THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ANALYSIS OF NEURAL NETWORK METHODS FOR SOLVING STRASSEN'S ALGORITHM

KALEY CHICOINE
SPRING 2017

A thesis
submitted in partial fulfillment
of the requirements
for baccalaureate degrees
in Computer Science
with honors in Computer Science

Reviewed and approved* by the following:

John Sustersic
Professor of Computer Science
Thesis Supervisor

Jesse Barlow
Professor of Computer Science
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Abstract

Neural networks are a machine learning technique modeled after clusters of biological neurons. They have shown great capacity in solving computing problems of an imprecise nature, and have led to large advances in the areas such as image and facial recognition. However, neural networks also have their limits. When applied to a problem that has a numerically precise answer, neural networks are likely a suboptimal technique. As an exercise in the limits of neural network capabilities, neural network methods of learning were applied to learn Strassen's algorithm for 2×2 matrix multiplication. However, even with the state space extremely restricted, the network failed to converge on the correct answer. This suggests that problems with precise solutions should be approached with more symbolic methods. Neural networks are good tools for abstracting concepts, but more complex learning will require multiple layers of abstraction and careful selection of error function to produce meaningful results.

Table of Contents

List of Tables

Acknowledgements

Many thanks to Dr. John Sustersic and Sara Jamshidi for supporting me throughout the research and writing process.

Chapter 1

Introduction

1.1 Motivation

Neural networks are a programming paradigm that enables algorithms to learn patterns in large data sets [1]. Recent advancements in parallel and distributed computing have brought neural networks into the spotlight as these systems' capabilities allow programs to be bigger than ever before while still speeding up overall runtime. However, just like any computational technique, neural networks have their limitations. Neural networks still largely operate as black box procedures, with the specifics of the learning parameters as much a game of guess-and-check as a meticulously thought-out process. As such, neural networks have proven extremely capable of solving certain types problem sets through pattern identification. The kinds of problems where neural networks have proven useful are generally "fuzzy" problems, or problems with non-precise solutions where an approximate result is sufficient. Fuzzy problems are frequently classification problems, with solutions not reliant on every parameter being correct. Instead, a confidence threshold needs to be met for the network to recognize the solution. For example, neural networks have led to large advancements in face and object recognition and image processing [2]. Facial recognition algorithms needs to pass a certain uncertainty threshold before the network correctly identifies a face in an image. Precise numerical problems do not have this luxury and rely on every component to be configured exactly correctly. We know that the human brain can solve both imprecise and precise problems, but the crossover between neural networks and precise problems is minimal. This paper will look at the limitations of discovering a precise algorithm using neural network methods. In particular, we focus on designing a neural network to find Strassen's algorithm.

1.2 Strassen's Algorithm

Volker Strassen developed a method of multiplying 2×2 matrices that reduces the required number of multiplications from eight to seven using a series of simple equations [3]. Let us define the matrices A and B to be as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \tag{1.1}$$

The traditional approach to matrix multiplication $A * B = C$ produces the output matrix:

$$C = \begin{bmatrix} A_{11} * B_{11} + A_{12} * B_{21} & A_{11} * B_{12} + A_{12} * B_{22} \\ A_{21} * B_{11} + A_{22} * B_{21} & A_{21} * B_{12} + A_{22} * B_{22} \end{bmatrix} \tag{1.2}$$

This approach uses eight multiplications and four additions for the 2×2 case, and runs in $O(n^3)$ for the general case. Strassen's algorithm gives us a more efficient method by defining intermediate matrices composed of seven multiplied pairs of elements from matrices A and B:

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) * (B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22}) * B_{11} \\ M_3 &= A_{11} * (B_{12} - B_{22}) \\ M_4 &= A_{22} * (B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12}) * B_{22} \\ M_6 &= (A_{21} - A_{11}) * (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) * (B_{21} + B_{22}) \end{aligned} \tag{1.3}$$

Each of these matrices $M_1 ... M_7$ uses only one multiplication, leading to a total of seven multiplications instead of the eight necessary for traditional matrix multiplication. Our final matrix C can be expressed in terms of M_k by adding the intermediate matrices:

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned} \tag{1.4}$$

This process can be applied to larger matrices by subdividing each matrix into 2×2 matrices and recursively computing the products. If a matrix is not of the form $2^n \times 2^n$, we pad the edges of the matrix with zeros until it has the necessary number of rows and columns. This method reduces the time complexity of large matrix multiplication from $O(n^3)$ to $O(n^{2.8})$ [4].

