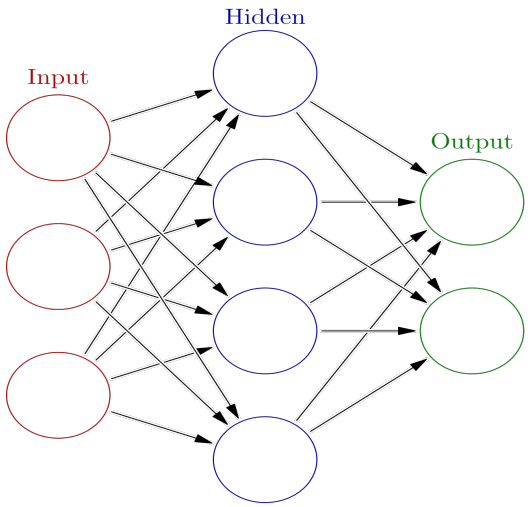


Figure 1.2: Neural Network
Image credit: [4]

The similarity between the architecture of neural networks and sorting networks led us to wonder whether neural networks can learn how to sort. In a fully connected neural network, each of the neurons in layer i is connected to the neurons in layer $i + 1$. This means that at each of the neurons in layer $i + 1$, we could treat the neurons at each of the layers as the elements at the j position, where j is the index of the neuron. We believe that each neuron could learn the compare and swap operation needed for bitonic sort as well as the elements to apply the operation to at each layer, allowing us to create the bitonic sorting network. Additionally, there is the possibility the neural network could learn different types of new sorting networks that were previously not known.

1.3 Reinforcement Learning

Another approach to finding parallel sorting algorithms is to focus on learning only how to apply the different operations needed for sorting. The set of valid operations can be

defined as all the ways we can select $\frac{n}{2}$ pairs of elements to compare and swap. We know that bitonic sort involves comparing and swapping, if needed, pairs of all the elements at each layer of the network. By defining the allowable operations, it becomes possible to use reinforcement learning to try finding the bitonic network or even new sorting networks. Reinforcement learning is used to determine the optimal set of actions an agent can take in order to maximize rewards in a specific environment. A very popular example is known as Gridworld (Figure 1.3), where an agent begins at a start state and needs to navigate around all the grids using the actions Up, Down, Left, and Right. The environment also contains two end, terminal, from where the agent can no longer take actions.

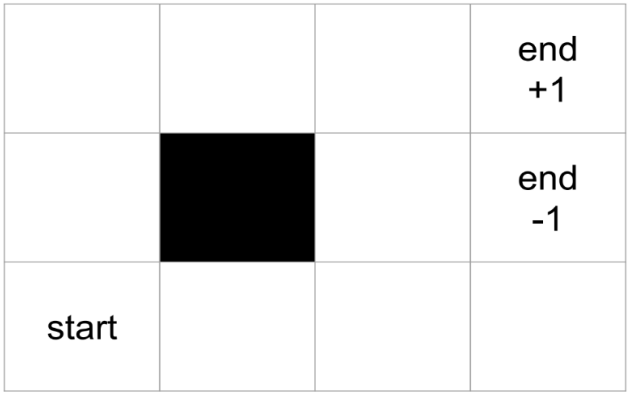


Figure 1.3: Gridworld
Gridworld is an example of environment, where we can use reinforcement learning to determine what actions (up, down, left, or right) an agent should take at any state.
Image credit: [5]

For learning sorting algorithms, we already know how to define the actions, but the bigger challenge is determining the environment, state space. This is because there are a number of different representations that can be used, and we need to choose the ideal state space for our objective: learning how to sort.

By framing the problem in this way, we no longer need to learn how to sort from scratch. This approach simplifies the problem, and makes it possible to focus only one aspect of learning how to sort. In comparison, the neural network approach involves learning

the operations and applying them all in the training process in order to sort.

1.4 Goal

The goal of the project is to explore how we can build neural networks that learn parallel sorting algorithms and to compare them against other known sorting algorithms. Bitonic Sorting Networks [1] are a popular method for sorting in parallel, and it has an architecture that is similar to that of a neural network. The bitonic sorting network contains $O(\log^2 n)$ layers where each layer contains $\frac{n}{2}$ compare and swap operations. If our neural network learns the operations done at each layer similar to a bitonic sorting network, then it would be significant as all the steps taken by each of the hidden layers in the neural network would be understood. Neural networks are often treated as just mathematical functions that have no real meaning: we know the inputs and the outputs of the neural network but cannot understand the reasoning used to determine the output. Therefore, it would be a new example where the neural network is self-explainable, and it would show that neural networks can learn efficient sorting algorithms. We also have the potential to learn even more efficient sorting algorithms or approximate sorting algorithms, which are approximately correct but are faster than known algorithms.

CHAPTER 2
LITERATURE REVIEW

2.1 Sorting Networks

Ken Batcher developed Bitonic Sort [1], an alternative to parallel implementations of algorithms such as Quick sort and Merge sort, that is very efficient in practice. Bitonic sort involves creating smaller bitonic sequences, which means that the sequence of numbers initially increases and then decreases. Sorting networks have comparators at each level of the circuit in order to maintain these bitonic sequences and has a time complexity of $O(n \log^2 n)$. The sorting network approach has been revolutionary as it takes advantage of the large amounts of compute power that are now ubiquitous.

There has also been research focused on determining the mathematical intuition behind Bitonic Sort [6]. Perl's paper focused on how the arrangement and values of the entries in a sequence impacts a bitonic merge network's ability to sort - the necessary conditions for bitonic merge to sort. Since we have not determined whether we can replicate the operations in bitonic sort using neural networks, this intuition will be useful when creating training and testing data for the neural network.

The runtime differences in sorting algorithms when used in large scale computations has been another area of active work. This type of research provided insight on the difference between the theory and applications of sorting algorithms. For example, Histogram and Radix sort, which are non-comparing sorting algorithms, have proven to be very efficient in practice [7]. This indicated that it might be worth exploring whether neural networks can replicate these non-comparing sorting algorithms that scale well.

2.2 Neural Networks

Neural networks were first discovered during the 1950s, and they were initially inspired by a biological perspective on human's neurological systems [8]. Early work such as the perceptron was considered to model the behavior of neurons, and perceptrons became the building blocks for neural networks. The biological point of view was considered significant because scientists believed that we should be able to create neural networks that learn similar to the brain. However as time passed, the influence of neuroscience on deep learning has started to decrease due to the lack of insight that we have about the human brains.

There have been numerous breakthroughs in neural networks since the 1950s that have propelled Deep Learning methods to the forefront. Using Stochastic Gradient Descent (SGD) to update the weights in neural networks has become a very popular optimization method [8]. SGD provides an efficient way by using batches of data to update weights and help fit a neural network to a set of data. In contrast, gradient descent requires the entire dataset to be used for updating weights, which becomes very computationally expensive if the dataset is too large. In the 1980s backpropagation was designed to calculate the gradients of the neurons efficiently [9]. SGD and backpropogation are two core operations that are heavily used today when training neural networks. More recently, there has been lots of focus on finding methods for training these networks such as random dropouts [10] – randomly ignoring the output of neurons to avoid overfitting. Overfitting often occurs when the model learns the training data very well, but it performs worse in new data that has not been seen before.

Loss functions that are used for backpropagation [11] have also been heavily researched since different loss functions cause the weights of the neurons to vary, which ultimately impacts accuracy. This research is important to the planned study because it provided some context for determining the ideal loss function. When comparing an unsorted array to its sorted state, there are many different loss functions that can be used. For instance, each

of the elements in the unsorted array's current position can be measured in relation to its position in the sorted array. Another option is to consider the sum of the squared difference for each of the elements in the unsorted and sorted array at all of the indices (positions in the array).

2.3 Sorting using Neural Networks

Research has been conducted using neural networks to aid in sorting, but solely relying on neural networks to sort has not been heavily researched. One approach involves using neural networks to partially sort an array, and then using sequential sorting algorithms such as Merge sort or Quick sort to "polish" [12] the neural network's results. This process is repeated multiple times, and it has shown to be effective when compared to C++ 's standard sorting function. The research from Zhu et al.'s paper [12] implies that sorting using neural networks is comparable to more traditional sorting algorithms from a performance standpoint. However, the paper does not conclude whether only a neural network can be used to sort an array.

There has also been some research done using Harmony Theory Artificial Neural Networks in order to sort numbers [13]. A unique aspect of this approach is that it uses "binary activations" [13], which means that the range of outputs from the neurons are not continuous. Tambouratzis' [13] paper provided a different approach to sorting, and it can be used as a comparison to the sorting method we find in the proposed study. The current study aimed to prove that neural networks can be used for sorting efficiently similar to both sequential and parallel sorting algorithms. This research planed on analyzing the computational time needed to sort arrays using neural networks along with methods to speed up the training time. The study was also interested in how the architecture of the neural network must change as the number of elements needed to be sorted changes. For instance, we wanted to see whether a neural network that can accurately sort has the same number of comparators as well as layers as a sorting network.

