Neural Networks
and Shape Identification

an Honors Project submitted by

Andrew D. Hansen
2021 N Sizer Ave
Jefferson City, Tennessee 37760
(865) 696-8509

a BS in Computer Science

5 March 2010

Project Advisor: Dr. Henry Suters

Contents

List of Tables

List of Figures

Chapter 1

Neural Networks

Computers have been made to solve problems both common and complex, using algorithms that range from the simple to the sophisticated. The issue with many of these problem-solving methods is that they must be created with every aspect of the problem in mind in order to achieve efficiency, which can make the process of writing them very difficult. This is where artificial intelligence steps in, the premise of which is to simulate human intelligence in a computer, which allows us to create a program with a much broader task in mind that can solve the same problem as the aforementioned minutely specific algorithm. This paper is concerned with only one particular type of artificial intelligence, namely that of neural networks, which allow computers to "learn" from the data they are shown. Neural nets are effective problem solvers, which this paper will show by demonstrating their effectiveness at identification, particularly when applied to shape recognition.

1.1 An Introduction

Artificial neural nets are modeled after their biological counterparts, which are comprised of nerve cells and the interactions between them. The nerve cells, called neurons, are connected by synapses, which are responsible for causing excitation and inhibition of the neurons [1]. In other words, the neurons are responsible for holding values of data passing through the network, while the weights of the synapses adjust the strength of these values from one neuron to another.

The usefulness of artificial neural nets comes from their ability to generalize [1]. To begin, a network must be trained on a series of patterns that are given as input values, which then pass through its neurons and synapses and are used to adjust the weights of the synapses. This step is repeated until the network has

"learned" the general features of the patterns and is ready to apply what it has learned to other similar patterns. Advantages of neural nets over other algorithms is that they have the ability to pick up on subtle details in the data that would otherwise have been glossed over, they work well with nonlinear and even noisy data, and they can even operate on information considered "fuzzy", which may be human opinion or have poorly defined categories or large error [6].

Another great advantage of neural nets is that they need only the training data, and from that they are able to learn patterns of which the user may not even be aware [9]. Whereas the programmer is required to have detailed knowledge of the data in order to write a usable traditional algorithm, neural nets are widely adaptable and require only to be shown accurate data and given time to train. As an example, let us say we have a coordinate plane, upon which are a triangle and a square. We can write an algorithm that takes the vertices of these shapes and identifies them by counting the number of vertices per shape, where a triangle has three and a square has four. However, if we were to move from this particular coordinate plane to a checker board with each piece laid out in such a way as to resemble the two shapes, there would no longer be any vertices but instead only a discrete collection of points. This would remove the effectiveness of the algorithm and require that a new one be written. But if we have trained a neural network with enough cases so that it truly learns each shape, then it has the ability to generalize and to identify each shape no matter the situation.

It is not hard to see that there are many uses for neural nets, particularly in fields that use pattern matching such as with prediction, noise reduction, or shape recognition [6]. For instance, if trained on accurate and plentiful meteorological data, a neural net could predict the weather. Or when dealing with home security, a neural net could be shown pictures of the residents in a particular house and then be used to identify intruders. And as for currently implemented applications, neural nets are being used on the space shuttle to monitor the behavior of a hydrogen flow control valve.

1.2 Neural Nets In General

The basic process of using a neural net has four stages: training, validation, testing, and deployment. The network's ability to learn is its basis of usefulness, and this all begins with training, the exact process of which is determined by the type of neural net. During this phase, the data used to train the network is taken as a sample of the overall batch of data and called the set of training cases [6]. Each pass through this training set is accompanied by an adjustment of the synapse weights in an attempt to reduce the overall error, which in this case is

the difference between the actual output of the network and the desired output. Each iteration of this process is called an epoch [3], and the maximum number of these can be adjusted to generate better results from training.

There are two types of training that can be used: supervised and unsupervised. In supervised training, the network is given a set of inputs for each training case and a corresponding set of correct outputs. This allows the network to see the error between its actual output and the desired output, and change its synapse weights accordingly [6]. As a parallel, remember that training a dog involves rewarding good behavior and punishing bad. With neural nets, the 'rewards' and 'punishments' are simply increases or decreases in the weights of each synapse [9]. In unsupervised training, the goal is not to produce outputs, but to reveal patterns in the data. A similar process is used, except without the desired output values being revealed [6]. Depending on whether the network's training is supervised or unsupervised, it will most likely be used for either pattern recognition or pattern reconstruction, respectively [9].

After training is completed, the network goes through validation and testing. Validation uses the remainder of the overall data to test how well the network was trained [6], which is done by running that data through the network and checking the resulting output against the desired output. This gives the network a chance to test its effectiveness on different data, which helps to prevent overfitting of the synapse weights to the specific data on which it was trained [6]. In testing, the user gets a chance to specify certain inputs and see what the network produces as output in order to ensure that the network is accurate and able to generalize. The network can then be deployed, meaning the weights of each synapse are saved in a file for later use. At this stage, desired output values are no longer needed, so the neural net may use inputs for further testing or for the actual task for which it was trained.

1.3 Various Neural Nets

There are many different types of neural networks in existence, some better suited to particular problems than others. Here is a glimpse at three of the more well-known networks.

1.3.1 Hopfield Networks

Also known as an attractor network, Hopfield networks have proven themselves useful for pattern reconstruction. In other words, they perform well when completing data with holes, similar to the role of interpolation, or identifying the

correct pattern from corrupt input [1].

Given the associative nature of the Hopfield net, it also has the ability to approximate a solution to the famous 'Traveling Salesman' problem [3]. In this situation, we have a salesman who must travel between each city in his vicinity, and the route between each of them holds a certain cost for the salesman. The goal is to find the cheapest series of routes that allows the salesman to stop in every city [2]. An algorithm does not yet exist that can efficiently solve this problem, so we must instead resort to testing every possible outcome, a task which would very likely take far more than a practical amount of time to complete. The implications of finding a solution are evident when we consider postal services and the like, making the Hopfield net's ability to find approximate solutions extremely useful in such situations.

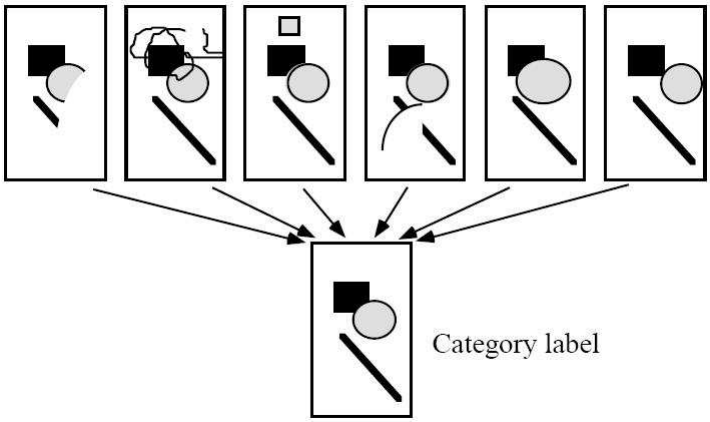

Figure 1.1: Hopfield network pattern matching [1]

The structure of the Hopfield net consists of a network of binary neurons connected with symmetric synapses [1]. Each neuron has the value of either -1 or +1, which allows for incrementing or decrementing weights [8]. During the input stage, some of these neurons are assigned initial values. On subsequent passes through the network, the update rule determines the activity of a neuron based on the "remembered" values of other neurons from preceding passes. The values of these preceding neurons are multiplied by the weights of the synapses connecting them to the neuron in question, and the sum of all these multiplications determines the activity or inactivity of that neuron [1]. Once values for all the neurons in the network are calculated and training is finished, then pattern reconstruction is completed.

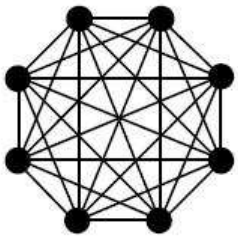

Figure 1.2: Hopfield Network [1]

1.3.2 Kohonen Networks

Kohonen networks use unsupervised training in order to self-organize, which in turn makes them very useful for classification. More specifically, they use a type of unsupervised training called competitive learning, where only one or a few neurons change their weights for each pattern presented during training [6]. The network has a structure of neurons in two layers, the first being a normalized set of the inputs that is then used to calculate the values of the neurons in the output layer. The key to Kohonen networks that distinguishes them from other types of self-organizing nets is that neurons with similar weights are grouped closer together in the network, a product of the competitive learning. When an input is shown to the net, it is shown to each neuron, and the more similar each neuron's set of weights is to the input, the larger response that neuron will give, thereby becoming the winner in the competition [8].

1.3.3 Feedforward Networks

The feedforward network has a structure consisting of at least two layers, for input and output, with any number of hidden layers between. It uses supervised training, where inputs are run through the network and the results are then compared against the desired output values, at which point the synapse weights are corrected in a process called backpropagation. This network is further explained in the next chapter.

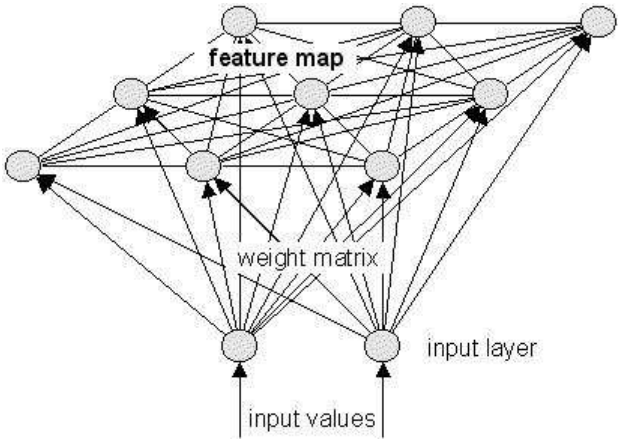

Figure 1.3: Kohonen Network [5]

Chapter 2

Multilayer Feedforward Neural Networks

This paper will concentrate on one type of neural net: the multilayer feedforward network. Unlike other types of neural nets, which are generally used for pattern reconstruction, the feedforward net produces a definite set of output values. The benefit is that, once trained, this network can use given inputs to produce output values which the user may apply for practical use. Because of its ability to produce outputs and its effectiveness in pattern recognition, the feedforward net is now the most widely used neural network for commercial applications [1] and has proven very successful with prediction, categorization, and other such applications.