Strassen's algorithm has only been found for the 2×2 matrix case. It has been shown that the 3×3 matrix case may take as few as 19 multiplications–the precise number remains unknown–and the traditional naive approach takes 27 multiplications [5].

1.3 Neural Networks

We attempt to use neural network methods to recreate Strassen's algorithm in the 2×2 case in a transparent way that breaks the algorithm down into multiple steps which allows us to ensure the neural network is actually finding the optimal number of multiplications for the matrix and not solving the problem in another manner. Additionally, we hope to gain a better understanding of exactly how neural networks learn and gain insight into which kinds of problems are best suited for these algorithms. While neural networks have been used to mimic Strassen's algorithm [6], it has never been done in a manner that can prove that the network is finding the optimal solution and not simply finding another, non-optimal method of multiplying two matrices.

We use a standard gradient descent method for training the network. Gradient descent is an algorithm for minimizing functions. Given a set of initial parameters, it iteratively moves closer to the optimal solution by changing the set of weights of the neural network so that the function always moves in a negative direction. Essentially, the algorithm looks to change weight values in a way that only decreases error, thus always moving down toward the minimum error.

Previous work has been done using neural networks and matrix multiplication. A paper published to arXiv by Viet Elser claimed to use a network with a single hidden layer to solve Strassen's algorithm in the 2×2 case. They used a single hidden layer with seven neurons to represent the seven multiplications required by Strassen's algorithm. However, they also use a nonstandard error function (tensor rank). We used a typical least squares error function with gradient descent and were not able to achieve the same results, even with an extremely restricted problem set. This goes to show that the error function plays a huge role in determining the success of neural networks, and that sometimes the standard approach to error functions is not sufficient for more precise numerical problems.

Chapter 2

Methodology

2.1 Designing the Network

In order to create a neural network which finds Strassen's algorithm, we first have to look at how the algorithm is constructed, and then transform the mathematical processes to a neural network structure. Strassen's algorithm involves two main steps: creating the intermediate matrices and selecting which of the intermediate matrices should be included in the summations of the final matrix's components. The first step, creating the intermediate matrices, is the most complicated. First, a summation of one or two elements from each of the multiplicand matrices needs to be found for each intermediate matrix. Then, each contributing summation from A and B are multiplied to create the multiplication of two sums for each intermediate matrix. The components of these summations can have either a positive or negative weight.

For example, intermediate matrix M_4 from equation 1.3 is made up of two parts: from multiplicand matrix A, the value of A_{22} is taken. From multiplicand matrix B, we have the summation of B_{21} with the negative of B_{11}, giving us the final equation

$$M_4 = A_{22}(B_{21} - B_{11})$$

Once the correct multiplications of summations from A and B have been determined, the final values of the result are a matter of weighting these multiplications with either -1, 1, or 0 to represent whether they show up in the calculation of the final values (equations 1.4). In order to limit the scope of the problem and test in a smaller state space, we started by only looking at the last step: the weighting of the intermediate matrices. Since the weights of the matrix need to be -1,0, or 1, the best representation of this step of the algorithm is to use a perceptron network that allows for negative weights.

The entire network needs three hidden layers, each of which represents one step in the algorithm. The first layer selects the correct summations from each multiplicand matrix. The second layer combines the summations from the first layer to create the intermediate matrices. The third layer would then be responsible for selecting the correct weights (0,1,-1) for each intermediate matrix, which would lead to the final result. The input layer consists of eight inputs, four each from the two multiplicand matrices, and the output layer is merely four outputs representing the values present in the product matrix.