2.4 Reinforcement Learning

One of the early influences of reinforcement learning can be traced back to the 1950s, when mathematicians such as Richard Bellman were working on control theory [14]. Bellman set the foundation for using dynamic programming to solve a set of problems known as Markov Decision Problem (MDP) that are still used today. MDP's are used to describe problem where an agent needs to navigate through a state space, and the future state is only impacted by the current state - Gridworld (Figure 1.3) is an example of a MDP. There is also a factor of randomness in the actions selected by an agent that can be modeled using a MDP. Reinforcement learning was also getting shaped by studies in psychology since the 1930s. The work of psychologists such as Pavlov on the impact of different rewards and punishments (positive and negative reinforcement) were very significant. Pavlov notably studied dogs, and how their behavior/actions were influenced by previous rewards. Pavlov and other psychologists work inspired computer scientists to believe that computers can use a similar method (reinforcement) to learn.

Over the next 50 years, the field continued to evolve and in the 1980s, Watkins [15] designed an algorithm, Q-learning, that can be used when an agent does not have a full understanding of a state space or the rewards in an environment. Q-learning is especially relevant to our study since it was one of the methods used to attempt learning a sorting algorithm. In 2014, Watkins work on Q-learning was eventually applied to create a new method called Deep Q-learning, where the Q values (a value based on the expected reward for taking an action from a state) are approximated using neural networks to play Atari video games [16]. Deep Q-learning made it possible to solve even larger problems than Q-learning enabled since it scales better for large state spaces and actions. Deep Q-learning also served as an important potential method in our study. We believed that once we designed a Q-learning agent for sorting arrays of small length, we could use Deep Q-learning for sorting arrays of larger length.

2.5 Sorting using Reinforcement Learning

There has not been any research focused on using reinforcement learning to learn parallel sorting algorithms, but sequential sorting algorithms have been successfully learned [17]. Algorithms such as Quick sort, Merge sort, and Bubble sort have all been learned when the agent used is a Neural Turing machine, a computer. The research in the paper created a specific Instruction Set for the allowable operations such as moving variables and swapping two elements. We are not interested in learning parallel sorting algorithms using a Neural Turing machine view. Instead, we want our allowable actions to be restricted to only comparing and swapping multiple elements at one time like known sorting networks. However, the paper demonstrates that sequential sorting algorithms can be learned using reinforcement learning.

CHAPTER 3
NEURAL NETWORKS

3.1 Overview

We tried a variety of different experiments in order to determine whether and how we could train neural networks to create sorting algorithms. Initially, we were creating neural networks that we hoped would just learn all of the operations needed for sorting, and we attempted to sort arrays of varying lengths using different types of architectures, training data, and etc. As we conducted more experiments and analysis, we focused more on the building blocks needed for sorting, specifically comparing and swapping elements.

3.2 Existence of Solution

The first step involved determining whether it would be possible to mathematically sort an array using matrix multiplication/addition and activation functions such as rectified linear unit (ReLu). Therefore, we decided to manually construct the weight and bias matrices needed in order to sort arrays of length 2 and 3 - consisting solely of positive numbers - to prove whether a solution exists. Examining the dimensions of the weight and bias matrices also served as intuition behind the types of arrays that we would try to initially sort using neural networks.

3.2.1 Analysis of Sorting Array of Length 2

In order to sort an array of length 2, we need two operations (Figure 3.1): compare and swap. The compare operation will help determine the larger number while the swap operation will exchange the numbers at the two positions if they are not in ascending order.

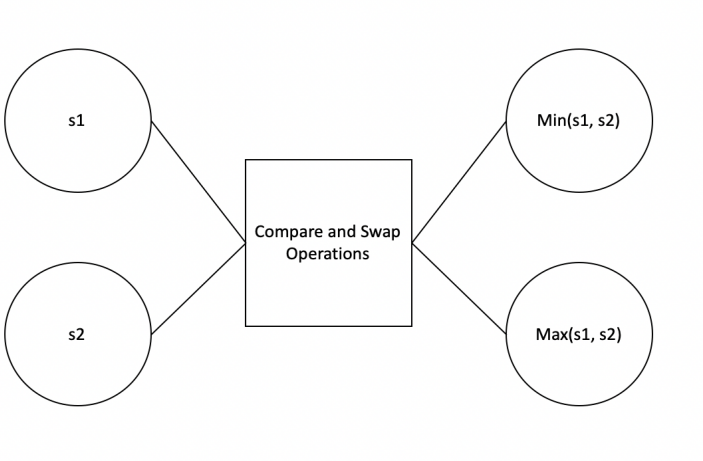


Figure 3.1: A single compare and swap operation is necessary to sort 2 numbers.

Matrix Representation

In this analysis, we created separate layers for swapping and comparing, but when using neural networks we tried to see if these operations could be done in one layer.

Comparing x_1 and x_2

$$A_1 = W_1^T x + b_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_1 - x_2 \\ -x_1 + x_2 \end{bmatrix}$$

$$Z_1 = \text{relu}(A_1) = \begin{bmatrix} \max(0, x_1) \\ \max(0, x_2) \\ \max(0, x_1 - x_2) \\ \max(0, -x_1 + x_2) \end{bmatrix}$$

Swapping x_1 and x_2, if needed

$$O = W_2^T Z_1 + b_2 = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} \max(0, x_1) \\ \max(0, x_2) \\ \max(0, x_1 - x_2) \\ \max(0, -x_1 + x_2) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$O = \begin{bmatrix} \max(0, x_1) - \max(0, x_1 - x_2) \\ \max(0, x_2) + \max(0, x_1 - x_2) \end{bmatrix}$$

Proof: We will show that for any two positive integers x_1 and x_2, the matrix representation above can be used to sort x_1 and x_2 in ascending order.

Case 1: $x_1 \leq x_2$

$$Output = \begin{bmatrix} \max(0, x_1) - \max(0, x_1 - x_2) \\ \max(0, x_2) + \max(0, x_1 - x_2) \end{bmatrix} = \begin{bmatrix} \max(0, x_1) - 0 \\ \max(0, x_2) + 0 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Since $x_1 \leq x_2$, we know that $\max(0, x_1 - x_2) = 0$. This means that $\max(0, x_1) - \max(0, x_1 - x_2) = \max(0, x_1) - 0 = x_1$ and $\max(0, x_2) + \max(0, x_1 - x_2) = \max(0, x_2) + 0 = x_2$ since x_1 and x_2 are positive.

Case 2: $x_1 > x_2$

$$Output = \begin{bmatrix} \max(0, x_1) - \max(0, x_1 - x_2) \\ \max(0, x_2) + \max(0, x_1 - x_2) \end{bmatrix} = \begin{bmatrix} \max(0, x_2) - (x_1 - x_2) \\ \max(0, x_1) + (x_1 - x_2) \end{bmatrix} = \begin{bmatrix} x_2 \\ x_1 \end{bmatrix}$$

Since $x_1 > x_2$, we know that $\max(0, x_1 - x_2) = x_1 - x_2$. This means that $\max(0, x_1) - \max(0, x_1 - x_2) = \max(0, x_1) - (x_1 - x_2) = x_2$ and $\max(0, x_2) + \max(0, x_1 - x_2) = \max(0, x_2) + (x_1 - x_2) = x_1$ since x_1 and x_2 are positive.

Therefore, the matrix representation described above can be used to sort two numbers.

We also did a similar representation for sorting arrays of length 3, and we noticed that many fewer parameters (total number of elements in all the weight and bias matrices) needed to be learned for sorting an array of length 2 when compared to an array of length 3: 20 parameters in contrast to 158 parameters. Since sorting an array of length 2 would require many fewer parameters, we decided to start with sorting an array of length 3.

3.3 Neural Network for Sorting an Array of Length 3

3.3.1 Methodology

The next step involved deciding the neural network architecture, and we choose to create a neural network with 2 hidden layers and 1 output layer (Figure 3.2). In this case, we expected the neural network to learn each of the compare and swap operations within one layer. This differed from the prior analysis where the compare and swap operations were done in separate layers.