2.1 Structure

The structure of a feedforward net is comprised of at least two layers, an input and an output, with any number of hidden layers between. The number of neurons in the input layer is equal to the number of input values to be given to the network, and likewise the number of output neurons matches the required number of outputs. There does not have to be a certain number of hidden layers or even a particular size for each one, and their details vary greatly from network to network based on the specifics of the task for which the net is being used. It is often the case that each hidden layer's exact size must be determined by trial and error in order to find the setup that will generate good results from the network. Each neuron in a layer is connected to each neuron in the adjacent layer via synapses, which at first hold randomized weights.

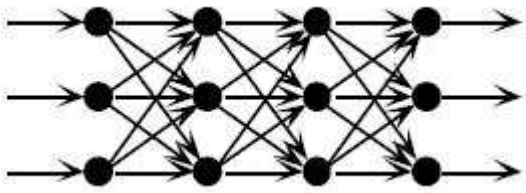

Figure 2.1: Feedforward Network [1]

2.2 Neural Net Learning

The usefulness of a neural net comes from its ability to "learn" based on the given training cases. These cases consist of inputs and their corresponding desired outputs. The values of these and all other neurons are real numbers within a certain range, such as 0 to 1. In order to achieve successful training, the set of cases must represent all the types of problems the network will be solving [9].

Learning is accomplished in the training stage, where data from each case is given to the input layer and run through the network to the output layer. The values of the output layer's neurons become the output values of the network, which are then compared against the desired output values to find the error. This is used to see by how much the synapse weights must be altered in order to generate more accurate output during subsequent passes.

2.2.1 Calculating An Output

The first step in training is running the data through the network, once for each training case. Input data for the first case is stored in the input layer, one real number value for each neuron. Calculation for each neuron in the following layers, from the first hidden layer to the output layer, is done one neuron at a time. Values from the neurons in each layer are used to calculate the neuron values in the next layer, hence the name "feedforward." It should be noted that in the following explanations, only one hidden layer is used, though there may be any number of hidden layers in a feedforward network.

When calculating neuron values, the result may fall outside the required range, and so we use the sigmoid function to adjust the value of each neuron to fit into that range. This is done through the use of a function, such as the hyperbolic tangent [1]. The range of the value returned to the neuron depends on the sigmoid, and so the function used must be chosen to match the range of the training set.

The usual values that these functions receive as parameters are sums, as seen in the following algorithm.

We begin by calculating the value of the first neuron in the hidden layer, which we will call H_1. To do this, we need to take the values of each neuron in the input layer, denoted I_i, and the weight of the synapse connecting each I_i to H_1, which is called S_{i1}^H and at this point still holds its initial random value. The double subscript in S_{i1}^H denotes its connection between the ith neuron in the input layer and the first neuron in the hidden layer. We now sum the product of each I_i and S_{i1}^H and pass this value to the sigmoid function. In equation form, we get

$$H_1 = \text{Sigmoid}(\sum(I_i * S_{i1}^H))$$

Repeat this procedure for every neuron H_j in the hidden layer to acquire their values, and then finally for the output layer [4].

2.2.2 Backpropagation

At this point, no actual training has been done and the values for each synapse weight are very likely nowhere close to being able to generate correct output. So what needs to be done is an adjustment of the weights, each one being altered in a way that allows the network to generate outputs closer to their desired targets. This is where backpropagation steps in, which is the process of taking the existing weights and the error, and calculating new values for weights that will allow for better network performance.

Before we begin backpropagation, we need to know the mean squared error (MSE) for each training case. The MSE is the sum of each squared difference of the actual output for a given neuron in the output layer and the desired output for that neuron [4]. In equation form where n is the number of neurons in the output layer, we have

$$MSE = \sum_{1}^{n}(actual - desired)^2$$

The MSE is used to determine whether or not backpropagation is necessary. It is calculated after each run through the network and compared against a specified minimum error. If it has gone below this minimum error, then the network has the necessary weight values to generate reasonably correct output. However, if it has not gone below this minimum, then the weights need to be updated, and the real work of backpropagation begins.

We now want to find the gradient, or the direction of steepest descent, and to do so we must calculate the delta values, which are used to calculate by how much

each synapse weight must be changed. We begin with the synapses denoted S_{ij}^O, which are between the hidden layer and the output layer, where i and j index the neurons in those layers, respectively. For instance, S_{23}^O is the synapse connecting the second neuron in the hidden layer to the third neuron in the output layer. To find the delta values, we use the chain rule to calculate the derivative of the MSE with respect to each S_{ij}^O to get

$$delta_{ij}^O = \text{Sigmoid}'(\sum_i (S_{ij}^O * H_i)) * (2 * (desired - actual)) * H_i$$

For easier understanding, we will look at the resulting derivative in pieces, the first of which takes each neuron in the hidden layer, called H_i, and finds the sum of each H_i times S_{ij}^O, as we see in

$$\sum_i (S_{ij}^O * H_i)$$

We then find the derivative of our sigmoid function [1], and call this function Sigmoid$'$. The parameter given to Sigmoid$'$ is the error, which is

$$2 * (desired - actual)$$

where

$$actual = \sum_i (S_{ij}^O * H_i)$$

These values are combined with H_j to get the delta value for each synapse. We repeat this process for each of the j neurons in the output layer.

We use a similar process for synapses S_{ij}^H between all other layers, where j indexes the layer between the layers indexed by i and k, i indexes the layer adjacent to j on the input side, and k indexes the layer on the output side. In this case, we again use the chain rule to calculate the derivative of the MSE with respect to each S_{ij}^H, where

$$delta_{ij}^H = \sum_k (delta_{jk}^O * S_{jk}^H * \text{Sigmoid}'(\sum_i (S_{ij}^H * H_i)) * H_i)$$

Similarly to our first step of backpropagation, we calculate our parameter for the derivative of the sigmoid function as

$$\sum_i (S_{ij}^H * H_i)$$

Then we take each synapse labeled S_{jk}^H, which are between layers indexed by j and k, and multiply these by their previously calculated delta values, denoted $delta_{jk}^O$.

We then multiply these by the value returned from the derivative of the sigmoid function and H_j. The delta value for the current synapse is calculated as a sum of these products [1]. We repeat this process for each of the j neurons in the layer.

After calculating every delta value, we acquire the global deltas for the epoch by summing each delta value from every training case. This helps to prevent overfitting of the training data and allows the network to be useful on more than just the specific cases on which it was trained [6]. So where m indexes each training case, we have

$$global = \sum_{1}^{m} delta_{ij}$$

The delta values are then multiplied by a previously chosen step size and used to alter the synapse weights [9]. This calculation of steepest descent hopefully results in a decrease in error, meaning we have come closer to the necessary weight configuration.

A good way to visualize this process is to imagine oneself on a hillside that makes up one side of a valley. We start anywhere on this hill, and gradually work our way to the lowest point in the valley. This is like our weight configuration, which is initially randomized but then adjusted during backpropagation so that it slowly approaches the minimum error through a process called convergence [9]. Complications may arise from the fact that there are many smaller valleys, or divots, on this hillside in which the network can get stuck while moving down the hill. When choosing a step size, we must keep in mind that it should be small enough so that we will not continually overshoot the bottom of the valley in which we are trying to land but large enough so that we do not spend too much time stepping down the hill or find ourselves stuck in a divot and unable to step out of it. This is also why we randomize our initial synapse weights, so that if one initial set causes us to get stuck in a divot, perhaps another will not [9]. Nevertheless, we always run the risk of reaching the maximum number of epochs and quitting without convergence, so the step size must be chosen carefully.

2.2.3 The Bias

Depending on the problem being solved by the network, it may be necessary to implement a special neuron called the bias. This neuron is always equal to 1, no matter the situation, and is never altered. It is not actually a neuron in any particular layer, but instead is a standalone neuron that connects to every non-input neuron. Its synapse weights are also adjusted in backpropagation [6]. The bias allows the network to produce output values that it would otherwise not be able to, such as when input is small and output needs to be large, or when all

input values are 0 and therefore all output values are also 0 even if they need to have another value.

2.2.4 After Training

After the network is finished training it enters the deployment stage, where we may choose to save the synapse weights to a file for later use. When convergence was reached and testing shows that the network was successfully trained, deployment allows us to have accurate weights at any time. This means that we will not need to retrain each time the network is used in order to reacquire a useful weight configuration, and we may now use the network for its intended purpose.

Chapter 3

From XOR to Grayscale

The bulk of this project involved creating a multilayer feedforward neural net in C code called `neural_net.c`. To show the effectiveness of a neural network's ability to classify, I ran several training sets using this program. This chapter contains a description of the code and the results of these tests, starting with a simple XOR network and working up to complex gray scale images.

3.1 The Code

To use the program, simply run it and provide it with either a '.train' or '.weight' file. The train files contain data for the size of the network, maximum number of epochs to be run, number of training cases (all are used for training, as validation is not implemented), size of input grids, and the data itself. The data is in the form of grids that may have any rectangular shape, though for the training sets referenced in this paper they are always squares. These grids are further explained elsewhere in this chapter.

When the network is finished training, either from convergence or reaching the maximum number of epochs, the program looks for a '.test' file of the same name as the train file. This file contains desired outputs and corresponding grids similar to the inputs used by the train file, which are run through the network to test the accuracy of the its outputs. The code can also be configured to allow direct input by the user to the program for testing, though this can prove quite tedious when inputs are large. The outputs of the network are displayed with their corresponding desired values, and the network is determined to be successfully trained when that output is within a reasonable range around its desired value.

The weight files contain some of the same data as the train files, with an additional list of the number of neurons in each layer. Instead of training data,

input1	input2	output
0	0	0
1	0	1
0	1	1
1	1	0

Table 3.1: XOR

however, these files contain the weights for each synapse in the network. This way, when the network is well-trained, the weight configuration can be saved for later use.

3.2 XOR

We begin using neural networks with the exclusive-or (XOR) problem. The XOR problem is a simple logical situation that takes two inputs and produces one output, pictured in Table 3.1. To put it simply, if exactly one of the inputs is 1 then the output is 1. Otherwise the output is 0.

I used the basic 3 layers (input, hidden, and output) to train this network. As one can see from the chart, it had 2 inputs and produced 1 output, so it simply needed 2 neurons in the input layer and 1 in the output. Determined by trial and error, the most accurate training was achieved when using a hidden layer of 4 neurons, where the output generated by the weights were often slightly above 0 or slightly below 1 to represent outputs of 0 or 1, respectively.