2.2 Experimental Bounds

Since little research has been done using neural networks to extract algorithms from data, we initially looked at a single layer perceptron network representing the last hidden layer of a functional network. By giving the network the initial information, the hardest steps were abstracted out to see if the network could converge correctly in the smallest state space. We generated eight random numbers to represent two 2×2 matrices:

```
input[0,1,2,3,4,5,6,7,8]:
```

$$A = \begin{bmatrix} input[0] & input[1] \\ input[2] & input[3] \end{bmatrix} \; B = \begin{bmatrix} input[4] & input[5] \\ input[6] & input[7] \end{bmatrix} \tag{2.1}$$

We then combined these numbers to produce each combination of correct summations for each of the input matrices. From Strassen's algorithm, we know that the correct summations from the two input matrices A and B:

A Summations	**B Summations**
$A_{11} + A_{22}$	$B_{11} + B_{22}$
$A_{21} + A_{22}$	B_{11}
A_{11}	$B_{12} - B_{22}$
A_{22}	$B_{21} - B_{11}$

Table 2.1: Input Summations

Combining these summations gives 49 multiplications which includes the correct seven needed for Strassen's algorithm (shown in table 2.2 in bold):

$(\mathbf{A_{11}} + \mathbf{A_{22}}) * (\mathbf{B_{11}} + \mathbf{B_{22}})$	$(A_{11} + A_{12}) * (B_{11} + B_{22})$
$(A_{11} + A_{22}) * (B_11)$	$(A_{11} + A_{12}) * (B_11)$
$(A_{11} + A_{22}) * (B_{12} - B_{22})$	$(A_{11} + A_{12}) * (B_{12} - B_{22})$
$(A_{11} + A_{22}) * (B_{21} - B_{11})$	$(A_{11} + A_{12}) * (B_{21} - B_{11})$
$(A_{11} + A_{22}) * (B_{22})$	$(\mathbf{A_{11}} + \mathbf{A_{12}}) * (\mathbf{B_{22}})$
$(A_{11} + A_{22}) * (B_{11} + B_{12})$	$(A_{11} + A_{12}) * (B_{11} + B_{12})$
$(A_{11} + A_{22}) * (B_{21} + B_{22})$	$(A_{11} + A_{12}) * (B_{21} + B_{22})$
$(A_{21} + A_{22}) * (B_{11} + B_{22})$	$(A_{21} - A_{11}) * (B_{11} + B_{22})$
$(\mathbf{A_{21}} + \mathbf{A_{22}} * (\mathbf{B_11})$	$(A_{21} - A_{11}) * (B_11)$
$(A_{21} + A_{22} * (B_{12} - B_{22})$	$(A_{21} - A_{11}) * (B_{12} - B_{22})$
$(A_{21} + A_{22} * (B_{21} - B_{11})$	$(A_{21} - A_{11}) * (B_{21} - B_{11})$
$(A_{21} + A_{22} * (B_{22})$	$(A_{21} - A_{11}) * (B_{22})$
$(A_{21} + A_{22} * (B_{11} + B_{12})$	$(\mathbf{A_{21}} - \mathbf{A_{11}}) * (\mathbf{B_{11}} + \mathbf{B_{12}})$
$(A_{21} + A_{22} * (B_{21} + B_{22})$	$(A_{21} - A_{11}) * (B_{21} + B_{22})$
$(A_{11}) * (B_{11} + B_{22})$	$(A_{12} - A_{22}) * (B_{11} + B_{22})$
$(A_{11}) * (B_11)$	$(A_{12} - A_{22}) * (B_11)$
$(\mathbf{A_{11}}) * (\mathbf{B_{12}} - \mathbf{B_{22}})$	$(A_{12} - A_{22}) * (B_{12} - B_{22})$
$(A_{11}) * (B_{21} - B_{11})$	$(A_{12} - A_{22}) * (B_{21} - B_{11})$
$(A_{11}) * (B_{22})$	$(A_{12} - A_{22} * (B_{22})$
$(A_{11}) * (B_{11} + B_{12})$	$(A_{12} - A_{22}) * (B_{11} + B_{12})$
$(A_{11}) * (B_{21} + B_{22})$	$(\mathbf{A_{12}} - \mathbf{A_{22}}) * (\mathbf{B_{21}} + \mathbf{B_{22}})$
$(A_{22}) * (B_{11} + B_{22})$	
$(A_{22}) * (B_11)$	
$(A_{22}) * (B_{12} - B_{22})$	
$(\mathbf{A_{22}}) * (\mathbf{B_{21}} - \mathbf{B_{11}})$	
$(A_{22}) * (B_{22})$	
$(A_{22}) * (B_{11} + B_{12})$	
$(A_{22}) * (B_{21} + B_{22})$	