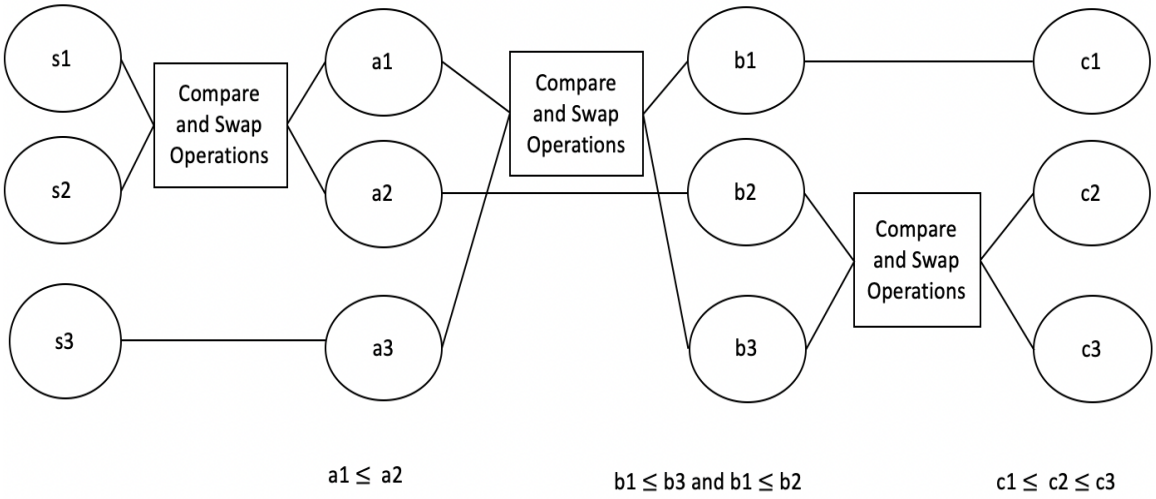


Figure 3.2: 3 different compare and swap operations need to be applied to sort 3 numbers. These operations cannot occur in parallel, and must be applied successively.

The neural networks were trained on scaled data, where each of the arrays were scaled independently. This caused the first and third numbers to be 0 and 1, respectively, while

the second number could be in the range of 0 - 1. The training data was also generated by sampling from a uniform distribution.

3.3.2 Results

When there were very few neurons on each hidden layer, the results were not very promising (Figure 3.3): the accuracy had a lot of variance for different trials. Therefore, we decided to increase number of neurons at each layer, which over-parametrized the problem.

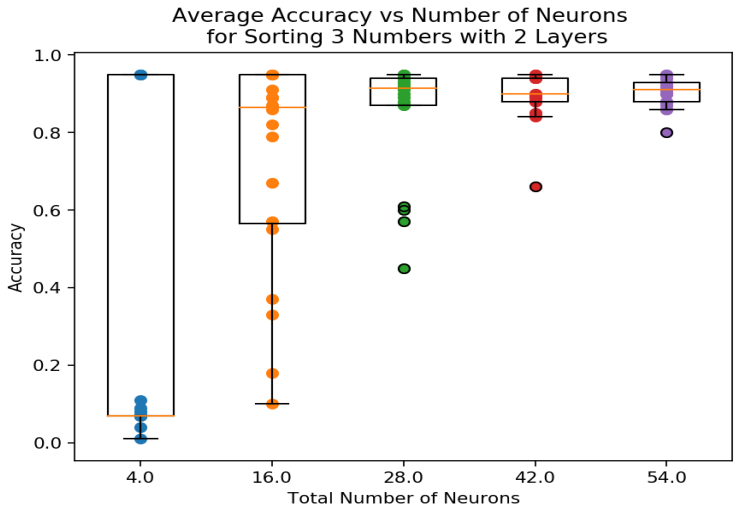


Figure 3.3: We ran 20 trials of each neural network configuration, where there were 2 hidden layers. In general, as the number of total neurons increased, the accuracy also increased. In the box plot, the dots represent all of the trials, and each color represents a different neural network.

Over-parametrizing the problem helped increase the accuracy of the neural network as nearly 100% accuracy was achieved.

3.4 Neural Network for Sorting an Array of Length 4

3.4.1 Methodology

Since we produced high accuracy for sorting arrays of length 3, we decide to conduct a similar experiment for sorting arrays of length 4. This helped determine whether

solely over-parametrizing the number of parameters, number of layers and neurons, would allow us to accurately sort arrays. We tried using three and four hidden layers (over-parameterization) with varying number of neurons at each layer to represent the compare and swap operations needed for sorting. We decided on this architecture of the neural network after analyzing the operations needed for sorting 4 numbers (Table 4.1).

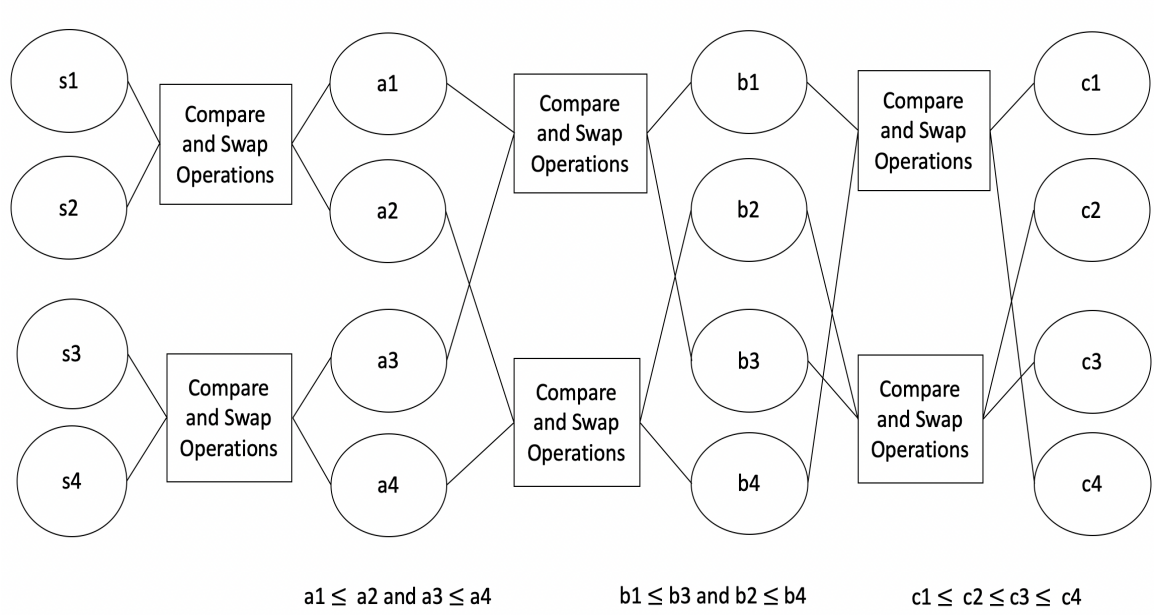
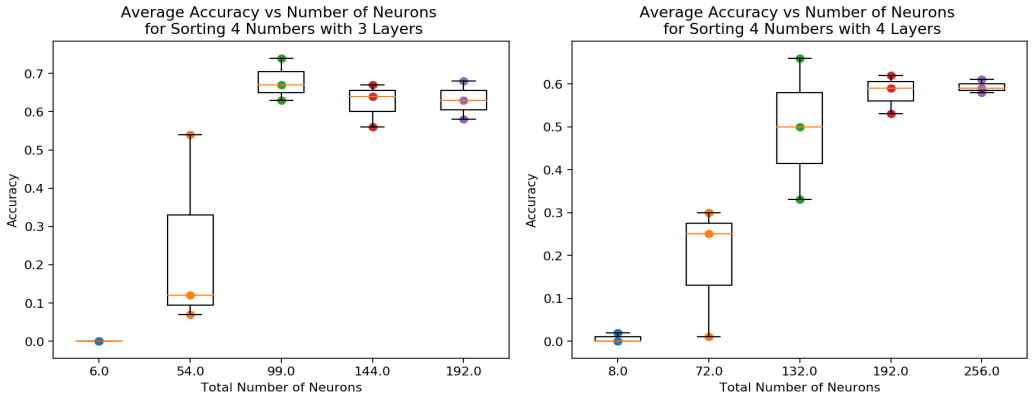


Figure 3.4: For sorting 4 numbers, a total of 6 compare and swap operations are necessary. This figure resembles the Bitonic Sorting Network for 4 numbers.

3.4.2 Results

The accuracy for sorting 4 numbers (Figure 3.5) was significantly lower than the accuracy for sorting 3 numbers, which caused the study to shift its focus to understanding the actions taken by the neural network.



(a) (b)

Figure 3.5: 20 trials of each neural network configuration, where there were 3 (plot a) and 4 (plot b) hidden layers, were ran. There was a general trend that the accuracy increased as the number of neurons increased, but when more than 99 neurons were used for the 3 hidden layer neuron network the accuracy decreased.