3.2.1 Why XOR?

This type of input to the network was chosen as a starting point due to its logical simplicity and, primarily, its small size. Since the number of inputs given to the neural net affects the overall size of the network, it therefore affects how fast it can complete training. Having only two inputs meant much fewer calculations were necessary and therefore allowed for more time to fix bugs in the code with less time spent running the model.

Of course, there are other kinds of simple starting points that could have been used that also fit the above criteria, but they are not the kind of input that gives

neural nets a real workout. In other words, to have chosen an inclusive-or or "and" problem would have given us an equivalently sized network but would have been too easy for a neural net to solve. The fact that these two other types are linearly separable is the reason for this ease. As we see by the following chart 3.1, we can place a line between all the outputs labeled 0 and those labeled 1. In this case, a neural net could easily solve an inclusive-or problem, but since the XOR problem is not linearly separable, we must introduce the bias neuron [6].

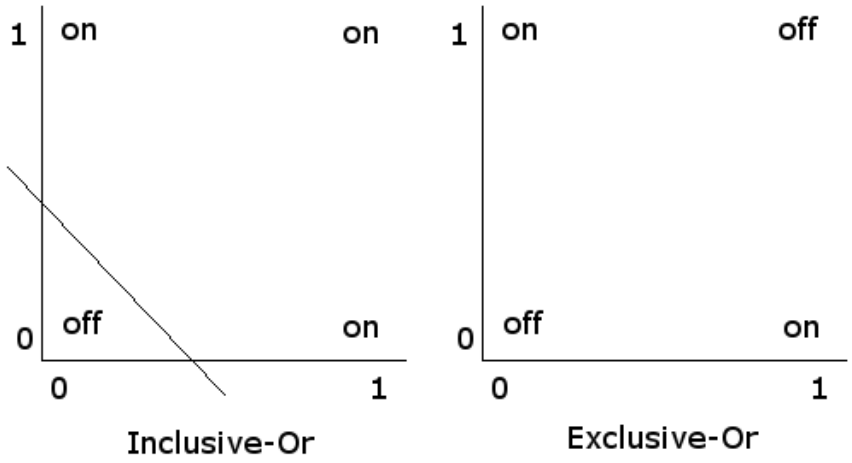

Figure 3.1: Linear Separability [6]

3.2.2 The Bias Neuron's Role

Though easy for a human to understand, the XOR problem can be quite difficult for a neural network to decipher given that it is linearly inseparable. This is where the bias neuron comes into play, serving to offset the otherwise incorrect answers we would receive as output from the network. Essentially, the bias provides the network with power in order to achieve convergence. Without the extra neuron always equal to 1, the first case of the XOR problem would generate exactly 0 each time, as both inputs are 0 and their product from the synapse weights would also be exactly 0.

The sigmoid function can be modified to have an added constant in order to serve the same purpose as the bias neuron [9]. For instance, running the network with the hyperbolic tangent as the sigmoid function and no bias, the network fails when presented with inputs similar to the XOR situation. However, when we instead use the modified hyperbolic tangent

$$\frac{1}{2} + \frac{1}{2} \tanh x$$

without bias, the network converges in just a few hundred epochs.

3.2.3 The Results

As stated earlier, the XOR problem was chosen for its simplicity and size. As such, training the four cases was very quick and convergence at the default minimum error of 0.1 was achieved during every run in usually less than 500 epochs. The network generally produced accurate outputs during testing, which were less than 0.1 for a desired output of 0 and greater than 0.9 for a desired value of 1. Given simplicity of the problem, perhaps the best result we can gather from a network with XOR input is the strong evidence that neural nets can be trained in logical situations.

3.3 Simple Shapes

The next step of this paper is to show that neural nets operate not only on the very simple cases, but also on inputs that are increasingly larger and more complicated. For this section, those inputs will be shapes including squares, triangles, and crosses with value of either 0 or 1.

3.3.1 Structure

The structure of the inputs is that of a grid, or an array when dealing with C code. It is comprised of either 0's or 1's with an inscribed shape of 1's or 0's, respectively. So that the network can learn to identify the shape itself, these shapes may be placed with their center at various points in the grid with or without added noise, which is a decrease in the clarity of the inputs by an alteration of their values. For instance, we could have the following examples of a regular square, a recentered square, and a noisy square, as seen in Table 3.2.

These grids with their inscribed shapes are essentially crude bitmaps. A bitmap, which is a two-dimensional array of pixel values, creates an image, such

0 0 0 0	0 0 0 0	0 0 1 0
0 1 1 0	0 0 1 1	0 1 1 0
0 1 1 0	0 0 1 1	0 1 1 0
0 0 0 0	0 0 0 0	0 0 0 0
Basic	Recentered	Noisy

Table 3.2: 4x4 Squares

0 0 0 0 0
0 0 1 0 0
0 1 1 1 0
1 1 1 1 1
0 0 0 0 0

Table 3.3: 5x5 Triangle

as the triangle in Table 3.3. The grids used here to train the neural network are similar to these bitmaps, in that they too are two-dimensional arrays containing values to create an image. The difference is that the grids used here, which are generally around 4 rows by 4 columns, are considerably smaller than normal bitmaps. The goal then is to show that the neural network can be trained on these grids to identify the shapes contained within, which will show that by merely scaling up the number of inputs we can use the same network to identify shapes. We will not, unfortunately, be identifying shapes contained in bitmaps with standard formatting due to the need for much further coding, which is beyond the scope of this project.

3.3.2 Initial Problems

Moving away from XOR made apparent certain flaws in the code that were initially irrelevant due to the small input size. To achieve convergence, the code originally

multiplied the step size by the global delta in

$$step * \sum delta$$

This approach worked in the two-input XOR problem, but when training on 16 inputs in grids of 4x4, it would not converge. The suspected reason for this is the static step size used to acquire the direction of steepest descent [9], which worked sometimes but at other times caused overshooting of the desired minimum error. In other words, the chosen step size would sometimes cause the error to drop appropriately, but at other times it would change too much. To bring back the example of the hillside, a static step size caused us to continually overstep the bottom of the valley for which we were aiming, which repeated until the maximum epochs were reached. To correct this issue, we needed to use the method of damped steepest descent, so the norm of all delta values was calculated, where

$$norm = \sqrt{\sum (delta^2)}$$

This was then used to divide the original equation, so that

$$synapse = step * (\sum delta)/norm$$

Using the norm created a dynamic step size, which let the network use smaller or larger steps on steeper or gentler descents, respectively.

At this point yet another convergence problem popped up, this one stemming from the sigmoid function. The code makes use of the hyperbolic tangent, pictured in Table 3.2. Again this did not matter in the small XOR problem, but dramatically increasing the input size, especially at and beyond the 4x4 grids, led to a clear reminder that the range of this function is -1 to 1. Since the input and output values are expected to have a range of 0 to 1, the negative outputs produced by the sigmoid function were not supposed to exist. This was fixed by changing the tanh function to

$$\frac{1}{2} + \frac{1}{2} \tanh x$$

resulting in the required range of 0 to 1. As previously noted, the addition step used in this function serves the same purpose as the bias neuron, making it unnecessary. However, since leaving the bias in the network seems to cut down on training time, it was not removed.

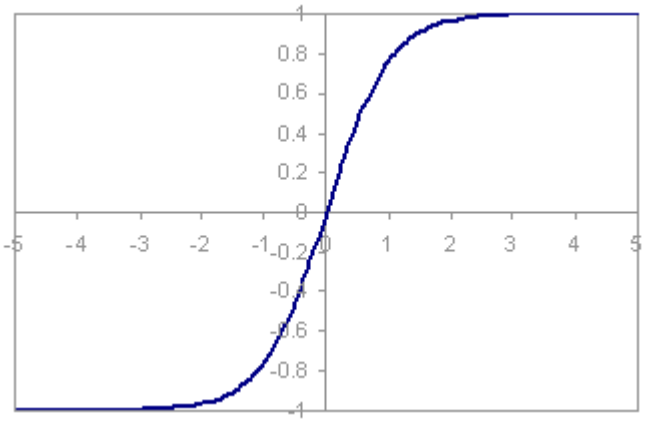

Figure 3.2: Hyperbolic Tangent [7]

3.3.3 Grid Maker

In order to create a large number of input values in a timely manner, it was necessary to create a program that would take a basic input grid with a shape at its center and turn it into several altered versions of that grid. The program I created was `grid_maker.c`. This program allows the user to enter the basic, unaltered version of an input grid, then calculates its center, which is also the shape's center. It places this in the upper left corner of the new grid along with the remaining values contained in the lower right portion of the old grid, then fills in the rest of the values in the new grid according to the designated background values (ie, all the values contained in the grid besides those that are part of the shape). The program then moves the center to one coordinate over and fills in background or shape values appropriately, continuing this until the center of the shape has been in every coordinate in the grid, as seen in Table 3.4. Given that each grid has x rows and y columns, then this program creates $x * y$ input grids.

3.3.4 Training Files and Their Names

I felt it appropriate at this point in the paper to mention the method with which I named the train, test, and weight files. As mentioned earlier, a certain input file's name is reused in both the corresponding test file and resulting weight file. In other words, if we train with 'xor.train', then we will also need a file called 'xor.test'. When these have both run through the program, we have the option of creating 'xor.weight'. The purpose for having all of these use the same name is to avoid confusion when we have multiple input files.

0 0 0 0 0	1 1 0 0 0	1 1 1 0 0
0 0 1 0 0	1 1 1 0 0	1 1 1 1 0
0 1 1 1 0	0 0 0 0 0	0 0 0 0 0
1 1 1 1 1	0 0 0 0 0	0 0 0 0 0
0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
Basic	1st Iteration	2nd Iteration

Table 3.4: Sample Output from Grid Maker

As for the nomenclature itself, I have left it as simple as possible while ensuring the relatively short names for each file. The aforementioned 'xor' input file refers to the XOR network, as mentioned earlier in the paper. When dealing with shapes, the inputs become much more complex, so I used certain letters and numbers to denote the size and type of each input file, where 's', 't', and 'x' denote squares, triangles, and crosses, respectively. For instance, the file '55s.train' is comprised of 5x5 grids with squares only, and the file '44stx.train' contains 4x4 grids with squares, triangles, and crosses. In training sets where we have data with noise, 'n' is appended to the name to represent noisy input values.

3.3.5 Training One to Two Shapes

As mentioned earlier, one of the goals of this project is to show that neural networks are effective in identifying shapes. The following are several training sets, with inputs of one to two shapes, that were run through the network.