Table 2.2: Multiplicand Summations

The last step of the network is to select the correct intermediate matrices for each final output. To accomplish this, the network was given the correct summations from each of the two multiplicand matrices. The task was to choose the correct weights for the combinations of summations. Assuming the previous layers of the neural network had correctly selected the summations from matrices A and B, we are left with seven summations from A and seven summations from B shown in table 2.1.

Combining these summations gives us 49 possible intermediate matrices of the form $Summation_A * Summation_B$ to feed our network. However, even simplified down to this level, the task of selecting the correct weights proves to be too much for a single neural network with a least squares error function to handle.

This simplified network consisted of a single fully connected layer. The 49 input combinations of each intermediate matrix mapped to each of the 4 output values, with the hope that the neural network would be able to correctly weight the inputs to select the ones needed to produce the correct outputs. Each iteration of the matrix ran for either a predetermined number of cycles, or until the error stopped decreasing. Since the error rarely decreased, cycles were used more frequently than error.

```
def main(self, trainingData, cycles, testData=None):
        self.trainingLength = len(trainingData)
        if testData:
            testLength = len(testData)
        thisError = 10000
        lastError = float("inf")
        thisCycle = 0
        while (thisError <= lastError and thisCycle < cycles):
            self.cost(trainingData)
            lastError = thisError
            if testData:
                error = self.evaluate(testData)
                thisError = max(error)
                print "thisError: {}".format(thisError)
            else:
                print "Cycle {0} complete".format(thisCycle)
            thisCycle += 1
```

Chapter 3

Testing and Results

3.1 Testing

A variety of methods were used to try and get the neural network to converge to the correct weight set for Strassen's algorithm. All of them failed, with most causing the network error to grow extremely quickly. This result was very unexpected due to the nature of gradient descent- the error is only supposed to decrease. The increasing error suggests the geometry of the space for matrix multiplication does not work well with gradient descent.

3.1.1 Initial Weights

We varied initial weights on the network. We tried initializing with all zeroes, all ones, all negative ones, a random selection of integers from [-1,1], a random selection of floats between [-1,1], a random selection of integers between [-10,10], and a random selection of floats between [-10,10]. The initial weights did not seem to have a particularly large impact on the convergence of the network, with the only difference being that the larger the range in initial rates, the faster the error grew.

3.1.2 Training Data

What caused the network to vary the most and perhaps holds the most interest for future study was how the variation of the input values effected the behavior of the network. First, we simply generated 100 sets of two matrices with random integers between 1 and 10 and calculated the result of their multiplication. This failed to produce meaningful weights or the correct answer. Interestingly, the number of instances in the training data did not seem to have any effect on the growth of the weights. We tried giving the neural network a training set of 5 sets of matrices, and this produced the same results as 100 and 500 sets: all failed to converge and had exponentially increasing error values. Using random decimal numbers between 1 and 10 and between -1 and 1 produced similar results.