3.5 Evaluation of Models

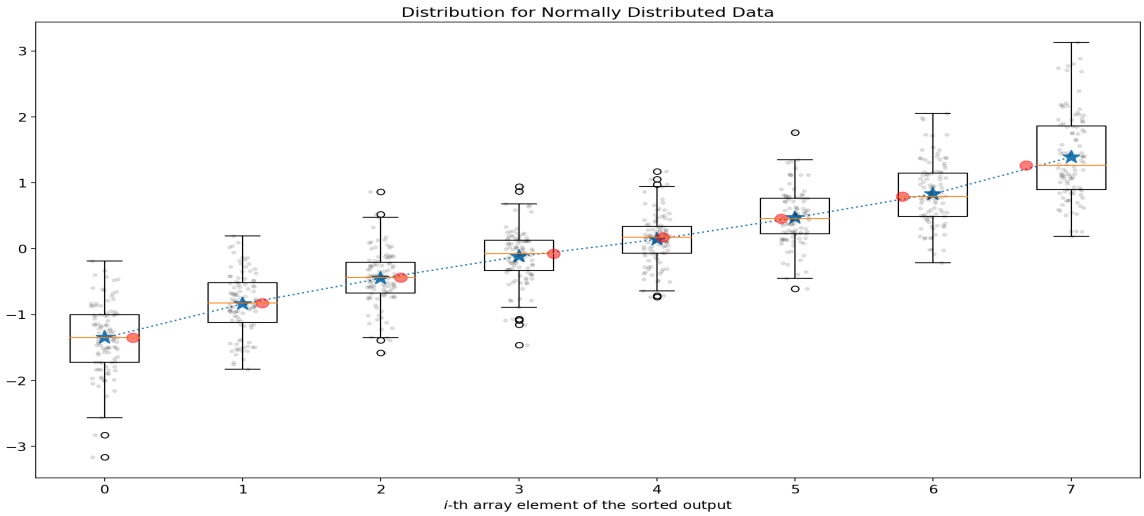
3.5.1 Discussion

We decided to re-examine the models for sorting 3 numbers since the neural network may not have been actually be sorting. After analyzing the weight matrices for sorting 3 numbers, we noticed that the neural network was zeroing the first and third terms in the output, and the bias term was setting the first and third values to 0 and 1 respectively. These values in the array were constant across all of the samples in the training data, so the neural network was manually setting them to 0 and 1 instead of learning how to sort. This led us to eliminate the bias term when retraining since it was preventing the neural network from learning how to sort. We also decided not to scale each of the arrays independently in order to ensure that the neural network was actually trying to learn the compare and swap operations needed for sorting. Each array in the training array is no longer scaled at once, so all three numbers would be between 0 and 1.

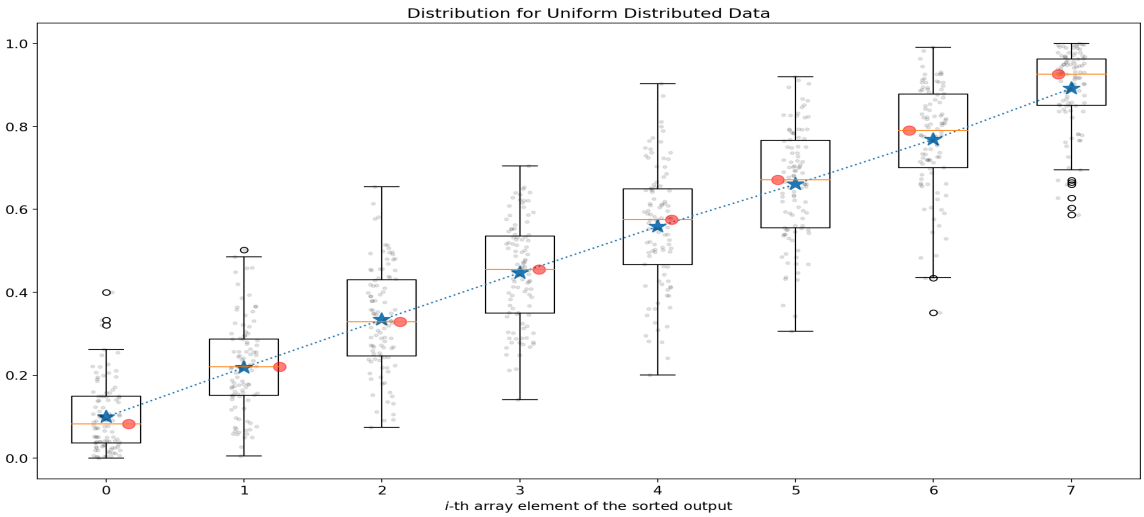
After eliminating the bias term and retraining on a different training dataset, we started to encounter poor performance for sorting 3 numbers, contrasting with our prior results

(Figure 3.3). This caused us to investigate the impact of the training data on results from the neural network. We noticed that the neural network was performing poorly in sorting arrays that were not from the uniform distribution used to generate the training dataset. We then decide to see the impact made by training and testing on data from different distributions (Figure 3.6).

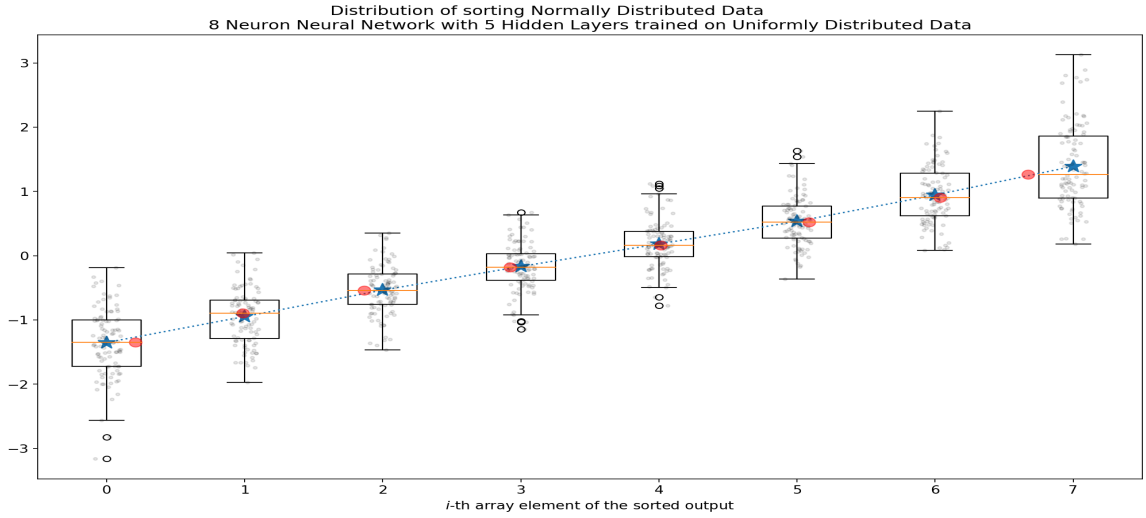
The blue star in the plots represent the mean value at each index in the array, and the red circle represents the median at each index in the array.



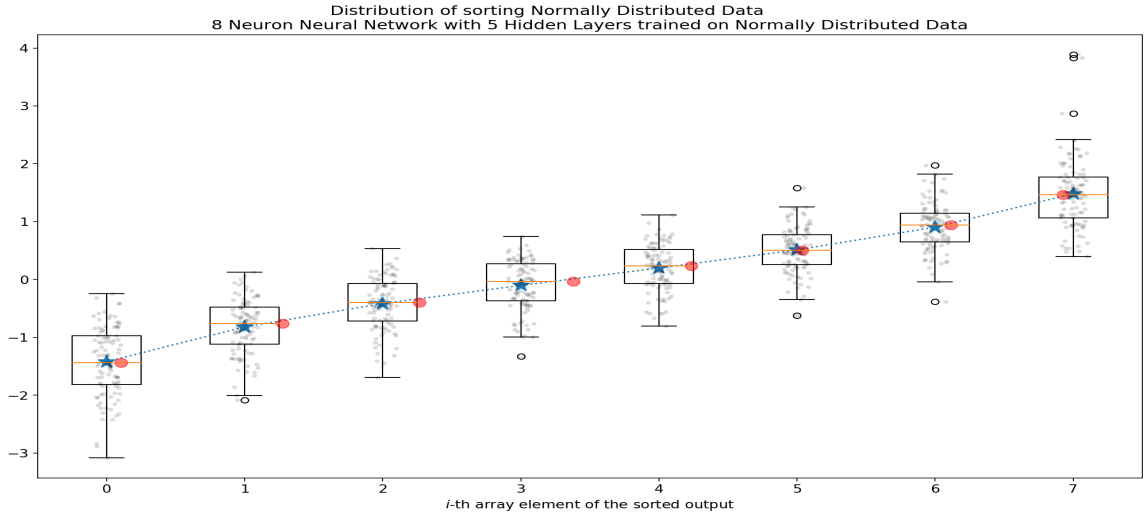
(a)