We begin with '44s', which finished training by achieving convergence in 622 epochs. Results were excellent, as we see in 3.5.

When using '44sn' for input (see Table 3.6), the training finished in a quick 491 epochs and produced fairly accurate results, despite the added noise. The results are displayed below in Table 3.7.

Training on '55st' took considerably longer, which is understandable given the increased size and complexity. At first, the maximum epochs were set at 6000, which the network reached without convergence. The maximum number of epochs were then gradually increased to 13000, at which point the minimum error was achieved at epoch 12141 and the results produced were excellent. See Table 3.8 for outputs.

Desired	0	1
Input Values	1111	0000
	1001	0110
	1001	0110
	1111	0000
Result	0.012447	0.999062

Table 3.5: 44s.test Output

0 0 0 0	1 1 1 1
0 1 1 0	1 0 1 1
0 1 1 0	1 0 0 1
0 0 0 0	1 1 1 1
Basic	Noisy

Table 3.6: 44sn Inputs

Desired	0	0	1	1
Input Values	1111	1111	0000	0000
	1001	1011	0110	0110
	1001	1001	0110	0100
	1111	1111	0000	0000
Result	0.014253	0.014606	0.998937	0.996156

Table 3.7: 44sn.test Output

Desired	0	0	1	1
Input Values	00000	11111	00000	11111
	00100	11011	01110	10001
	01110	10001	01110	10001
	11111	00000	01110	10001
	00000	11111	00000	11111
Result	0.000107	0.002826	0.990588	0.997922

Table 3.8: 55st.test Output

The outputs produced by the neural net are very accurate for each training set, so much so that they show the network's ability to easily identify the shapes contained in each grid shown to it, which serves to exemplify the neural net's effectiveness in identification. The training was not always easily accomplished, as exemplified by the larger '55st' which required a large number of iterations to achieve decent results. This was likely due to its size, which makes training on each input more complicated, and so we have further evidence that neural nets can identify even when the input increases in complexity.

3.4 Complex Shapes

To fully test the ability of neural net identification, we move further than sets containing only two values and two shapes and start adding diversity in the inputs.

3.4.1 Training with Complex Values

Due to the limited range of possible values in the network being used here (the range being 0 to 1), we must now choose either to use additional output neurons, which will be avoided for this paper due to the extra coding requirements, or introduce a third desired value to the previously used 0 and 1, which will be the value of 0.5. Given that we were previously trying to achieve values on either extreme end of the spectrum, adding a third desired value to the middle of the spectrum makes the task of training the network considerably more difficult. For instance, now we no longer have the luxury of knowing that an output of 0.25 is meant to identify a desired output of 0 during testing.

1 1 1 1	1 1 1 1	.5 .5 .5 .5
1 .5 .5 1	1 0 0 1	.5 1 1 .5
1 .5 .5 1	1 0 0 1	.5 1 1 .5
1 1 1 1	1 1 1 1	.5 .5 .5 .5
Desired: .5	Desired: 0	Desired: 1

Table 3.9: 44sh Inputs

We begin simply, using a training set called '44sh'. This contains 4x4 grids consisting of squares, as we see in Table 3.9. The difference between this training set and the '44s' set is that it contains extra inputs with values of 0.5, as denoted by the 'h' for 'half' in the name. Unlike most earlier sets, the purpose of these inputs is to train the network to identify the different values contained within the one shape instead of identifying the different shapes.

Convergence of this training set proved quite difficult. In one instance, the set was trained all the way to 80000 epochs without convergence, but nearly met the minimum error requirement, finishing with an error of 0.3 and having fairly successful testing results. On the next run, the error converged just past 70000 epochs to produce excellent results for desired values 0 and 1, but only fair results for a desired value of 0.5. Looking at both of these runs, we see the difficulty a neural net faces when training values increase in complexity. Though not optimal for identification, these results maintain a decent degree of accuracy and are certainly useful in identifying values. See Table 3.10.

The results remained much the same after increasing the number of hidden layer neurons to 35, which is more than the default number of 17, a number decided upon because it is an increase of 1 neuron over the size of the input layer. In one run, the network stopped at a predetermined 80000 epochs without convergence, ending with an error of 0.7 but producing very accurate output, as seen in Table 3.11.

Interestingly, if we raise the minimum error to 0.6, then we reach convergence at just over 10000 epochs, but with results of far lower quality than when the error did not converge. This shows that raising the minimum error to achieve convergence is doable if accuracy is not too important, but one must be very careful when adopting this strategy.

These cases provide evidence of the neural net's ability to identify the values of

Desired	0	0	0.5	0.5	1	1
Input Values	1 1 1 1 1 0 0 1 1 0 0 1 1 1 1 1	.5 .5 .5 .5 .5 0 0 .5 .5 0 0 .5 .5 .5 .5 .5	0 0 0 0 0 .5 .5 0 0 .5 .5 0 0 0 0 0	1 1 1 1 1 .5 .5 1 1 .5 .5 1 1 1 1 1	0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0	.5 .5 .5 .5 .5 1 1 .5 .5 1 1 .5 .5 .5 .5 .5
Result	0.000563	0.001341	0.627418	0.643226	1.000000	0.999264

Table 3.10: 44sh.test Output, 70000 Epochs

Desired	0	0	0.5	0.5	1	1
Input Values	1 1 1 1 1 0 0 1 1 0 0 1 1 1 1 1	.5 .5 .5 .5 .5 0 0 .5 .5 0 0 .5 .5 .5 .5 .5	0 0 0 0 0 .5 .5 0 0 .5 .5 0 0 0 0 0	1 1 1 1 1 .5 .5 1 1 .5 .5 1 1 1 1 1	0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0	.5 .5 .5 .5 .5 1 1 .5 .5 1 1 .5 .5 .5 .5 .5
Result	0.001223	0.000004	0.478569	0.474294	0.999998	0.987823

Table 3.11: 44sh.test Output, 80000 Epochs

1 1 1 1	.5 0 0 .5
1 0 0 1	0 .5 .5 0
1 0 0 1	0 .5 .5 0
1 1 1 1	.5 0 0 .5
Square	Cross

Table 3.12: Two Shapes, Three Values

Desired	0	1
Input Values	0 1 1 0	.5 .5 .5 .5
	1 0 0 1	.5 1 1 .5
	1 0 0 1	.5 1 1 .5
	0 1 1 0	.5 .5 .5 .5
Result	0.010112	0.993321

Table 3.13: 44sxh.test Output

inputs despite the added difficulty caused by an increasingly complex set of those values. However, we can note that identifying the different values comprising an identical shape proves to be a far more challenging task than identifying different shapes.

3.4.2 Training with Complex Shapes and Values

In these examples, we will test the network's ability to identify on training sets containing two shapes and three values, like in 3.12.

We begin with '44sxh', which contains combinations of values 0, 0.5, and 1 on a 4x4 grid in two different shapes, a square and a cross. The error converged at just over 7000 epochs with excellent results; Table 3.13 has some examples.

When adjusting the number of hidden neurons to 25, convergence was achieved at epoch 9500 and the output results were all close to the desired values. It would then seem that, in this case at least, a lower count of hidden neurons can be more

Desired	0	1
Input Values	.5 0 0 .4 0 .4 .3 0 0 .5 .6 0 .6 0 0 .5	1 .9 1 .9 .9 .5 .4 1 1 .4 .6 1 .9 1 1 .9
Result	0.009028	0.975398

Table 3.14: 44sxhn.test Output

beneficial to achieving accurate training. This is something to keep in mind when using neural nets, that adding neurons to the hidden layer may benefit training but may also hurt it. Unfortunately, trial and error seems to be the most effective method of finding the optimal size for the hidden layer, which requires several runs through the network for each training set and may therefore be impractical.

Now we move on to '44sxhn', which is identical to the above set except for the added noise. Basically, values are increased or decreased slightly in order to train the network on less than perfect values. The test revealed accurate results from training, during which convergence was reached at above 11000 epochs with 17 hidden neurons. See Table 3.14 for examples. It should be noted that when training the same set with 20 hidden neurons, convergence was achieved at just below 9000 epochs with far more accurate results.

So what we can learn from these tests is that even with greater complexity of values, the neural network still does its job of correctly differentiating between two shapes. And once properly trained, the more complex values have hardly any negative affect on the network's ability to identify.

3.4.3 Training with Three Shapes

Now that we have mixed complex values, we can use those values for identifiers. This allows us the opportunity to move up from only two shapes, with simple identifiers 0 and 1. Unfortunately, since triangles become flawed when placed in 4x4 grids, we are forced to lose our small input sizes and move exclusively to 5x5 grids. This is certainly possible, but prevents rapid training and testing due to the much larger neuron count required for the input layer and, subsequently, the hidden layer. For the first test, we use '55stx', where we are identifying three

0 0 0 0 0	1 1 1 1 1	1 0 0 0 1
0 1 1 1 0	1 1 0 1 1	0 1 0 1 0
0 1 1 1 0	1 0 0 0 1	0 0 1 0 0
0 1 1 1 0	0 0 0 0 0	0 1 0 1 0
0 0 0 0 0	1 1 1 1 1	1 0 0 0 1
Square	Triangle	Cross
Desired: 0	Desired: .5	Desired: 1

Table 3.15: 55stx Inputs

Desired	0	.5	1
Input Values	1 1 1 1 1	1 1 1 1 1	0 1 1 1 0
	1 0 0 0 1	1 1 0 1 1	1 0 1 0 1
	1 0 0 0 1	1 0 0 0 1	1 1 0 1 1
	1 0 0 0 1	0 0 0 0 0	1 0 1 0 1
	1 1 1 1 1	1 1 1 1 1	0 1 1 1 0
Result	0.016689	0.479408	0.988467

Table 3.16: 55stx.test Output

different shapes regardless of the values they hold. Table 3.15 contains examples of inputs.

There was ultimately no convergence achieved during any run, though the test outputs were of decent accuracy when the network had 30 hidden neurons and maxed out the 20000 epochs with error 1.87. This evidence shows that, though not as simple a task as identifying two shapes, identifying three shapes is definitely possible for a neural net. See Table 3.16.

