F.C.T. Universidade do Algarve

Departamento de Engenharia Eletrónica e Informática (DEEI)

Gradient descent applied to non-layered neural networks.

Diogo Fonseca

(no. 79858, grp. 3)

Professor(s): José V. Oliveira and Pedro M. S. V. Martins

A report submitted in partial fulfilment of the requirements of
UAlg for the degree of Engenharia Informática in *Artificial Intelligence*

(Problem 3: Perceptrons)

November 24, 2024

Abstract

This is an undergraduate project mainly aiming to showcase an algorithm capable of efficiently perform propagation and backpropagation in any neural network that is not ordered by layers. Mainly graph theory is applied, with basic machine learning algorithms. Some basic machine learning and algebra knowledge are required, as it is not a goal of this report to explain the mechanisms behind backpropagation or why it works, but only to show it's implementation and explain it on a higher level. The algorithm used for backpropagation is the base gradient descent (not the stochastic version), with the mean square error (MSE) as a loss function and the sigmoid as an activation function. The code was devised in a highly modular fashion in such a way that subsequent parallelization, or even a change of loss function or base algorithm is easy to implement.

Contents

Chapter 1

Introduction

1.1 Background

The purpose of this project is mainly to implement the gradient descent algorithm for back-propagation with the delta rule (using MSE as the loss function) and using the sigmoid as an activation function. This is to be done in such a way that it allows the algorithm to work on a non-layered network. Graph theory, and more specifically Kahn's algorithm is used to perform such a task.

Such a model could be modified or used to simulate and train any neural network that doesn't contain any loops. Moreover it should be noted that a model specifically tuned for layered neural networks would be much faster, and it's parallelization potential would be much greater.

1.2 Problem statement

Consider the following two-neuron network.

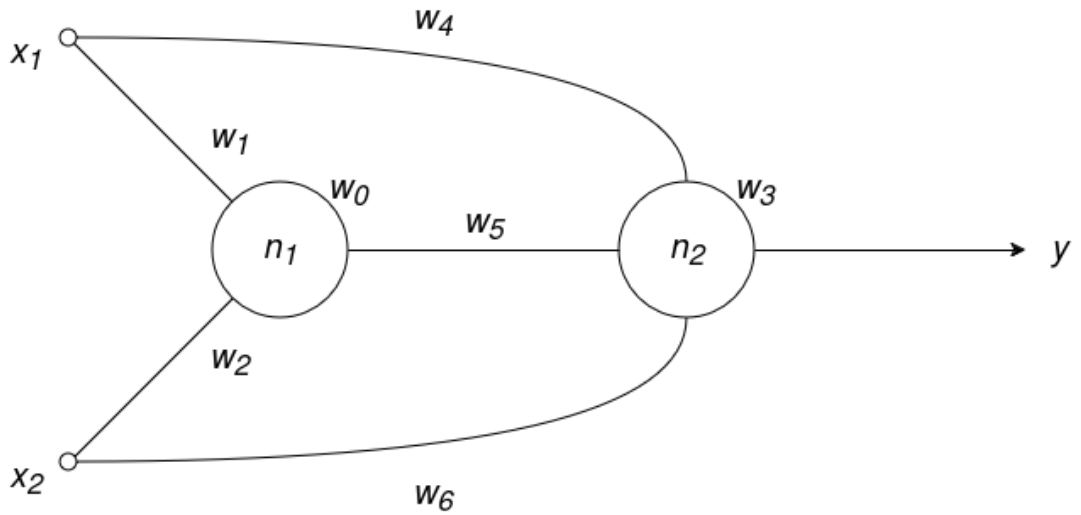

Figure 1.1: Two-neuron non-layered network.

Where:

- n_n is a neuron.
- w_n is the weight of a connection, or the bias if on the top right of a neuron.
- x_n is an input.
- y is the output.

Furthermore,

- s_n is the weighted sum of the n th neuron's inputs.
- $\sigma(z)$ is the sigmoid activation function.
- a_n is the output of neuron n, or $\sigma(s_n)$.

The sigmoid function can be defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{1.1}$$

The main goal is to construct a program that is able to perform backpropagation on any non-layered (or layered) network. The network in 1.1 will be used for demonstration and testing.

There are 2 tasks to be solved as an exercise in machine learning, the third task being the main one. The first and second will be implemented in the model devised in the third.