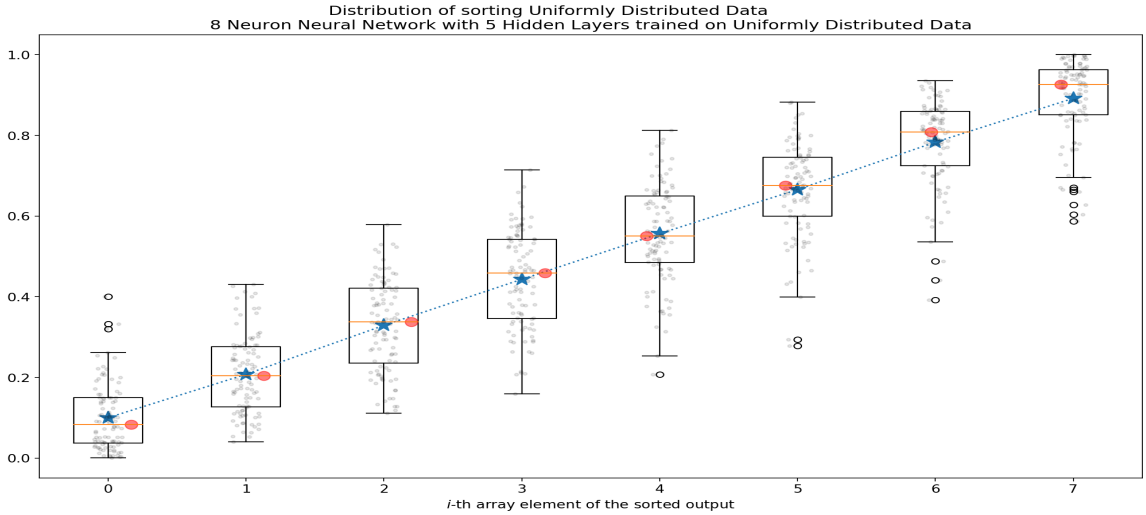
(b)


Distribution of sorting Normally Distributed Data
8 Neuron Neural Network with 5 Hidden Layers trained on Uniformly Distributed Data
3
2
1
0
−1
−2
−3
0
1
2
3
4
5
6
7
i-th array element of the sorted output

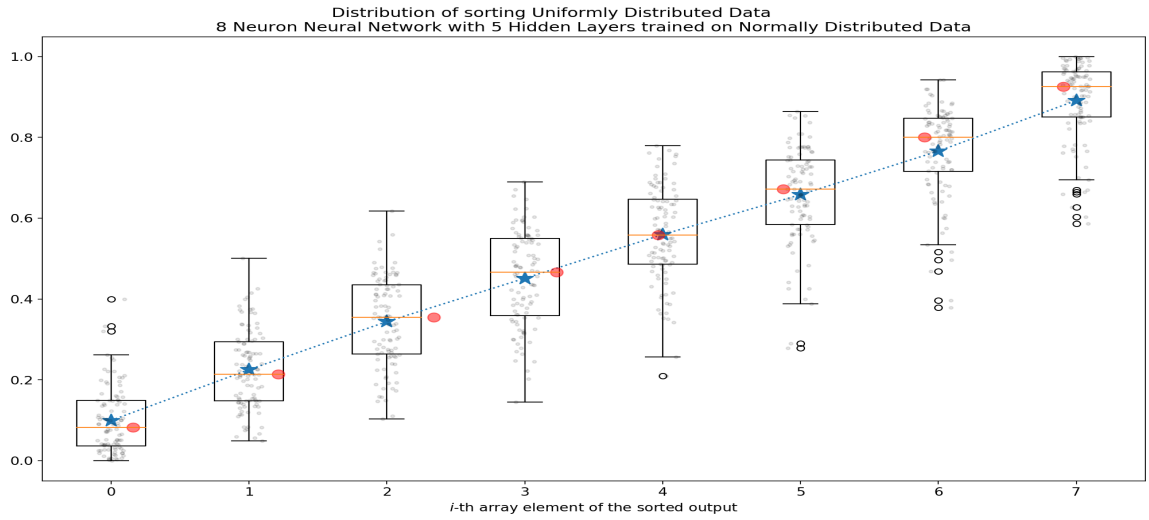
(c)


Distribution of sorting Normally Distributed Data
8 Neuron Neural Network with 5 Hidden Layers trained on Normally Distributed Data
4
3
2
1
0
−1
−2
−3
0
1
2
3
4
5
6
7
i-th array element of the sorted output

(d)


Distribution of sorting Uniformly Distributed Data
8 Neuron Neural Network with 5 Hidden Layers trained on Uniformly Distributed Data
1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4
5
6
7
i-th array element of the sorted output

(e)



(f)

Figure 3.6: Plot a and b shows the Normal and Uniform Distributions used for testing. Plot c and d shows the output of the neural network tested on normally distributed data. Plot e and f shows the output of the neural network tested on uniformly distributed data.

In general, the neural network was outputting results that resembled the distribution used to generate the training data. For instance, a neural network trained on arrays generated from normal distributions would have an output distribution containing characteristics of the training data. When the neural network was trained on arrays made out of multiple distributions, it was learning aspects from all of the different distributions used: the sorting was still not very accurate. This caused us to believe that the neural network was memorizing the distributions, so it was not generalizing well. These results were unexpected since we were not successfully learning how to sort; on the contrary, we were just mirroring different distributions.

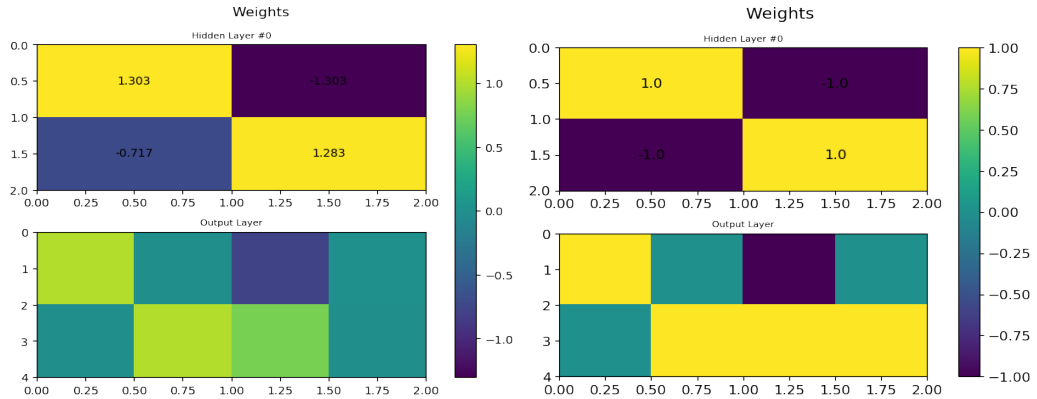
3.6 Sorting Array of Length 2

3.6.1 Methodology

Since we were unable to sort arrays of length 3 and 4 accurately, we then tried to train a neural network to sort 2 numbers with 100% accuracy. We knew that we could

mathematically represent how to sort 2 numbers using matrices, but wanted to determine whether we could learn the weights needed to sort.

3.6.2 Results



(a) (b)

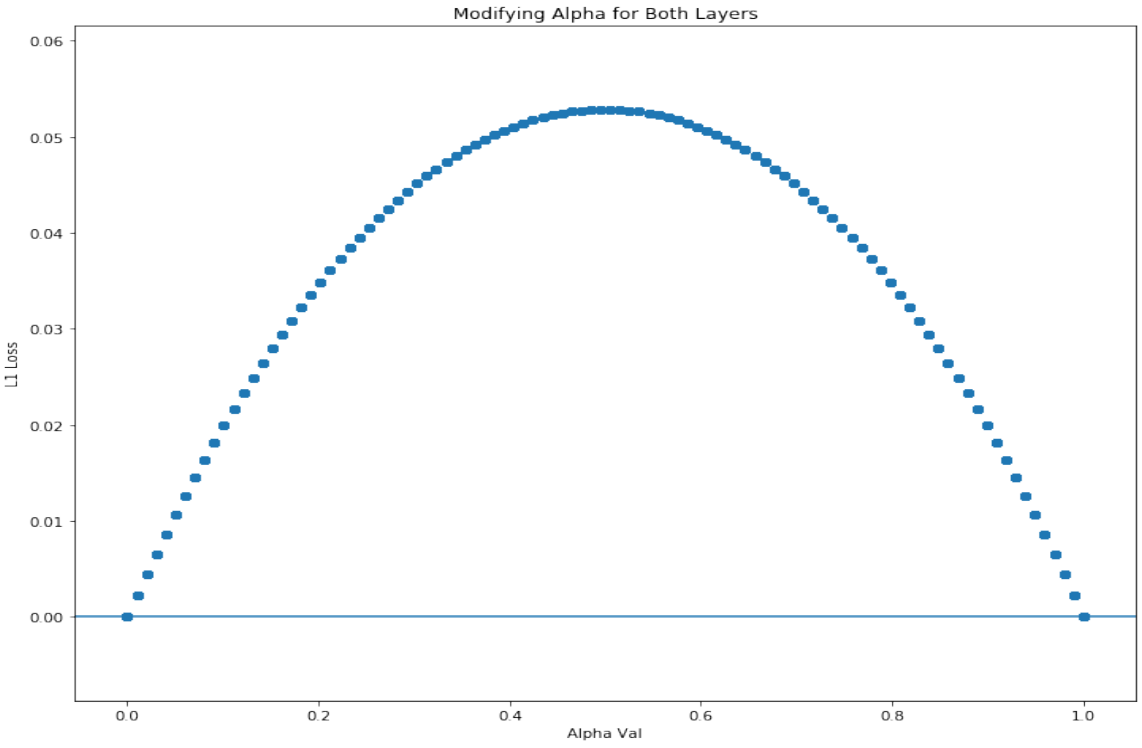
Figure 3.7: Plot (a) shows the actual weights and Plot (b) shows the expected weights. In Hidden Layer 0, each of the weights differ by ± 0.3, which means that the original values will not be able outputted by the neural network.