0 0 0 0	.5 .5 .5 .5
0 1 1 0	.5 .25 .25 .5
0 1 1 0	.5 .25 .25 .5
0 0 0 0	.5 .5 .5 .5

Table 3.17: 44sg Inputs

3.5 Grayscale

Grayscale is a type of image used by computers in which each pixel has a certain value. This value determines its coloring, from 0 (white) to 1 (black) with values in between representing shades of gray. Now suppose we have, for instance, a black square on a white background, and this square is composed of pixels with values of 0 set on pixels with values of 1. We have already shown that a neural network can identify shapes and values in this format, and the only difference between the training sets used in this paper and actual grayscale images is the much greater size of the images and far larger set of values, making grayscale images far more complex than that of the simple 4x4 or 5x5 grids used here for testing.

It stands to reason, however, that if a neural network can identify a black square on a white surface, then it can also identify a light gray square on a dark gray surface. And in fact, this has been shown to be the case. When values of 0.5 were introduced, we began to explore using neural nets to identify grayscale images. When identifying different values of identical shapes, we saw that the network can see the difference between black, gray, and white.

3.5.1 Training with Grayscale

For the sake of completeness, we will run another test on more complex grayscale values, which will have the same 0, 0.5, and 1 as before, but also add in 0.25 and 0.75 to the mix. This training set will be denoted '44sg', where 'g' denotes the additional grayscale values. See Table 3.17 for sample inputs. When running the set, the network stopped at 20000 epochs without convergence. Though it had a fairly large error of 1.99, compared to the desired minimum error of 0.1, the network still produced fairly accurate outputs. See Table 3.18 for examples.

We now run '44sgn', which adds noise to the '44sg' training set. The training completes without convergence at an error of approximately 1.8, again maintaining

Desired	0	.25	.5	.75	1
Input Values	1 1 1 1 1 0 0 1 1 0 0 1 1 1 1 1	.5 .5 .5 .5 .5 .25 .25 .5 .5 .25 .25 .5 .5 .5 .5 .5	0 0 0 0 0 .5 .5 0 0 .5 .5 0 0 0 0 0	0 0 0 0 0 .75 .75 0 0 .75 .75 0 0 0 0 0	.5 .5 .5 .5 .5 1 1 .5 .5 1 1 .5 .5 .5 .5 .5
Result	0.084520	0.221771	0.604999	0.863050	0.951484

Table 3.18: 44sg.test Output

Desired	0	.25	.5	.75	1
Input Values	1 1 1 1 1 0 0 1 1 0 0 1 1 1 1 1	1 1 1 1 1 .25 .25 1 1 .25 .25 1 1 1 1 1	1 1 1 1 1 .5 .5 1 1 .5 .5 1 1 1 1 1	1 1 1 1 1 .75 .75 1 1 .75 .75 1 1 1 1 1	0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0
Result	0.054350	0.208707	0.603948	0.849543	0.941079

Table 3.19: 44sgn.test Output

acceptable output values. Again, we see that the neural net is able to succeed with identification. See Table 3.19.

Chapter 4

Conclusion

A neural network is a powerful and widely adaptable method of obtaining solutions to problems that would otherwise require highly specific algorithms for solving; however, difficulties arise in determining the details of the network's configuration. For a multilayer feedforward net, the format of both the inputs and outputs needs to be determined beforehand. Also, the correct size and number of hidden layers must be found to ensure correct training. The sigmoid function must be picked so that the neurons are kept within the proper range and it must be determined if the function will affect the use of the bias neuron. The step size also plays an important role in convergence, so this must either be implemented statically, with the correct value chosen, or be implemented dynamically.

There is also a great need for accurate data and time for training, but these issues cease to be problematic once the network is trained and its weight configuration is saved. This brings to light a helpful feature of neural nets, in that they need to be trained only once to be able to be used indefinitely. But when dealing with problems that can be solved by a traditional algorithm, it may be easier and faster to use this instead of a neural net [9]. Whereas traditional algorithms must be tailored exclusively to a certain problem, a neural net can be built and then implemented solely from the data on which it was trained. This is very useful in cases where the correct answer is known in training, which allows the network to "learn" the data and then solve a wide range of problems.

As we see from the results of several experiments, neural networks are indeed useful and practical in identification. The accompanying C code reveals that a neural net that is shown a series of bitmaps can learn to recognize the shape and the values inscribed in that image. Perhaps the next step beyond this project is to increase the complexity and size of the training data to the point of identifying truly complex images, which would open the door to many practical applications. My hope is that this project has revealed the vast potential of this powerful tool.

Bibliography

[1] Baneer Bar-Yam. *Dynamics of Complex Systems*. Perseus Books, 1997.

[2] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2007.

[3] Laurene Fausett. *Fundamentals of Neural Networks*. Prentice Hall, 1994.

[4] Daniel Franklin. Back-propagation neural network tutorial. URL: `http://ieee.uow.edu.au/~daniel/software/libneural/BPN_tutorial/BPN_English/BPN_English/`, 2003.

[5] Jochen Froehlich. Neural networks with java. URL: `http://fbim.fh-regensburg.de/~saj39122/jfroehl/diplom/e-13-text.html`, 1997.

[6] Timothy Masters. *Practical Neural Network Recipes in C++*. Academic Press, 1993.

[7] Mike O'Neill. Neural network for recognition of handwritten digits. *Code Project*, 2009.

[8] Phil Picton. *Neural Networks*. Palgrave, 2000.

[9] Henry Suters. Conversations and emails, 2010.