1. Propose a data set $\{(x_1^{(i)}, x_2^{(i)}, y^{(i)} | i = 1, ..., 4\}$ where x_1 and x_2 are independent binary variables and y is a linearly separable dependent binary variable. Compute without training the weights of a single neuron for obtaining a zero-training error in such a data set.
2. Compute, without training the two-neuron network in 1.1 so that it works as an XOR binary function.
3. Apply the delta-rule to the network in 1.1 using the data set of the XOR function.

1.3 Solution approach

The problem was solved using **Kahn's algorithm** integrated with both propagation and back-propagation, such that a node is only propagated/backpropagated if and only if it has all the necessary inputs.

Chapter 2

Methodology

2.1 Code structure

The code implementation is made up of 5 classes only.

- **Matrix** - this is a class for matrix calculations, it is not generic as it allows only 64bit floating point variables, this was a decision made mainly to avoid constant boxing of the class Double, and also to hyper-specialize the class. This class could be easily swapped by any other framework supporting matrix operations (especially because it is completely synchronous).
- **IPropagable** - interface that represents an object which can propagate forward or backwards. It is specially constructed for machine learning purposes, and in this context it represents a "node" in a neural network, which can be of two types in this implementation.
- **InputNode** - propagable node that mainly acts as to propagate the input.
- **Neuron** - propagable node that represents a neuron.
- **NeuralNetwork** - holds a graph of neurons (not necessarily layered) and has many utilities for easily training neurons.

The main usage is very simple, as can be seen in the implementations in Appendix A. The usual use case consists of making a matrix of the inputs and another of the target outputs, then creating the input nodes while feeding each input node with it's respective row in the matrix. This usually isn't implemented as such, as the input layer matrix can gracefully operate with the following layer, but this isn't the case since the network isn't layered.

Following creating the input nodes, all that's needed is to create the neurons, connect everything, give it to the neural network and train. Besides automation, the NeuralNetwork class acts as abstraction from what is going on underneath.

2.2 Propagation implementation details

2.2.1 Input layer

Starting with the input layer, and it's code representation, InputNodes, all they really do is propagate forward their respective row forward.

2.2.2 Hidden layer(s) and outputs

Because of it's non-layered nature, instead of working layer by layer, we must interpret the entire network as a graph, this comes with it's shortcomings.

Since neurons aren't ordered by layers, our first problem is how to recognize if a given neuron has all the required inputs to be able to do it's math, as the neuron could be reached in code before all of it's inputs nodes have been propagated.

This problem can be easily solved with topological sorting by implementing a version of Khan's algorithm, in which we perform a DFS starting on the input nodes, and every time a node is explored we increment a counter until it reaches the number of backwards connections. At which point we can propagate the given neuron.

This mainly brings problems to matrix operations, as we can't hold matrices for each layer as a neuron can have connections from many different "layers". Analogously we can't hold the weights of a layer, thus having to store both each layer's weights and inputs in itself, which comes with a big performance impact, as not only will we need matrix operations focused on each neuron instead of once per layer, but also because this requires us to make specific matrices for each neuron (as we propagate).

As soon as a neuron has all of it's inputs, all it has to do is propagate the result of the following expression

$$W^T I \tag{2.1}$$

$\boldsymbol{W}$ being the neuron's weights, and $\boldsymbol{I}$ it's inputs. NOTE: each row of I is made of another neuron's propagation matrix. Expanding the matrices for visualization we get

$$\begin{bmatrix} \omega_0 \\ \omega_1 \\ \vdots \\ \omega_n \end{bmatrix}^T \begin{bmatrix} a_0^{(0)} & a_0^{(1)} & \dots & a_0^{(i)} \\ a_1^{(0)} & a_1^{(1)} & \dots & a_1^{(i)} \\ \vdots & \vdots & \ddots & \vdots \\ a_n^{(0)} & a_n^{(1)} & \dots & a_n^{(i)} \end{bmatrix} \tag{2.2}$$

$\boldsymbol{n}$ being the number of backwards connections, and $\boldsymbol{i}$ the number of training cases. The final propagation will then be the output of that neuron.

2.3 Backpropagation implementation details

Applying the delta-rule to a non-layered network comes with the same shortcomings as forward propagation.

In general backpropagation and the delta-rule can be seen as a direct byproduct of the chain rule from differential calculus, derived through dynamic programming. We are chaining (multiplying) different functions of which we know the derivatives of as we explore the nodes. This is also a very specific case of automatic differentiation in which we only have very some specific functions and pattern to differentiate in a graph, and all we have to do is perform a backwards pass.