While training the neural network for sorting 2 numbers, we were never able to achieve the expected weights (Figure 3.7). The absolute difference in each of the elements in first hidden layer is approximately 0.3. This results in the original values of the array getting modified, causing the elements in the output of the neural network to vary from the inputs passed into the neural network. Since we are attempting to sort, we need the values to be preserved, so the weights cannot differ by such a large value.

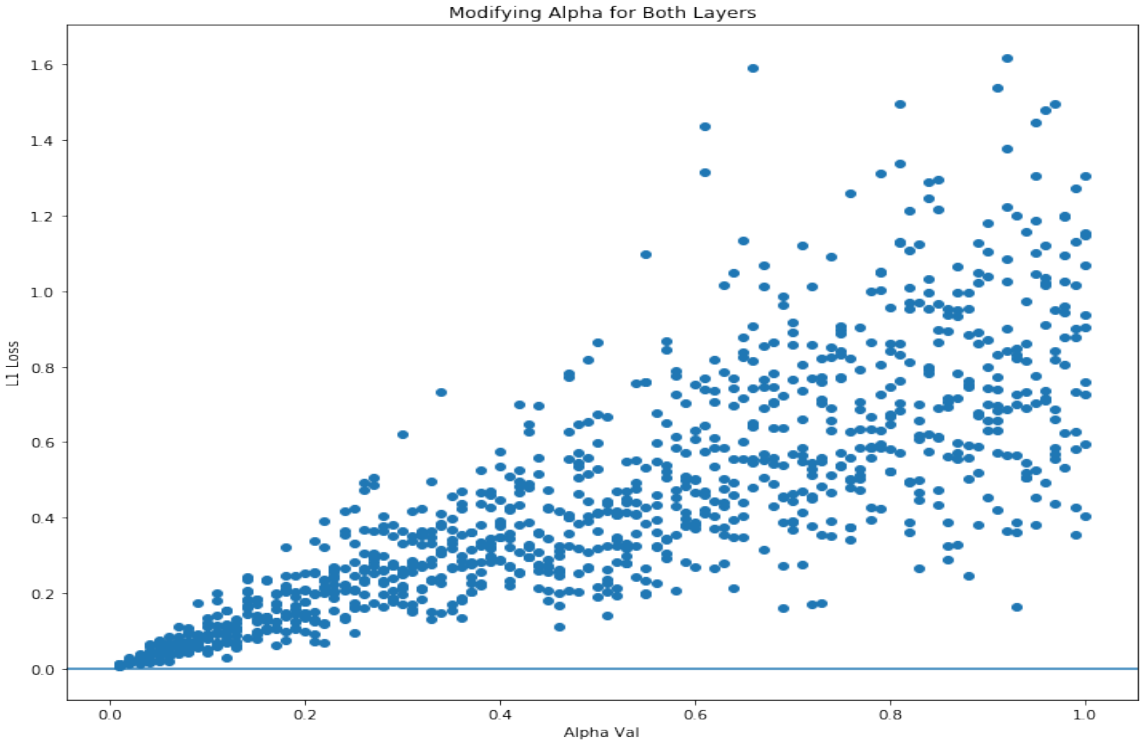
3.6.3 Discussion

The neural network's weights did not match the expected weights, so the loss function landscape was investigated as a result. It became evident that there are an infinite number of solutions, so the problem is not convex. Figure 3.8a shows how we can go from one solution to another solution by adding the same factor to all the weights. We also saw from plot b that it is very noisy, which means that the optimizer may get stuck during

the training process. Therefore, a different approach needed to be taken since the neural network was not able to create the compare and swap steps that were done by the weight matrices determined manually.



(a)



(b)

Figure 3.8: Alpha represents the factor added to all the weights in the neural network. In the plot a, we are able to go from one solution where the loss is 0 to another solution where the loss is 0. Plot b shows the the loss values if we add some noise, which contains a norm of alpha, to the expected weights.

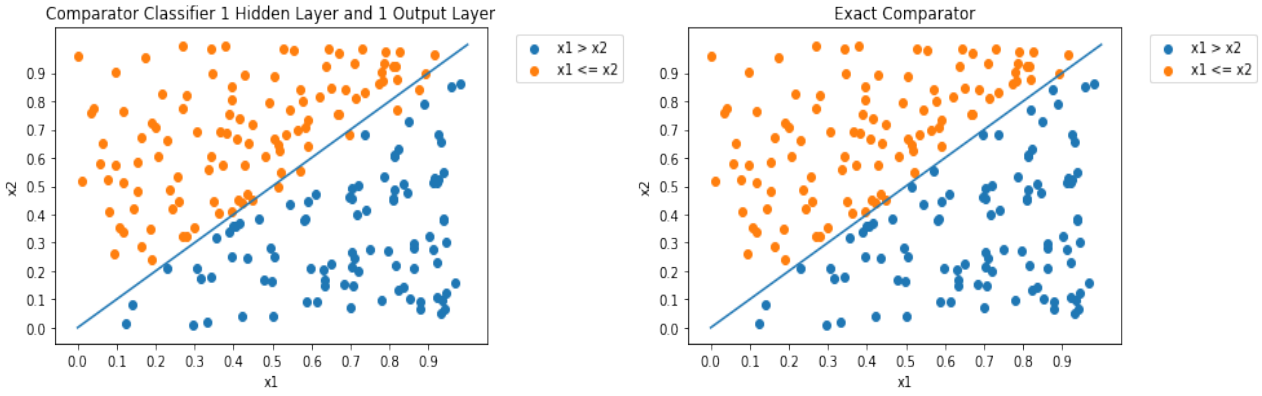
3.7 Learning Compare and Swap Operations

Since we were unable to sort 2 numbers accurately, it showed that the neural network is unable to learn how to compare and swap. Therefore, we decided that we would try to create a classifier for comparing and swapping.

3.7.1 Discussion

We learned through this experiment that the comparator classifier could not perform well for determining two numbers that are very close to each other, meaning points by the boundary of the line for $x_1 = x_2$. As a result, we can only learn an approximate

comparator using neural networks. In contrast, we could use an SVM [18] in order to accurately implement a comparator since SVM uses points in the data set as support vectors to determine the decision boundary. However, out goal is to use only neural networks for sorting, so we cannot use a comparator that is implemented by SVM. (Figure 3.9).



(a) (b)

Figure 3.9: The plot shows the Comparator Classifier (a) and SVM (b). The blue line represents the boundary separating where $x_1 > x_2$ and $x_1 \leq x_2$. SVM is an Exact Comparator since it accurately can always classify the larger number while the Comparator Classifier is an Approximate Classifier as some points are mislabeled on the plot.

Since the comparator is a fundamental building block needed, we realized that it would be impractical for the neural network to learn how to sort without having the ability to learn this operation. The swapping operator, which is used after a comparator, is even more challenging since we need to preserve the original values. The comparator can have a binary output to signify whether a number is larger than another number, but for swapping the two numbers actually need to be switched. For instance, if we have values a and b, which we would like to swap, at two neurons in layer i, then we want the two neurons in layer $i+1$ to have values b and a respectively. When swapping the numbers, a and b needs to be preserved, which is an even more challenging task than comparing as there is no clear way to swap using linear functions. Since we cannot learn a comparator or operator to swap, it means that we need to define these actions/operations. This means that we cannot

expect the neural network to learn these operations for comparators and swapping while trying to train a neural network to sort. This experiments provides some intuition behind why we encountered difficulty with training neural networks to sort thus far.