Appendix A

Neural Net Code

```
/*
neural_net.c
Andrew Hansen

Code for creation of multilayer feedforward neural networks,
using supervised training.

NOTE:
- All neuron values are in the range 0 to 1.
- Layer 0 is input layer (values must remain unaltered),
    layer data.layer_num - 1 is output layer.
- Training data file format (each value separated by white space):
           - 1st line: train_cases, max epochs, and layer_num
           - 2nd line: rows, columns (grid size for shape)
           - 3rd line: neuron counts for each hidden layer
           - Remaining lines: desired output, then corresponding
                input grid on following lines
- Grids are comprised of a background of values upon which
    different values represent the shape.
- To switch bias on or off, set to 1 or 0 respectively.
*/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
#include<string.h>
```

```
/* Step size for gradient descent */
float step = .07;
/* Minimum error to be achieved during training */
float min_error = .01;
/* Bias neuron, always equal to 1 */
int bias = 1;

/* Hold values for the network's structure */
struct network_data {
    /* Number of layers should be at least 3 to include input,
        hidden, and output */
    int layer_num;
    /* Maximum number of neurons should be equal to the
    largest layer */
    int max_neurons;
    /* Number of training cases for the network */
    int train_cases;
    /* Number of inputs for the network */
    int input_count;
    /* Number of outputs for the network */
    int output_count;
    /* Maximum number of epochs for training */
    int epochs;
    /* NETWORK SIZE, number of neurons in each layer in the
    network */
    int *neuron_count;
    /* Grid size, rows */
    int grid_row;
    /* Grid size, columns */
    int grid_col;
} data;
/* Hold values for the current network */
struct current_network {
    /* 2-dimensional array for neurons in network, number of
    layers by number of neurons, holds each neuron's value */
    float **neurons;
    /* 3-dimensional array for synapses, number of layers by
    number of neurons in left-hand layer by number of neurons
    in right-hand layer, holds each synapse's weight, layer
```

```
    number is left-hand layer */
    float ***synapses;
    /* 2-dimensional array for inputs, number of cases
    by number of inputs */
    float **inputs;
    /* 2-dimensional array for desired outputs used in training,
    number of entries is equivalent to the number of neurons in
    the output layer, train_cases by output_count */
    float **desired;
} net;

/* FUNCTION DECLARATIONS */
void get_train(char file_name[20]);
void get_weight(char file_name[20]);
void weight_init();
void backprop();
float sigmoid(float val);
float sig_prime(float x);
float mse(int curr_case);
void net_print();
void test(char name[13]);
void save_config();

/* MAIN */
int main()
{
    int i, j = 0;
    char file_name[20], ext[7], name[13];

    /* Get input file */
    printf("Please enter training file: ");
    scanf("%s", file_name);

    /* Check if input file is .train or .weight */
    j = strlen(file_name);
    for(i = 0; i < j; i++) {
        if ((file_name[i] == '.') && ((file_name[i+1] == 't')
            || (file_name[i+1] == 'w'))) {
            strcpy(ext, file_name + i);
            j = 1;
```

```
        }
        if (j == 1) {
            strncpy(name, file_name, i);
            name[i] = '\0';
        }
    }
    printf("name %s\n",name);
    printf("ext %s\n\n",ext);

    /* If user inputs .train file, train and test */
    if (strcmp(ext,".train") == 0) {
        printf("Training\n--------\n");
        get_train(file_name);
        weight_init();
        backprop();
        net_print();
        test(name);
        save_config();
    }
    /* If user inputs .weight file, test only */
    else if (strcmp(ext,".weight") == 0) {
        printf("Testing\n-------\n");
        get_weight(file_name);
        net_print();
        test(name);
    }
    else {
        printf("File must be of format .train or .weight
        (with no extra periods).\n");
        exit(1);
    }

    free(data.neuron_count);
    free(net.neurons);
    free(net.synapses);
    free(net.inputs);
    free(net.desired);

    return 0;
}
```

```
/* Get data from file for training */
void get_train(char file_name[20]) {
    FILE *file;
    int i, j, k;

    printf("Getting training data...\n");

    file = fopen(file_name, "r");
    if (file == NULL) {
        printf("Cannot open %s.\n", file_name);
        exit(1);
    }

    /* Get network structure data, line 1 of train file */
    fscanf(file, "%d %d %d", &data.train_cases,
            &data.epochs, &data.layer_num);

    /* Get grid size, line 2 of train file */
    fscanf(file, "%d %d", &data.grid_row, &data.grid_col);

    /* Set input & output counts */
    data.input_count = (data.grid_row * data.grid_col);
    data.output_count = 1;

    /* Allocate and acquire neuron counts for each layer,
     line 3 of train file, find max_neurons */
    data.neuron_count = (int *)
        malloc(data.layer_num * sizeof(int));
    data.neuron_count[0] = data.input_count;
    data.max_neurons = data.neuron_count[0];
    data.neuron_count[data.layer_num - 1] = data.output_count;
    if (data.max_neurons < data.neuron_count[data.layer_num - 1]) {
        data.max_neurons = data.neuron_count[data.layer_num - 1];
    }
    for (i = 1; i < data.layer_num - 1; i++) {
        fscanf(file, "%d", &data.neuron_count[i]);
        if (data.max_neurons < data.neuron_count[i]) {
            data.max_neurons = data.neuron_count[i];
        }
```

```
}
/* Max neurons must be at least 1 more than the highest neuron
count */
data.max_neurons++;

/* Allocate space for all arrays in structs */
net.neurons = (float **) malloc(data.layer_num
    * sizeof(float *));
net.synapses = (float ***) malloc(data.layer_num
    * sizeof(float **));
net.inputs = (float **) malloc((data.train_cases)
    * sizeof(float *));
net.desired = (float **) malloc((data.train_cases)
    * sizeof(float *));
for (i = 0; i < data.layer_num; i++) {
    net.neurons[i] = (float *) malloc((data.max_neurons + 1)
        * sizeof(float));
    net.synapses[i] = (float **) malloc((data.max_neurons + 1)
        * sizeof(float *));
    for (j = 0; j < data.max_neurons; j++) {
        net.synapses[i][j] = (float *)
        malloc((data.max_neurons) * sizeof(float));
    }
}
for (i = 0; i < data.train_cases; i++) {
    net.inputs[i] = (float *) malloc((data.input_count)
        * sizeof(float));
    net.desired[i] = (float *) malloc((data.output_count)
        * sizeof(float));
}

/* Get data, store desired output and input grid */
for (i = 0; i < data.train_cases; i++) {
    fscanf(file,"%f",&net.desired[i][0]);
    for (j = 0; j < data.input_count; j++) {
        fscanf(file, "%f", &net.inputs[i][j]);
    }
}

fclose(file);
```

```
    /* Include bias as last neuron in each layer, besides output */
    for (i = 0; i < data.layer_num - 1; i++) {
        net.neurons[i][data.neuron_count[i]] = bias;
    }

    printf("Acquired training data.\n");
}

void get_weight(char file_name[20]) {
    FILE *file;
    int i, j, k;

    printf("Getting weight configuration...\n");

    file = fopen(file_name, "r");
    if (file == NULL) {
        printf("Cannot open %s.\n", file_name);
        exit(1);
    }

    /* Get network structure data, line 1 of weight file */
    fscanf(file, "%d %d %d", &data.train_cases, &data.epochs,
        &data.layer_num);

    /* Get grid size, line 2 of weight file */
    fscanf(file, "%d %d", &data.grid_row, &data.grid_col);

    /* Set input & output counts */
    data.input_count = (data.grid_row * data.grid_col);
    data.output_count = 1;

    /* Allocate and acquire neuron counts for each layer,
    line 3 of weight file, find max_neurons */
    data.neuron_count = (int *) malloc(data.layer_num
    * sizeof(int));
    data.neuron_count[0] = data.input_count;
    data.max_neurons = data.neuron_count[0];
    data.neuron_count[data.layer_num - 1] = data.output_count;
    if (data.max_neurons < data.neuron_count[data.layer_num - 1]) {
```

```
    data.max_neurons = data.neuron_count[data.layer_num - 1];
}
for (i = 1; i < data.layer_num - 1; i++) {
    fscanf(file, "%d", &data.neuron_count[i]);
    if (data.max_neurons < data.neuron_count[i]) {
        data.max_neurons = data.neuron_count[i];
    }
}
/* Max neurons must be at least 1 more than the highest
neuron count */
data.max_neurons++;

/* Allocate space for all arrays in structs */
net.neurons = (float **) malloc(data.layer_num
    * sizeof(float *));
net.synapses = (float ***) malloc(data.layer_num
    * sizeof(float **));
net.inputs = (float **) malloc((data.train_cases)
    * sizeof(float *));
net.desired = (float **) malloc((data.train_cases)
    * sizeof(float *));
for (i = 0; i < data.layer_num; i++) {
    net.neurons[i] = (float *) malloc((data.max_neurons + 1)
        * sizeof(float));
    net.synapses[i] = (float **) malloc((data.max_neurons + 1)
        * sizeof(float *));
    for (j = 0; j < data.max_neurons; j++) {
        net.synapses[i][j] = (float *)
        malloc((data.max_neurons) * sizeof(float));
    }
}
for (i = 0; i < data.train_cases; i++) {
    net.inputs[i] = (float *) malloc((data.input_count)
        * sizeof(float));
    net.desired[i] = (float *) malloc((data.output_count)
        * sizeof(float));
}

/* Get neuron counts */
for (i = 0; i < data.layer_num; i++) {
```

```
        fscanf(file, "%d ", &data.neuron_count[i]);
    }

    /* Get weights */
    for (i = 0; i < data.layer_num - 1; i++) {
        for (j = 0; j < data.neuron_count[i]; j++) {
            for (k = 0; k < data.neuron_count[i + 1]; k++) {
                /* Read weights, skip identifying integers
                (ie, i and j) */
                fscanf(file, "%*d %*d %f", &net.synapses[i][j][k]);
            }
        }
        /* Read weights connected to bias */
        for (k = 0; k < data.neuron_count[i + 1]; k++) {
            fscanf(file, "%*d %f", &net.synapses[i][j][k]);
        }
    }

    fclose(file);
}

/* Initialize all weights, either to a set or random value */
void weight_init() {
    int i, j, k;
    printf("Initializing weights...\n");
    srand(time(NULL));
    for (i = 0; i < data.layer_num - 1; i++) {
        for (j = 0; j < data.neuron_count[i] + 1; j++) {
            for (k = 0; k < data.neuron_count[i + 1]; k++) {
                net.synapses[i][j][k] = (double)
                (rand() % 10) / 10;
                /*printf("%d,%d, %d: %f\n", i, j, k,
                net.synapses[i][j][k]);for testing */
            }
        }
    }
    printf("Weights initialized.\n");
}

/* Back-propagation function, training the network */
```

```
void backprop() {
    int i, j, k, l, m, ep;
    float val = 0, weight_sum = 0, error[data.train_cases],
        total_error = 0, delta_norm;
    float delta[data.train_cases][data.layer_num]
        [data.max_neurons][data.max_neurons], global_delta
        [data.layer_num][data.max_neurons][data.max_neurons];

    /* Initialize delta values to 0, entries remain unused */
    for (i = 0; i < data.layer_num; i++) {
        for (j = 0; j < data.max_neurons; j++) {
            for (k = 0; k < data.max_neurons; k++) {
                for (m = 0; m < data.train_cases; m++) {
                    delta[m][i][j][k] = 0;
                }
            }
        }
    }

    printf("Begin training...\n");
    /* Epoch loop */
    for (ep = 0; ep < data.epochs; ep++) {
        /* Loop through each training case */
        for (m = 0; m < data.train_cases; m++) {
            /* Place training data in input layer */
            for (k = 0; k < data.neuron_count[0]; k++) {
                net.neurons[0][k] = net.inputs[m][k];
            }
            /* Run through the network */
            /* Layer loop, start with first hidden layer */
            for (i = 1; i < data.layer_num; i++) {
                /* Right neuron loop, the layer of neurons to be
                updated */
                for (j = 0; j < data.neuron_count[i]; j++) {
                    /* Left neuron loop, updates the values of
                    neurons in the next layer */
                    /* neuron[RIGHT_NEURON] += synapse[LEFT_LAYER]
                    [LEFT_NEURON][RIGHT_NEURON]*neuron[LEFT_LAYER]
                    [LEFT_NEURON]; */
                    for (k = 0; k < data.neuron_count[i - 1]
```

```
        + 1; k++) {
            val += net.synapses[i - 1][k][j] *
            net.neurons[i - 1][k];
        }
        net.neurons[i][j] = sigmoid(val);
        val = 0;
    }
}
/* Get mse for each training case, used below to
test convergence */
error[m] = mse(m);

/* Calculate for final hidden to output layer */
i = data.layer_num - 1;
/* Loop over neurons in layer */
for (j = 0; j < data.neuron_count[i]; j++) {
    weight_sum = 0;
    /* Sum all connected synapse weights & neuron
    values in previous layer */
    for (k = 0; k < data.neuron_count[i - 1] + 1;
     k++) {
        weight_sum += net.synapses[i - 1][k][j]
        * net.neurons[i - 1][k];
    }
    /* Get delta for training case */
    for (k = 0; k < data.neuron_count[i - 1] + 1;
     k++) {
        delta[m][i - 1][k][j] = 2 * (net.desired[m][j]
         - net.neurons[i][j]) * sig_prime(weight_sum)
         * net.neurons[i - 1][k];
    }
}
/* Layer loop, from last hidden layer to 2nd layer */
for (i = data.layer_num - 2; i > 0; i--) {
    /* Loop over neurons in right-hand layer */
    for (j = 0; j < data.neuron_count[i]; j++) {
        weight_sum = 0;
        /* Sum all connected synapse weights & neuron
         values */
        for (k = 0; k < data.neuron_count[i - 1] + 1;
```

```
                k++) {
                    weight_sum += net.synapses[i - 1][k][j]
                    * net.neurons[i - 1][k];
                }
                /* Get delta for training case, loop over
                neurons in left-hand layer */
                for (k = 0; k < data.neuron_count[i - 1] + 1;
                 k++) {
                    delta[m][i - 1][k][j] = 0;
                    for (l = 0; l < data.neuron_count[i + 1];
                     l++) {
                        delta[m][i - 1][k][j] +=
                        delta[m][i][j][l]
                        * net.synapses[i][j][l]
                        * sig_prime(weight_sum)
                        * net.neurons[i - 1][k];
                    }
                }
            }
        }
        /* Print delta values
        printf("\nDelta values after last epoch\n");
        for (i = 0; i < data.layer_num - 1; i++) {
            printf("Layer %d to %d:\n", i, i + 1);
            for (j = 0; j < data.neuron_count[i]; j++) {
                for (k = 0; k < data.neuron_count[i + 1];
                 k++) {
                    printf("  %d,%d: %f\n", j, k,
                    delta[m][i][j][k]);
                }
            }
            for (k = 0; k < data.neuron_count[i + 1];
            k++) {
                printf("  bias,%d: %f\n", k,
                 delta[m][i][j][k]);
            }
        }*/
    }

    /* Get global deltas */
```

```
for (i = 0; i < data.layer_num - 1; i++) {
    for (k = 0; k < data.neuron_count[i] + 1; k++) {
        for (j = 0; j < data.neuron_count[i + 1];
         j++) {
            global_delta[i][k][j] = 0;
            for (m = 0; m < data.train_cases; m++) {
                global_delta[i][k][j] +=
                delta[m][i][k][j];
            }
        }
    }
}

/* Get norm of deltas */
delta_norm = 0;
for (i = 0; i < data.layer_num - 1; i++) {
    for (k = 0; k < data.neuron_count[i] + 1; k++) {
        for (j = 0; j < data.neuron_count[i + 1];
         j++) {
            delta_norm += global_delta[i][k][j]
            * global_delta[i][k][j];
        }
    }
}
delta_norm = sqrt(delta_norm);

/* Adjust synapse weights */
for (i = 0; i < data.layer_num - 1; i++) {
    for (k = 0; k < data.neuron_count[i] + 1; k++) {
        for (j = 0; j < data.neuron_count[i + 1];
         j++) {
            net.synapses[i][k][j] += step
            * global_delta[i][k][j] / delta_norm;
        }
    }
}

/* Get the total error for the epoch */
total_error = 0;
for (m = 0; m < data.train_cases; m++) {
```

```
            total_error += error[m];
            /*printf("Error (epoch %d) = %f\n", ep,
            total_error);*/
        }

        /* Check mean squared error */
        /* Stop training if less than minimum error,
        otherwise update weights and move to next epoch */
        if (total_error < min_error) {
            printf("Minimum error (%.2f) reached. Ended
            training at epoch %d.\n", min_error, ep);
            break;
        }
    }
    if (ep >= data.epochs) printf("Maximum epochs (%d) reached,
    training ended with error of %f.\n", ep, total_error);
}

/* Sigmoid update rule */
float sigmoid(float val) {
    return .5*tanh(val)+.5;
}

/* Derivative of the sigmoid function */
float sig_prime (float x) {
    return .5*(1/cosh(x))*(1/cosh(x));/* Equivalent to
    sech(x) squared */
}

/* Mean Squared Error, calculate (actual output - desired output)
 squared for specified case */
float mse(int curr_case) {
    int i, j;
    float error = 0;
    for (i = 0; i < data.neuron_count[data.layer_num - 1]; i++) {
        error += (net.neurons[data.layer_num - 1][i]
        - net.desired[curr_case][i]) * (net.neurons
        [data.layer_num - 1][i] - net.desired[curr_case][i]);
    }
    return error;
```

```
}

/* Print values for network */
void net_print() {
    int i, j, k;
    /* Print training inputs & outputs
    printf("\nTraining Data\n");
    for (i = 0; i < data.train_cases; i++) {
        printf("Set %d:\n", i);
        printf("  Input:\n");
        for (j = 0; j < data.neuron_count[0]; j++) {
            printf("    Neuron %d: %f\n", j, net.inputs[i][j]);
        }
        printf("  Output:\n");
        for (j = 0; j < data.neuron_count[data.layer_num - 1];
        j++) {
            printf("    Neuron %d: %f\n", j, net.desired[i][j]);
        }
    }*/

    /* Print neuron counts for each layer
    printf("\nNeuron Counts\n");
    for (i = 0; i < data.layer_num; i++) {
        printf("Layer %d: %d\n", i, data.neuron_count[i]);
    }*/

    /* Print neuron values
    printf("\nNeuron Values (for last epoch)\n");
    for (i = 0; i < data.layer_num; i++) {
        printf("Layer %d:\n", i);
        for (j = 0; j < data.neuron_count[i]; j++) {
            printf("  %d: %f\n", j, net.neurons[i][j]);
        }
        if (i < data.layer_num - 1) printf("  bias: %f\n",
        net.neurons[i][j]);
    }*/

    /* Print synapse weights
    printf("\nSynapse Weights\n");
    for (i = 0; i < data.layer_num - 1; i++) {
```

```
        printf("Layer %d to %d:\n", i, i + 1);
        for (j = 0; j < data.neuron_count[i]; j++) {
            for (k = 0; k < data.neuron_count[i + 1]; k++) {
                printf("  %d,%d: %f\n", j, k,
                net.synapses[i][j][k]);
            }
        }
        for (k = 0; k < data.neuron_count[i + 1]; k++) {
            printf("  bias,%d: %f\n", k,
            net.synapses[i][j][k]);
        }
    }*/
}

/* Test synapse weights, post-training */
/* Allows the user to manually input or read test
data from file */
void test(char name[13]) {
    int i, j, k = 1, count;
    float val = 0, desired = 0;
    char file_name[20];
    FILE *file;

    strcpy(file_name, name);
    strcpy(file_name + strlen(file_name), ".test");

    file = fopen(file_name, "r");
    if (file == NULL) {
        printf("Cannot open %s.\n", file_name);
        exit(1);
    }
    while (desired >= 0) {
        printf("\nTest Inputs: (q to quit)\n");
        count = 0;

        /* Type inputs
        for (i = 0; i < data.input_count; i++) {
            if ((i % data.grid_col) == 0) {
                printf("  Row %d: ", i /
                data.grid_col);
```

```
            }
            if ((desired = scanf("%f",
            &net.neurons[0][i])) == 0) break;
        }
        if (desired < 0) break;*/

        /* Retrieve inputs from file */
        fscanf(file, "%f", &desired);
        printf("Test %f\n", desired);
        if (desired < 0) break;
        for (i = 0; i < data.input_count; i++) {
            fscanf(file, "%f", &net.neurons[0][i]);
        }

        /* Test and print */
        for (i = 1; i < data.layer_num; i++) {
            for (j = 0; j < data.neuron_count[i]; j++) {
                for (k = 0; k < data.neuron_count[i - 1];
                k++) {
                    val += net.synapses[i - 1][k][j]
                    * net.neurons[i - 1][k];
                }
                val += net.synapses[i - 1][k][j] * bias;
                net.neurons[i][j] = sigmoid(val);
                val = 0;
            }
        }
        printf("Outputs:\n");
        for (i = 0; i < data.neuron_count[data.layer_num - 1];
         i++) {
            printf("  %d: %f\n", i, net.neurons
            [data.layer_num - 1][i]);
        }
    }
    fclose(file);
}

/* Save the weight configuration */
void save_config() {
    char c, file_name[20];/*The file name has maximum of 20
```

```
 characters*/
FILE *file;
int i, j, k;

fflush(stdin);
printf("Would you like to save the weight configuration?
 (y or n)");
scanf("%c", &c);
if ((c == 'y') || (c == 'Y')) {
    printf("File name: ");
    scanf("%s", file_name);
    strcpy(file_name + strlen(file_name), ".weight");
    file = fopen(file_name, "w");
    if (file == NULL) {
        printf("Cannot open %s.\n", file_name);
        exit(1);
    }

    /* Write network structure data */
    fprintf(file, "%d %d %d\n", data.train_cases, data.epochs,
    data.layer_num);
    /* Write grid size, line 2 of input */
    fprintf(file, "%d %d\n", data.grid_row, data.grid_col);
    /* Write max neurons */
    fprintf(file, "%d\n", data.max_neurons);
    /* Write neuron counts */
    for (i = 0; i < data.layer_num; i++) {
        fprintf(file, "%d ", data.neuron_count[i]);
    }
    fprintf(file, "\n");
    /* Write weights */
    for (i = 0; i < data.layer_num - 1; i++) {
        for (j = 0; j < data.neuron_count[i]; j++) {
            for (k = 0; k < data.neuron_count[i + 1]; k++) {
                fprintf(file, "%d %d %f\n", j, k,
                net.synapses[i][j][k]);
            }
        }
        /* Write weights connected to bias */
        for (k = 0; k < data.neuron_count[i + 1]; k++) {
```

```
                fprintf(file, "%d %f\n", k,
                net.synapses[i][j][k]);
            }
        }
        printf("File saved. ");
        fclose(file);
    }
    else printf("Goodbye\n");
}
```