Exploiting the specificity of automatic differentiation for our use case brings us to the delta-rule. Without further explaining the calculus of gradient calculation and automatic differentiation as a whole, as it is out of the scope of this report, here is explained the process that performs backpropagation in a neural network graph.

First the current node calculates it's associated error/delta, we can call it e. It is the section of the differentiation that all backwards connections as well as the current node's weights will use.

$$e_n^{(i)} = \sigma'(s_n^{(i)}) \sum_{j=1}^{m} e_j^{(i)} \tag{2.3}$$

n being the current node ***m*** the number of forward connections. e for a node then becomes

$$E_n = (\sigma'(S_n)_d E_m^T)^T \tag{2.4}$$

NOTE: do not mistake the E in the equation for the MSE. Furthermore E_n is the delta associated with the current node, while E_m is the deltas of all the forward nodes. Expanding for a better understanding.

$$E_n = \left(\begin{bmatrix} \sigma'(s_n)^{(0)} & 0 & \dots & 0 \\ 0 & \sigma'(s_n)^{(1)} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma'(s_n)^{(i)} \end{bmatrix} \begin{bmatrix} \sum_{j=1}^{m} \omega_j E_j^{(0)} & \sum_{j=1}^{m} \omega_j E_j^{(1)} & \dots & \sum_{j=1}^{m} \omega_j E_j^{(i)} \end{bmatrix}^T \right)^T \tag{2.5}$$

$$= \begin{bmatrix} \sigma'(s_n)^{(0)} \sum_{j=1}^{m} \omega_j E_j^{(0)} \\ \sigma'(s_n)^{(1)} \sum_{j=1}^{m} \omega_j E_j^{(1)} \\ \vdots \\ \sigma'(s_n)^{(i)} \sum_{j=1}^{m} \omega_j E_j^{(i)} \end{bmatrix}^T \tag{2.6}$$

Where ω_j is the weight of a connection and $\boldsymbol{E_j}$ it's delta term. The weights matrix is not included in 2.4 nor in 2.3, this is by design as these are the formulas to be applied by the current node: After calculating E_n it will be propagated backwards, for each backwards connection what will be backpropagated is

$$\omega_j E_n$$

As the receiving node receives $\omega_j E_n$, it will sum what it received to it's already cached E, performing the sum described in 2.5 ($\sum_{j=1}^{m} \omega_j E_j^{(1)}$).

After propagating backwards, the current weights will be updated as such

$$\omega_n = \eta \frac{2}{m} \sum_{i=1}^{m} \delta\omega_n^{(i)} \tag{2.7}$$

Where $\boldsymbol{\eta}$ is the learning rate, ***m*** is the number of training samples and

$$\delta\omega_n^{(i)} = a_n^{(i)} E_n \tag{2.8}$$

Given that $\boldsymbol{a_n^{(i)}}$ is the output of the node of the weight's "input" and $\boldsymbol{E_n}$ is the error/delta term calculated for the current node ("output" of the weight). Note that the "2" is often cut off from this expression, but in this implementation it was decided to keep it.

In matrix form, if a node's weight's updates is to be described as

$$W = W + \Delta W \tag{2.9}$$

And notice how we have a sum and not a subtraction, this comes from the fact that we are using $(t - y)$ for the derivative of the loss function, instead of $(y - t)$. Then

$$\Delta W = \eta \frac{1}{mo} I E^T \tag{2.10}$$

where once again $\boldsymbol{m}$ is the number of training samples and $\boldsymbol{o}$ is the number of output nodes and $\boldsymbol{I}$ is all the inputs of a node. Note that both these constant terms come from the initial step of the chain derivative of the loss function shown later in section 2.4. Expanding these matrices for clarity, we have E^T as described in 2.4 and I as

$$I = \begin{bmatrix} 1 & 1 & \dots & 1 \\ a_b^{(0)} & a_b^{(1)} & \dots & a_b^{(m)} \\ a_{b+1}^{(0)} & a_{b+1}^{(1)} & \dots & a_{b+1}^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{b+c}^{(0)} & a_{b+c}^{(1)} & \dots & a_{b+c}^{(m)} \end{bmatrix} \tag{2.11}$$

In which $\boldsymbol{a_b}$ represents the output of a backward connection's node and $\boldsymbol{c}$ is the number of backwards connections. Note how the first row's inputs are all 1, as the first row corresponds to the bias, and it does not have an input.