CHAPTER 4

REINFORCEMENT LEARNING

4.1 Overview

The problems with solely using neural networks to learn sorting algorithms led us to begin exploring using Reinforcement Learning, where the compare and swap operations are already defined as allowable operations. The major difference of using reinforcement learning was that we could merely focus on determining how the operation should be applied. By framing the problem in this manner, we eliminated the problem of preserving the initial values, which was a problem we faced earlier when we trained a neural network with no restrictions on the actions allowed. We primarily explored different types of state spaces in order to determine how to create sorting algorithms using Q-Learning.

For reinforcement learning, methods such as Q-learning [19] can be used to learn the optimal actions that can be taken from every state (called a policy). Q-values can be computed using the following formula, where s is the state, a is the action, r is the reward, α is the learning rate, and γ is the discount factor.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\Big(r_t + \gamma \max_{a'}(Q(s_t, a')) - Q(s_t, a_t)\Big)$$

We store all of the Q-Values for each pair of state and action - after Q-learning converges - in a table. However, if the number of actions and states gets too large, it becomes challenging and impractical to store all Q-values. In these cases, we can use a neural network to approximate the Q-Value. This makes it possible for us to learn sorting algorithms similar to the bitonic sorting network for large sequences of numbers, where the number of states and actions can become very large.

4.2 Index embedded State Space

4.2.1 Methodology

We decided to work on sorting arrays with length of 4 since bitonic sorting networks are for sorting arrays that contains a length which is a power of 2. For the reinforcement learning approach, we defined each action as a sequence of tuples (pairs) consisting of indices that would be compared and swapped (Table 4.1).

Action

Table 4.1: Actions for Sorting 4 Numbers

Action ID	Pair 1	Pair 2
0	(0, 1)	(2, 3)
1	(0, 2)	(1, 3)
2	(0, 3)	(1, 2)

For instance in action 0, we would compare and swap elements at indices 0 and 1 if the element at index 1 was less than the element at index 0. We would also compare and swap the elements at indices 2 and 3 as a part of action 0.

State

For the state space (Table 4.2), we decided to represent all permutations of the indices as the possible states. Initially, all arrays will start at the state (0, 1, 2, 3) since this represents the original ordering of the array, where the arrays indices are in ascending order.

Example:

$$\text{Original Array } = [10, 12, 13, 5], \text{State} = [0, 1, 2, 3]$$

Table 4.2: Index embedded State Space for Sorting 4 Numbers

Start = (0, 1, 2, 3)	(1, 0, 2, 3)	(2, 0, 1, 3)	(3, 0, 1, 2)
(0, 1, 3, 2)	(1, 0, 3, 2)	(2, 0, 3, 1)	(3, 0, 2, 1)
(0, 2, 1, 3)	(1, 2, 0, 3)	(2, 1, 0, 3)	(3, 1, 0, 2)
(0, 2, 3, 1)	(1, 2, 3, 0)	(2, 1, 3, 0)	(3, 1, 2, 0)
(0, 3, 1, 2)	(1, 3, 0, 2)	(2, 3, 0, 1)	(3, 2, 0, 1)
(0, 3, 2, 1)	(1, 3, 2, 0)	(2, 3, 1, 0)	(3, 2, 1, 0)

If we take action 0, we compare the elements at index (0, 1) and (2, 3).

Element at index 0 (10) is less than the element at index 1 (12), so we do not need to swap these elements.

Element at index 2 (13) is greater than the element at index 3 (5), so we need to swap the elements at index 2 (13) and 3 (5).

$$\text{Array after taking Action 0 } = [10, 12, 5, 13], \text{New State } = [0, 1, 3, 2]$$

This design allowed us to have a finite number of states since the indices can only be arranged in 24 different ways. However, there is no specific terminal state.

Reward

In this approach, we limited the number of actions allowed to three since know we can sort 4 numbers with three actions (Figure 3.4). We always begin at the start state (0, 1, 2, 3), but the terminal states will vary based on the actions taken. Since it is challenging to determine an appropriate reward function for intermediate states, we assigned a reward of 0 for all non-terminal states. For all terminal states (any state we reach after taking the maximum number of allowed actions or the array becomes sorted), we assigned a reward of 10 if the array is sorted. If the array is not sorted after the maximum number of actions, then we assign a reward of -10.

The goal of this approach was to determine the sequence of actions to maximize our reward. The reward function in our case correlated with whether the array of length 4 was sorted. For instance, if we learned the policy Action 0, Action 1, and Action 2 from the start

state, then we could maximize our reward using this sequence of actions. If we were trying to sort an array of length 2 using this approach, there was one possible action (Figure 3.1): compare and swap between elements at index 0 and 1. Therefore, we decided sorting arrays of length 4 would be appropriate since it is the next largest power of 2.

4.2.2 Results

After using Q-Learning, we get a Q-Table where the values do not converge. Therefore, the accuracy (Table 4.3) is low for both the training and testing examples.

Table 4.3: Accuracy for Index embedded State Space

Training Accuracy	Testing Accuracy
0.475	0.255

4.2.3 Discussion

With this approach, however, we encountered problems because a state could be a terminal state when training one array but a non-terminal state when training a different array. For instance, the state (0, 1, 3, 2) could be a terminal state in one case, but it may be a non-terminal state if the array is unsorted when the state is (0, 1, 3, 2). The lack of a specific terminal state was causing the values not to converge. Additionally, there was no concept of time embedded in this state space, meaning taking 2 actions could result in the same state as taking 1 action. In situations like Gridworld (Figure 1.3), it is useful to go back to a previous state in order to maximize rewards if we landed in a bad state. In our case, we were looking for a deterministic algorithm, so we did not want the ability to undo going into a bad state.

4.3 Action embedded State Space

In order to create a state space more suited for learning a sorting algorithm, we decided to make our states a function of our previous actions.

Table 4.4: Q-Values for Index embedded State Space

Actions			
State	[[(0, 1), (2, 3)]	[(0, 2), (1, 3)]	[(0, 3), (1, 2)]]
(0, 1, 2, 3)	60.746801	33.68534699	50.7669668
(0, 1, 3, 2)	31.05822791	32.11008799	58.72647791
(0, 2, 1, 3)	49.27464063	24.28412992	16.43935001
(0, 2, 3, 1)	-144.	-135.	-135.
(0, 3, 1, 2)	-279.	-279.	-270.
(0, 3, 2, 1)	47.18575482	41.45171411	23.94971968
(1, 0, 2, 3)	31.00934544	21.87231909	48.80343193
(1, 0, 3, 2)	50.89830381	12.19240531	8.61990081
(1, 2, 0, 3)	-225.	-216.	-216.
(1, 2, 3, 0)	-153.	-144.	-144.
(1, 3, 0, 2)	-198.	-189. -	189.
(1, 3, 2, 0)	-144.	-144.	-153.
(2, 0, 1, 3)	-117.	-108.	-108.
(2, 0, 3, 1)	-63.	-63.	-63.
(2, 1, 0, 3)	33.85440185	14.78013327	27.48249695
(2, 1, 3, 0)	-81.	-81.	-72.
(2, 3, 0, 1)	38.96311898	23.2234004	34.24393917
(2, 3, 1, 0)	-252.	-252.	-243.
(3, 0, 1, 2)	-144.	-153.	-144.
(3, 0, 2, 1)	-90.	-81.	-81.
(3, 1, 0, 2)	-117.	-117.	-117.
(3, 1, 2, 0)	33.93477328	47.11611743	55.58444128
(3, 2, 0, 1)	-207.	-207.	-207.
(3, 2, 1, 0)	78.85499701	17.49915814	23.76250987

4.3.1 Methodology

Actions/Reward

The actions (Table 4.1) and rewards are defined in the same way as our previous experiment where the state space was represented by different arrangements of the indices.

States

In our new state space (Figure 4.1), we defined each state as all the previous states that have been visited. After action 1, action 2, and action 3 a certain set of states can be reached, so there was no way that we could reach a state that is accessible at a previous step. For instance, from state 4, we cannot go to state 1. We always start at state 0 (not represented

in Figure 4.1), which corresponds to no actions taken. As a result, there were 1 + 3 + 9 + 27 = 40 total states since were 3 different actions that can be taken from each state and a total of 3 actions can be selected.