Appendix B

Grid Maker Code

```
/*
grid_maker.c
Andrew Hansen

Creates files of grids for use in neural network code.
*/

#include <cstdio>
#include <cstdlib>
#include <strings.h>
#include <cmath>

int main()
{
    int row, col, i, j, k, l, m, count;
    int row_cen, col_cen, row_even = 0, col_even = 0;
    /* grid holds all values in the given grid
    /* maximum size of 10x10 */
    float grid[10][10], desired, background, temp;
    char file_name[20];
    FILE *file;

    /* Get name of file to be created, grid size, and raw shape */
    printf("Enter name of file to create: ");
    scanf("%s", file_name);
    printf("Enter number of shapes to be entered: ");
    scanf("%d", &count);
```

```
printf("Enter dimensions of grid (rows columns): ");
scanf("%d %d", &row, &col);

/* Open grid file */
file = fopen(file_name, "w");
if (file == NULL) {
    printf("Cannot open %s.\n", file_name);
    exit(1);
}

/* Print metadata to file */
fprintf(file, "%d %d %d\n", (row - row_even)
         * (col - col_even) * count, 20000, 3);
fprintf(file, "%d %d\n", row, col);
fprintf(file, "%d\n", row * col + 1);

for (m = 0; m < count; m++) {
    printf("Shape %d\n", m);
    printf("Enter desired output value: ");
    scanf("%f", &desired);
    printf("Enter background value: ");
    scanf("%f", &background);
    printf("Enter raw shape: \n");
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++) {
            scanf("%f", &grid[i][j]);
        }
    }

    /* Find center of grid (ie, center of row and columns) */
    temp = row;
    if (fmod(temp,2) == 0) {
        row_cen = row/2 - 1;
        row_even = 1;
    }
    else row_cen = row/2;
    temp = col;
    if (fmod(temp,2) == 0) {
        col_cen = col/2 - 1;
        col_even = 1;
```

```
        }
        else col_cen = col/2;

        /* Move center of shape to each point in the grid
        /* Write to file */
        for (l = 0; l < row - row_even; l++) {
            for (k = 0; k < col - col_even; k++) {
                fprintf(file, "%.2f\n", desired);
                for (i = 0; i < row; i++) {
                    for (j = 0; j < col; j++) {
                        if ((row_cen + i - l >= row)
                            || (col_cen + j - k >= col)
                            || (row_cen + i - l < 0)
                            || (col_cen + j - k < 0))
                            fprintf(file, "%.2f ",background);
                        else
                            fprintf(file, "%.2f ",
                                grid[row_cen + i - l]
                                [col_cen + j - k]);
                    }
                    fprintf(file, "\n");
                }
            }
        }
    }

    fclose(file);
}
```

Appendix C

XOR Train and Test Files

```
xor.train
4 10000  3
2 1
4
0
0 0
1
0 1
1
1 0
0
1 1

xor.test
0
0 0
1
0 1
1
1 0
0
1 1
-1
```