2.3.1 Output neurons

Output neurons do not gather the errors/deltas talked about in this section from their forward connections, as they have none. Their E_m term in equation 2.4 comes from the first derivative of the chain derivative process of the loss function, which is

$$E_{m,out} = T - Y \tag{2.12}$$

2.4 Loss function

The loss / error of the network output is given by the MSE, characterized as such

$$E = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{n}\sum_{i=1}^{n}(t_n^{(i)} - y_n^{(i)})^2 \tag{2.13}$$

$\boldsymbol{m}$ being the number of training samples, $\boldsymbol{n}$ the number of outputs, $\boldsymbol{t}$ the target output and $\boldsymbol{y}$ the output of the neuron. A single neuron can then calculate it's own error with the following formula

$$E = \frac{1}{m}\sum_{i=1}^{m}(t^{(i)} - y^{(i)})^2 \tag{2.14}$$

and further simplified with the following matrix expression

$$\frac{1}{m}(T - Y)(T - Y)^T \tag{2.15}$$

2.5 Obtaining the weights of a single neuron for a linearly separable binary function

2.5.1 Problem analysis

The function for this demonstration will be the binary function $y = AND(x_1, x_2)$.

We can plot the truth table in 2.1 and get the following Cartesian representation.

Upon analyzing the above graph 2.1 it is clear the function is linearly separable by the line described by

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.1: AND function truth table.

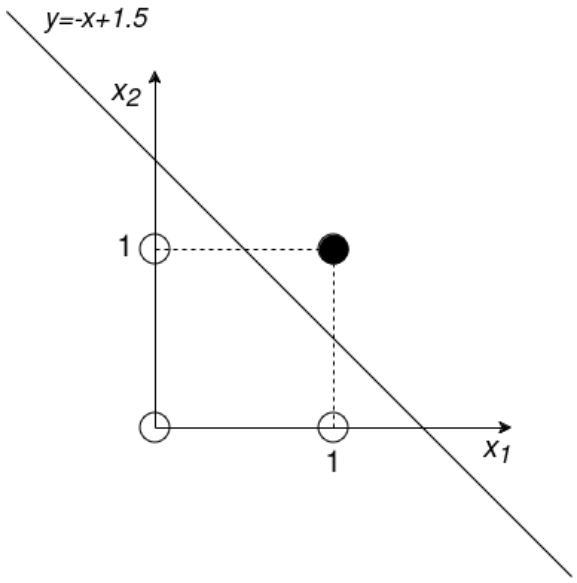

Figure 2.1: Cartesian representation and linear separation of the AND function.

$$x_2 = -x_1 + 1.5 \tag{2.16}$$

this line was chosen in specific because it separates the outputs in such a way that it is perfectly in between the two groups (such was not needed for binary inputs, but as a general good-practice, to account for noise or as to not overfit the line to either output group, making the "fairest" estimation/extrapolation for values outside it's training data/intended input).

2.5.2 Weight deduction

As we are representing a binary function through a neuron working in continuous space, and y can never be 0 or 1 except for when s tends to $\pm\infty$ which never happens computationally, we can interpret y as a binary function such as

$$y = \begin{cases} 1 \text{ when } \sigma(s) \geq 0.5 \\ 0 \text{ when } \sigma(s) < 0.5 \end{cases} \tag{2.17}$$

furthermore, we can simplify the system 2.17 by analysis of the sigmoid function into the following system

$$y = \begin{cases} 1 \text{ when } s \geq 0 \\ 0 \text{ when } s < 0 \end{cases} \tag{2.18}$$

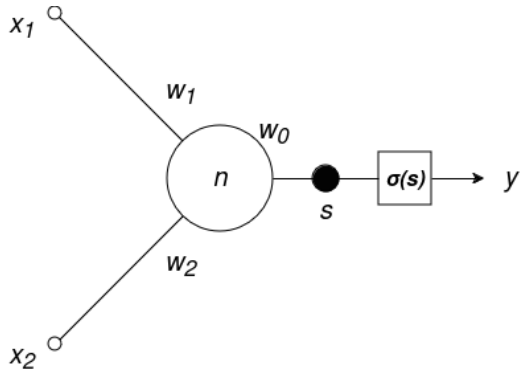

Figure 2.2: Single neuron with two input parameters.