Action 1	Action 2	Action 3
{(0,): 1, (1,): 2, (2,): 3}	{(0, 0): 4, (0, 1): 5, (0, 2): 6, (1, 0): 7, (1, 1): 8, (1, 2): 9, (2, 0): 10, (2, 1): 11, (2, 2): 12}	{(0, 0, 0): 13, (0, 0, 1): 14, (0, 0, 2): 15, (0, 1, 0): 16, (0, 1, 1): 17, (0, 1, 2): 18, (0, 2, 0): 19, (0, 2, 1): 20, (0, 2, 2): 21, (1, 0, 0): 22, (1, 0, 1): 23, (1, 0, 2): 24, (1, 1, 0): 25, (1, 1, 1): 26, (1, 1, 2): 27, (1, 2, 0): 28, (1, 2, 1): 29, (1, 2, 2): 30, (2, 0, 0): 31, (2, 0, 1): 32, (2, 0, 2): 33, (2, 1, 0): 34, (2, 1, 1): 35, (2, 1, 2): 36, (2, 2, 0): 37, (2, 2, 1): 38, (2, 2, 2): 39}

Figure 4.1: State Space for Sorting 4 Numbers
The element in parenthesis represents the previous actions, and the number after the previous actions refers to the id of the state.

4.3.2 Results

With this new approach, we were able to learn an algorithm for sorting 4 numbers (Table 4.5).

Table 4.5: Accuracy for Action embedded State Space

Training Accuracy	Testing Accuracy
1.0	1.0

When we evaluated the policy for the state space, we got the following sequence of actions:

$$Action\ 2 : (0, 3), (1, 2)$$

$$Action\ 0 : (0, 1), (2, 3)$$

$$Action\ 2 : (0, 3), (1, 2)$$

Using this sequence of actions, we were able to sort all permutations of arrays, which demonstrated that we can learn how to sort 4 numbers using reinforcement learning.

4.4 Extension to Sorting 8 Numbers

4.4.1 Limited Actions

Action

We defined the actions as all of the actions that were present in the Bitonic Sorting Network (Table 4.6).

Table 4.6: Actions for Sorting 8 Numbers

Action ID	Pair 1	Pair 2	Pair 3	Pair 4
0	(0, 1)	(2, 3)	(4, 5)	(6, 7)
1	(0, 3)	(1, 2)	(4, 7)	(5, 6)
2	(0, 7)	(1, 6)	(2, 5)	(3, 4)
3	(0, 2)	(1, 3)	(4, 6)	(5, 7)

Reward

The reward was also still the same as previous experiments: 0 if non-terminal state, 10 if terminal state and sorted, and -10 if terminal state and not sorted.

States

There were a total of 5461 different states (Table 4.7) since we were allowing 6 total actions to be selected.

Table 4.7: Number of States for Sorting 8 Numbers

# of Actions Completed	# of states
0	1
1	4
2	16
3	64
4	256
5	1024
6	4096
Total	5461 states

4.4.2 Result

When we evaluated the policy from the start state, then we got the following sequence of actions to follow: [0, 1, 0, 2, 3, 0].

$$Action\ 0 : (0,1), (2,3), (4,5), (6,7)$$

$$Action\ 1 : (0,7), (1,6), (2,5), (3,4)$$

$$Action\ 0 : (0,1), (2,3), (4,5), (6,7)$$

$$Action\ 2 : (0,7), (1,6), (2,5), (3,4)$$

$$Action\ 3 : (0,2), (1,3), (4,6), (5,7)$$

$$Action\ 0 : (0,1), (2,3), (4,5), (6,7)$$

Table 4.8: Accuracy for Action embedded State Space

Training Accuracy	Testing Accuracy
1.0	1.0

This sequence of actions gave us an accuracy of 100% for sorting all permutations of our test arrays (Table 4.8). The sequence of actions also mirrored the bitonic sorting network for 8 numbers (Figure 1.1). This showed that when we restricted the allowable actions to the actions used in the bitonic sorting network, we could learn the bitonic sorting network.

4.4.3 Discussion

Even though we were able to learn the bitonic sorting network by using Q-learning, we limited the action space to all of the actions used in the bitonic sorting network: only 4 different sequence of pairs were considered. There were at most 105 actions that can be taken, so our action space was severely limited. The number of actions can be computed by selecting all pairs of 2 different elements to compare and swap, and then dividing by the

number of permutations of these pairs.

$$\text{max number of actions} = \prod_{i=0}^{\frac{n}{2}-1} \binom{n-2i}{2} \frac{1}{\frac{n}{2}!}$$

$$= \frac{n!}{\left(\frac{n}{2}!\right)2^{\frac{n}{2}}}$$

$$\text{For } n = 8, \frac{8!}{\left(\frac{8}{2}!\right)2^{\frac{8}{2}}} = 105$$

If we express our states as a function of our actions, then the total number of states increases exponentially.

$$1 + 105^1 + 105^2 + 105^3 + 105^4 + 105^5 + 106^6 = 1,431,404,647,262 \text{ states} \approx 1.4e12$$

This implied that we could not use this Q-learning approach to learn a sorting algorithm if we represent our states as a function of all the possible actions (105 actions). Therefore, we could only learn how to sort arrays of length larger than 8 using Q-learning if we only considered the actions needed for bitonic sorting the larger length arrays. We learned that by considering all of the actions possible, we need to use a different method such as deep reinforcement learning or to re-express the state space in another way.

CHAPTER 5
CONCLUSION

This study provided some insight on the difficulties of training neural networks in order to create parallel sorting algorithms. It showed that it is very challenging to create neural networks where each hidden layer is interpretable as the results of the neural networks could not be explained. The study also highlighted that deep reinforcement learning may help rectify some of the issues as we would be able to specify the actions (swapping and comparing), which could not be learned even independently. More importantly, it demonstrated that there is potential for parallel sorting algorithms to be created using neural networks. Using deep learning for designing algorithms may result in new findings that previously were not considered since deep learning uses mathematical functions to find patterns in data that cannot be seen otherwise. Therefore, this research provides a foundation and example for future work that can be done to represent theoretical problems that can be solved using deep learning.

5.1 Future Work

In the future, we would like to further evaluate the Index embedded State Space to see whether we can make the environment more conducive to learning sorting algorithms. For instance, we did not penalize the agent for returning to the same state, which caused the agent to frequently revisit states. The optimal policy learned, however, should cause the agent to move to different states after every action, implying that we need to penalize if we revisit or remain at the same state. We also did not have a separate terminal state when the array was sorted, which prevented the Q-values from converging since each state served multiple purposes: terminal and non-terminal states. A potential solution would be to create a dedicated sort state that would be accessible from all of the states, and we could check

at every state whether we should directly go to the sort state. Upon finding a good state space, we would also like to explore using neural networks to approximate the Q-function for sorting larger sequences of number - 16 or 32 element arrays.

The Action embedded State Space provided promising results when the actions were limited to the pairs of compare and swaps needed for the bitonic sorting network. We were able to learn the bitonic sorting network for 8 numbers using the limited actions, but the problem was no longer practical when we considered all the actions due to the factorial raised to exponential number of states. Therefore, we could further explore whether it's possible to always learn the sorting network given only the necessary actions for larger arrays.

REFERENCES

[1] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring), Atlantic City, New Jersey: Association for Computing Machinery, 1968, pp. 307–314, ISBN: 9781450378970.

[2] Wikipedia contributors, *Bitonic sort network with eight inputs — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/wiki/Bitonic_sorter#/media/File:Batcher_Bitonic_Mergesort_for_eight_inputs.svg", [Online; accessed 12-November-2020], 2009.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017.

[4] Wikipedia contributors, *Artificial neural network with layer coloring — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Colored_neural_network.svg, [Online; accessed 12-November-2020], 2013.

[5] Jeremy Zhang, *Gridworld*, https://towardsdatascience.com/reinforcement-learning-implement-grid-world-from-%20scratch-c5963765ebff, [Online; accessed 17-November-2020], 2013.

[6] Y. Perl, "Better understanding of batcher's merging networks," *Discrete Applied Mathematics*, vol. 25, no. 3, pp. 257–271, 1989.

[7] E. Solomonik and L. V. Kalé, "Highly scalable parallel sorting," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[9] Hecht-Nielsen, "Theory of the backpropagation neural network," in *International 1989 Joint Conference on Neural Networks*, 1989, 593–605 vol.1.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[11] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," *CoRR*, vol. abs/1702.05659, 2017. arXiv: 1702.05659.

[12] X. Zhu, T. Cheng, Q. Zhang, L. Liu, J. He, S. Yao, and W. Zhou, *Nn-sort: Neural network based data distribution-aware sorting*, 2019. arXiv: 1907.08817 [cs.DS].

[13] T. Tambouratzis, "A novel artificial neural network for sorting," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 29, no. 2, pp. 271–275, 1999.

[14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018, ISBN: 0262039249.

[15] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, 2013. arXiv: 1312.5602 [cs.LG].

[17] Y. Li, F. Gimeno, P. Kohli, and O. Vinyals, *Strong generalization and efficiency in neural programs*, 2020. arXiv: 2007.03629 [cs.LG].

[18] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1995.

[19] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *J. Artif. Int. Res.*, vol. 4, no. 1, pp. 237–285, May 1996.