Appendix D

44s Train and Test Files

```
44s.train
32 6000 3
4 4
17
1.00
1.00 1.00 0.00 0.00
1.00 1.00 0.00 0.00
0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00
1.00
0.00 1.00 1.00 0.00
0.00 1.00 1.00 0.00
0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00
1.00
0.00 0.00 1.00 1.00
0.00 0.00 1.00 1.00
0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00
1.00
0.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00
1.00 1.00 0.00 0.00
0.00 0.00 0.00 0.00
1.00
0.00 0.00 0.00 0.00
0.00 1.00 1.00 0.00
```

```
0.00 1.00 1.00 0.00
0.00 0.00 0.00 0.00
1.00
0.00 0.00 0.00 0.00
0.00 0.00 1.00 1.00
0.00 0.00 1.00 1.00
0.00 0.00 0.00 0.00
1.00
0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00
1.00 1.00 0.00 0.00
1.00
0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00
0.00 1.00 1.00 0.00
0.00 1.00 1.00 0.00
1.00
0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00
0.00 0.00 1.00 1.00
0.00 0.00 1.00 1.00
0.00
0.00 0.00 1.00 1.00
0.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00
0.00
1.00 0.00 0.00 1.00
1.00 0.00 0.00 1.00
1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00
0.00
1.00 1.00 0.00 0.00
1.00 1.00 0.00 0.00
1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00
```

```
0.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00
1.00 0.00 0.00 1.00
1.00 0.00 0.00 1.00
1.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00
1.00 1.00 0.00 0.00
1.00 1.00 0.00 0.00
1.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00
0.00 0.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00
1.00 0.00 0.00 1.00
1.00 0.00 0.00 1.00
0.00
1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00
1.00 1.00 0.00 0.00
1.00 1.00 0.00 0.00

44s.test
0
1 1 1 1
1 0 0 1
1 0 0 1
1 1 1 1
0
1 1 1 1
1 0 1 1
1 0 0 1
1 1 1 1
1
```

```
0 0 0 0
0 1 1 0
0 1 1 0
0 0 0 0
1
0 0 0 0
0 1 1 0
0 1 0 0
0 0 0 0
-1
```

Appendix E

55stx Train and Test Files

```
55stx.train
150 20000 3
5 5
30
0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
0.00 1.00 1.00 1.00 0.00
0.00 1.00 1.00 1.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
```

```
0.00 0.00 0.00 0.00 0.00
0.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
0.00 1.00 1.00 1.00 0.00
0.00 1.00 1.00 1.00 0.00
0.00 1.00 1.00 1.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
0.00 0.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
```

```
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
0.00 0.00 0.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 1.00 1.00 1.00 0.00
0.00 1.00 1.00 1.00 0.00
0.00 1.00 1.00 1.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
0.00
```

```
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 1.00 1.00 1.00 0.00
0.00 1.00 1.00 1.00 0.00
0.00 1.00 1.00 1.00 0.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 1.00 1.00 1.00 0.00
0.00 1.00 1.00 1.00 0.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 1.00 1.00 1.00
```

```
0.00 0.00 1.00 1.00 1.00
0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
1.00 0.00 0.00 0.00 1.00
1.00 0.00 0.00 0.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
```

```
0.00 0.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
1.00 0.00 0.00 0.00 1.00
1.00 0.00 0.00 0.00 1.00
1.00 0.00 0.00 0.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
```

```
1.00 1.00 1.00 1.00 1.00
1.00 0.00 0.00 0.00 1.00
1.00 0.00 0.00 0.00 1.00
1.00 0.00 0.00 0.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 0.00 0.00 0.00 1.00
1.00 0.00 0.00 0.00 1.00
1.00 0.00 0.00 0.00 1.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
```

```
1.00 1.00 0.00 0.00 0.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 0.00 0.00 0.00 1.00
1.00 0.00 0.00 0.00 1.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
0.50
1.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
```

```
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 1.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
0.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
0.00 0.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
0.00 0.00 0.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
1.00 0.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
0.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 1.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
```

```
0.00 0.00 1.00 0.00 0.00
0.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
0.00 0.00 0.00 1.00 0.00
0.00 0.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
0.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
0.00 0.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 1.00 0.00
0.00 0.00 0.00 0.00 0.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 1.00
```

```
0.00 0.00 0.00 0.00 0.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 0.00 0.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 1.00 1.00 0.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 1.00 1.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 1.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
```

```
0.00 0.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 0.00 0.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 1.00 1.00 1.00 0.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 1.00 1.00 1.00
0.50
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 1.00 1.00
0.50
0.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.50
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 0.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.50
```

```
1.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.50
1.00 1.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.50
1.00 1.00 1.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.50
0.00 1.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.50
1.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 0.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.50
1.00 1.00 0.00 1.00 1.00
1.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.50
1.00 1.00 1.00 0.00 1.00
1.00 1.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
```

```
1.00 1.00 1.00 1.00 1.00
0.50
1.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.50
1.00 1.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 0.00 1.00
1.00 1.00 1.00 1.00 1.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
1.00 1.00 1.00 1.00 1.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
```

```
0.00 1.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.00 0.00 0.00 0.00 1.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 0.00 0.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 0.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00 1.00 0.00 0.00 0.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 1.00
0.00 0.00 1.00 1.00 1.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
0.00 0.00 0.00 1.00 1.00
0.50
```

```
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 0.00 0.00 0.00 1.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 0.00 0.00 0.00
0.50
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 0.00 0.00
1.00
1.00 0.00 0.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00
0.00 1.00 0.00 0.00 0.00
1.00 0.00 1.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00
0.00 0.00 1.00 0.00 0.00
0.00 1.00 0.00 1.00 0.00
1.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 1.00 0.00 1.00
0.00 1.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
```

```
0.00 0.00 0.00 0.00 0.00
1.00
0.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00
0.00 1.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00
1.00 0.00 1.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
1.00 0.00 1.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00
0.00 1.00 0.00 1.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 1.00 0.00 1.00 0.00
1.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 0.00 0.00
1.00
0.00 0.00 1.00 0.00 1.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 1.00 0.00 1.00
0.00 1.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00
0.00 0.00 1.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
```

```
1.00 0.00 0.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 0.00
1.00
0.00 0.00 0.00 1.00 0.00
1.00 0.00 1.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
1.00 0.00 1.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00
1.00
1.00 0.00 0.00 0.00 1.00
0.00 1.00 0.00 1.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 1.00 0.00 1.00 0.00
1.00 0.00 0.00 0.00 1.00
1.00
0.00 1.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 1.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 1.00 0.00 1.00
0.00 1.00 0.00 0.00 0.00
1.00
0.00 0.00 1.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 1.00 0.00 0.00
1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00
1.00 0.00 1.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
1.00 0.00 1.00 0.00 0.00
1.00
```

```
0.00 0.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 1.00
0.00 1.00 0.00 1.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 1.00 0.00 1.00 0.00
1.00
0.00 0.00 0.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 1.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 1.00 0.00 1.00
1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 0.00 0.00 1.00
0.00 0.00 0.00 1.00 0.00
1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 0.00
1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00
1.00 0.00 1.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
1.00 0.00 0.00 0.00 1.00
0.00 1.00 0.00 1.00 0.00
0.00 0.00 1.00 0.00 0.00
1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 1.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 1.00
```

```
0.00 0.00 0.00 1.00 0.00
1.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 1.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00
0.00 0.00 0.00 0.00 1.00
1.00
0.00 1.00 1.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00
1.00 0.00 1.00 1.00 1.00
0.00 1.00 0.00 1.00 1.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00
1.00 1.00 0.00 1.00 1.00
1.00 0.00 1.00 0.00 1.00
0.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 0.00 1.00 0.00
1.00 0.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00
1.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00
1.00 0.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 1.00
```

```
1.00 0.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00
0.00 1.00 0.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
0.00 1.00 0.00 1.00 1.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00
1.00 0.00 1.00 0.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 0.00 1.00 0.00 1.00
0.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 1.00 1.00
1.00
1.00 1.00 0.00 1.00 0.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 0.00 1.00 0.00
1.00 0.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00
1.00 1.00 0.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00
1.00 1.00 1.00 0.00 1.00
0.00 1.00 0.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
0.00 1.00 0.00 1.00 1.00
1.00 1.00 1.00 0.00 1.00
1.00
```

```
0.00 1.00 1.00 1.00 0.00
1.00 0.00 1.00 0.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 0.00 1.00 0.00 1.00
0.00 1.00 1.00 1.00 0.00
1.00
1.00 0.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 0.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 0.00 1.00 0.00
1.00 0.00 1.00 1.00 1.00
1.00
1.00 1.00 0.00 1.00 1.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 0.00 1.00
0.00 1.00 0.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
0.00 1.00 0.00 1.00 1.00
1.00
1.00 1.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 0.00
1.00 0.00 1.00 0.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 0.00 1.00 0.00 1.00
1.00
1.00 1.00 1.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 0.00
1.00 1.00 1.00 0.00 1.00
```

```
1.00 1.00 0.00 1.00 0.00
1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 1.00 1.00 0.00
1.00 1.00 1.00 0.00 1.00
1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 1.00
1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 0.00 1.00
0.00 1.00 0.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
0.00 1.00 1.00 1.00 0.00
1.00 0.00 1.00 0.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 0.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 0.00
1.00 1.00 1.00 0.00 1.00
1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 1.00 1.00 1.00
1.00 1.00 0.00 1.00 1.00
1.00 1.00 1.00 0.00 1.00
1.00 1.00 1.00 1.00 0.00

55stx.test
0
```

```
1 1 1 1 1
1 0 0 0 1
1 0 0 0 1
1 0 0 0 1
1 1 1 1 1
0
0 0 0 0 0
0 1 1 1 0
0 1 1 1 0
0 1 1 1 0
0 0 0 0 0
.5
0 0 0 0 0
0 0 1 0 0
0 1 1 1 0
1 1 1 1 1
0 0 0 0 0
.5
1 1 1 1 1
1 1 0 1 1
1 0 0 0 1
0 0 0 0 0
1 1 1 1 1
1
0 1 1 1 0
1 0 1 0 1
1 1 0 1 1
1 0 1 0 1
0 1 1 1 0
1
1 0 0 0 1
0 1 0 1 0
0 0 1 0 0
0 1 0 1 0
1 0 0 0 1
-1
```