Our next attempts limited the state space of the problem from all integer matrices to more specific matrices. We tried using only the binary matrices, made of zeroes and ones. This caused the error to rise more slowly, but after 10,000 iterations of the network, the error reached similar levels to that produced by integer matrices. Encouraged by this, we tried using rotational matrices:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{3.1}$$

With rotational matrices as input, the network did actually give the correct output after approximately 500 iterations. However, when fed any non-rotational matrix, it failed, showing that it was not actually learning to multiply matrices; it only learned the unique pattern shown when two rotational matrices are multiplied:

$$\begin{bmatrix} val_1 & -val_2 \\ val_2 & val_1 \end{bmatrix} \tag{3.2}$$

Looking at the weights confirmed this result. The weights did not converge to $0, 1,$ or -1 . In fact, none of the weights were larger than 0.02.

3.1.3 Learning Method

We used a standard back-propagation algorithm for learning. For the cost function, we initially used the rectified linear function:

$$X = \left\{ \begin{array}{lr} x & -1 \leq x \leq 1 \\ 1 & x > 1 \\ -1 & x < -1 \end{array} \right\}$$

We also tried mean squared error and a different rectified linear function:

$$X = \left\{ \begin{array}{lr} 0 & -0.5 \leq x \leq 0.5 \\ 1 & x > 0.5 \\ -1 & x < -0.5 \end{array} \right\}$$

None of the different cost functions had any effect on correctness of the network or the rate at which the error decreased.

3.2 Results

A sampling of the results from different trials is shown in Table 3.1. The columns in the table show the variety of adjustments made to the network in an attempt to achieve convergence. The first row indicates a network with the weight matrices correctly configured and shows that the network gives zero error with the desired configuration. Other interesting columns

- *Initial Weights:* Represents the initial value of the weight matrix. If "correct dist" is included, the weights which should have had a value of -1 or 1 were set to the value in the table, with the other weights in the weight matrix set to zero.

- *Error on Halt:* Shows the error when the algorithm terminated.

- *Training and Test Data:* The values used for training. LimitedRot signifies rotational matrices with $0 \leq \theta \leq 2\pi$.

- *Error Trend:* Describes behavior of the error. Interestingly, the only configuration which gave decreasing error was when both the testing data and the training data was set to rotational matrices, but as previously explained, the weights did not converge on the correct answer for any matrix but instead only converged to give the correct answer for rotational matrices.

Initial Weights	Error on Halt	Training Data	Test Data	Error Trend
1, correct dist	[0,0,0,0]	rand int	rand int	none
0.5, correct dist	all *10ˆ41	rand int	rand int	increasing
0.9, correct dist	all *10ˆ43	rand int	rand int	increasing
0.95, correct dist	all *10ˆ3	rand int	rand int	increasing
9.9900000000000003E-2	all *10ˆ40	rand int	rand int	increasing
0	[32.3 54.0 58.5 25.1]	limitedRot, 9 intervals	rand int	decreasing
0	[37.4 20.3 22.6 29.5]	limitedRot, 9 intervals	rand int	decreasing
from run 9	[37.4 20.3 22.4 29.6]	limitedRot, 9 intervals	rand int	static
0.999	[41.1 27.7 43.3 29.6]	rand int	rand int	increasing
0.999	[1.1e+20 7.6e+19 1.1e+20 8.2e+19]	rand int	rand int	increasing
0	[3.0e+7 2.0e+7 3.1e+7 2.2e+7]	rand int	rand int	increasing
0	[9.8e+44 6.7e+44 1.0e+45 7.2e+44]	rand int	rand int	increasing
0	[2.3e-07 1.6e-07 2.3e-07 2.3e-07]	limitedRot, 9 intervals	rot	decreasing
0	[3.7e+5 3.0e+4 4.3e+4 3.8e+5]	limitedRot, 9 intervals, weight 2	rand int	increasing
0	[7.5e+17 5.8e+16 5.7e+16 7.5e+17]	limited Rot, 9 intervals, weight 2	rand int	increasing
rand float -1:1	[1.6e+5 9.5e+5 5.6e+4 3.8e+4]	rand int	rand int	increasing
rand float -1:1	[8.2e+20 3.0e+21 1.5e+20 1.8e+20]	rand int	rand int	increasing
rand int -1:1	[8.0e+4 1.0e+5 9.3e+5 1.2e+4]	rand int	rand int	increasing
rand int -1:1	[1.2e+20 3.9e+20 2.7e+21 1.8e+18]	rand int	rand int	increasing
0	[1.5e+09 7.9e+07 1.7e+08 6.9e+08]	rand int	rand int	increasing