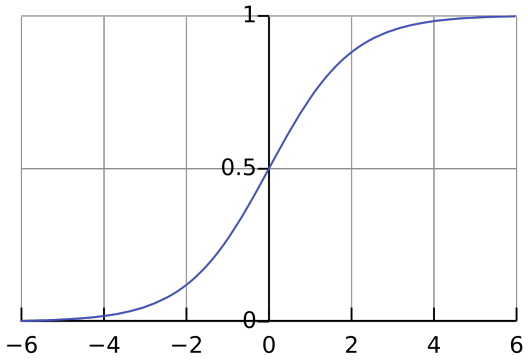

Figure 2.3: Sigmoid function graphed between -6 and 6.

Note that the sigmoid function will be ≥ 0.5 when it's input is ≥ 0 as shown in 2.3. It could then be stated that the decision frontier is $s = 0$.

The value of s in the current setup is defined as

$$s = x_1\omega_1 + x_2\omega_2 + \omega_0 \tag{2.19}$$

which for the decision frontier is

$$0 = x_1\omega_1 + x_2\omega_2 + \omega_0 \tag{2.20}$$

we can then solve for x_2, obtaining

$$x_2 = -\frac{\omega_1}{\omega_2}x_1 - \frac{\omega_0}{\omega_2} \tag{2.21}$$

by comparing 2.21 to 2.16 we get the following system

$$\begin{cases} \frac{\omega_0}{\omega_2} = -1.5 \\ \frac{\omega_1}{\omega_2} = 1 \end{cases} \tag{2.22}$$

There exists an infinity of solutions, but for convenience we can get the solution where $\omega_2 = 1$ which results in the following weights

$$\begin{cases} \omega_0 = -1.5 \\ \omega_1 = 1 \\ \omega_2 = 1 \end{cases} \tag{2.23}$$

2.5.3 Results

x_1	x_2	s	$\sigma(s)$	y
0	0	-1.5	≈ 0.18	0
0	1	-0.5	≈ 0.38	0
1	0	-0.5	≈ 0.38	0
1	1	0.5	≈ 0.62	1

Table 2.2: Resulting outputs from the weights in 2.23.

As evidenced by 2.2, the obtained weights do describe the binary AND function, while separating it equally.

2.5.4 Implementation

the neuron and weights obtained can be expressed in the code implementation shown in A.1. Giving us the following output

```
n1 : -1.5
x1 -> n1 : 1.0
x2 -> n1 : 1.0
(x1, x2) = (n1)
(0.000, 0.000) = (0.182)
(1.000, 0.000) = (0.378)
(0.000, 1.000) = (0.378)
(1.000, 1.000) = (0.622)
```

which we can confirm to match the numbers obtained in 2.2.

2.6 Obtaining the weights of the two-neuron network to act as an XOR

2.6.1 Problem analysis

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.3: XOR function truth table.

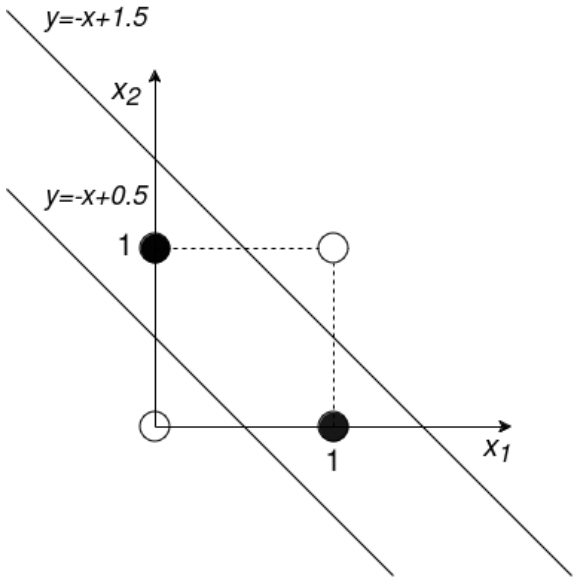

Figure 2.4: Cartesian representation and separation with two lines of the XOR function.