Table 3.1: Trial Results

Chapter 4

Discussion

While this approach with neural networks can theoretically find the correct distribution of weights to recreate Strassen's algorithm, it may be probabilistically highly unlikely using gradient descent. The geometry of the weight-error space may not have the typical bowl-shape landscape seen with problems solvable using gradient descent and standard error functions, meaning it could be possible to miss the optimal minimum but still be moving down the gradient of the space. While a critical point may exist, the geometry may be such that unless the function starts exactly correctly, the zero point will never be reached. The approach to this critical point in the weight-error space is so delicate that the probability of finding it randomly is practically zero. The work done by Elser suggests that for every such problem, there exists a unique error function that is necessary to produce meaningful results.

Neural networks are a fantastic tool for pattern recognition and for learning problems that do not have specific answers. However, they are insufficient to learn the very broad and abstract patterns found in mathematical reasoning. When humans learn math, we have a multilayered understanding of what is happening, abstracting out things like basic algebra from the larger problem of finding patterns in sets of numbers. We learn from rules, not data, and abstract out to find the rules that govern calculation. It would appear from the results of this study that neural networks are, on the basic level, incapable of such abstraction. Therefore, any attempts at teaching machines this layered type of thinking needs to also be layered in order to successfully mirror the way humans think and reason.

Bibliography

[1] Michael A. Neilsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[2] Paul Viola and Michael J. Jones. Robust, real-time face detection. *International Journal of Computer Vision*, May 2004.

[3] Volker Straussen. Gaussian elimanation is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[4] Steven S. Skiena. *The Algorithm Design Manual*. Steven S. Skiena, Berlin, New York, 1998. 8.2.3 Matrix Multiplication.

[5] Julian D. Laderman. A noncommutative algorithm for multiplying 3x3 matrices using 23 multiplications. *Bulletin of the American Mathematical Society*, 82(1), 1975.

[6] Veit Elser. A network that learns strassen multiplication. arXiv, January 2016.

Kaley Chicoine
Academic Vitae
Computer Science & Engineering

Contact Information	501 Vairo Blvd. Apt. 1123 State College, PA 16803	*Phone:* (515) 480-4064 *Email:* kmchicoine@gmail.com

Education

Pennsylvania State University, University Park, PA

B.S, Computer Science, *Expected:* May 2017
Advisor: John Sustersic, Ph.D. Minor: Mathematics

Research Experience

Solving Strassen's Algorithm Using Neural Networks, with Dr. John Sustersic and Shahrzad Jamshadi
Goal: see if neural networks can be used to solve precise problems
Coded perceptron network from scratch
Honors thesis topic

Facial Emotion Recognition in Nao Robot, with Dr. John Yen
Goal: incorporate emotion recognition into social humanoid robot's behaviors
Using Affectiva emotion recognition API

Applied Research Lab Internship Apr 2015 to May 2017
Penn State Applied Research Lab
Programmed steering system for autonomous vehicles
Attended SOAR workshop, University of Michigan, June 2015
Supervisor: John Sustersic, Ph.D.

Programming Languages

Confident with: Python, C/C++, Swift, Matlab, LaTeX

Some Experience with: HTML/CSS, Java, SOAR, SQL

Used It Once for Class: PHP, Adobe Solr

Special Distinctions

Schreyer Honors College Scholar	August 2012-May 2017
Distinguished Undergraduate Researcher	January 2015-May 2017

Extra-curriculars

Penn State Fencing Club
President (January 2015-December 2016) and Secretary (August 2013-December 2015)
Managed $10,000 budget and 100-person membership
Personally ranked in top 10% of fencers in the United States