The general strategy is to make the first neuron into a binary function such that it describes one of the two separations needed to make the XOR function. For this demonstration the NAND function will be used, as it describes the separation given by

$$x_2 = -x_1 + 1.5 \tag{2.24}$$

the solution isn't analogous to the demonstration with the AND function because we want a slightly different output as a means to make the XOR function. What we want is for this function to be a true binary operation, that is, it's output should be comprised of either 0 or 1, but such is not possible with the current setup due to the activation function. We can however simulate this behavior by getting mostly either a value very close to 0 or a value very close to 1. We can achieve such a behavior by passing a very high value to the sigmoid function, or a very low value. This is because

$$\sigma(-\infty) = 0 \tag{2.25}$$

$$\sigma(+\infty) = 1 \tag{2.26}$$

for our purposes we can assume 100 and -100 are enough

$$\sigma(-100) \approx 0 \tag{2.27}$$

$$\sigma(+100) \approx 1 \tag{2.28}$$

So then our goal becomes finding weights such that

$$\begin{cases} s_1 \geq 100 \text{ when } NAND(x_1, x_2) = 1 \\ s_1 \leq -100 \text{ when } NAND(x_1, x_2) = 0 \end{cases} \tag{2.29}$$

while still following the separation in 2.24.

2.6.2 Weight deduction for n_1

x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

Table 2.4: NAND function truth table.

by solving analogously to 2.5.2 we get the following system

$$\begin{cases} \frac{\omega_0}{\omega_2} = -1.5 \\ \frac{\omega_1}{\omega_2} = 1 \end{cases} \tag{2.30}$$

Furthermore, we now need to account for getting s_1 to values under/over 100. By defining the inequalities for each binary input, we get the following system of equations

$$\begin{cases} \omega_0 \geq 100 \\ \omega_1 + \omega_0 \geq 100 \\ \omega_2 + \omega_0 \geq 100 \\ \omega_2 + \omega_1 + \omega_0 \leq -100 \end{cases} \tag{2.31}$$

which when combined with the system in 2.30 can be evaluated and simplified to

$$\begin{cases} \omega_0 = -1.5\omega_2 \\ \omega_1 = \omega_2 \\ \omega_2 \leq -200 \end{cases} \tag{2.32}$$

when assigning ω_2 to -200, we then get the final weights of the NAND function for our XOR network.

$$\begin{cases} \omega_0 = 300 \\ \omega_1 = -200 \\ \omega_2 = -200 \end{cases} \tag{2.33}$$

2.6.3 Weight deduction for n_2

Firstly, the equation for s_2 is

$$s_2 = a_1\omega_5 + x_1\omega_4 + x_2\omega_6 + \omega_3 \tag{2.34}$$

Our goal now is to get $s_2 \geq 0.5$ to evaluate it as 1. However for this demonstration it was arbitrarily decided to also make the XOR function into as close to it's binary as possible for the sigmoid, which means it will be solved analogously to it's NAND counterpart, **aiming to output 1.0 or 0.0 exclusively**. This arbitrary rule also makes it easier to demonstrate and understand graphically later.

Once we have n_1 working as a binary NAND, one possible strategy for n_2 is to divide it into it's possible outcomes, where a 0 should equate to a -100 or less, and a 1 should equate to 100 or more, describing it's behavior like so:

When $x_1 = 1 \bigwedge x_2 = 1 \quad (a_1 \approx 0)$

$$\omega_6 + \omega_4 + \omega_3 \leq -100 \tag{2.35}$$

When $x_1 = 0 \bigwedge x_2 = 0 \quad (a_1 \approx 1)$

$$\omega_5 + \omega_3 \leq -100 \tag{2.36}$$

In any other case (except for the area negated by the NAND function) $(a_1 \approx 1)$

$$x_2 \geq -x_1 + 0.5 \tag{2.37}$$

is the line that should describe the activation of this neuron. And so solving analogously to 2.5.2 we get the following system

$$\begin{cases} \omega_6 + \omega_4 + \omega_3 \leq -100 \\ \omega_5 + \omega_3 \leq -100 \\ \frac{\omega_4}{\omega_6} = 1 \\ \frac{\omega_5 + \omega_3}{\omega_6} = -0.5 \end{cases} \tag{2.38}$$

solving for each variable equates to

$$\begin{cases} \omega_3 = -2\omega_6 - 100 \\ \omega_4 = \omega_6 \\ \omega_5 = 2\omega_6 \\ \omega_6 = 200 \end{cases} \tag{2.39}$$

which gives us the following weights for n_2

$$\begin{cases} \omega_3 = -500 \\ \omega_4 = 200 \\ \omega_5 = 400 \\ \omega_6 = 200 \end{cases} \tag{2.40}$$

and so the final weights for this configuration are

$$\begin{cases} \omega_0 = 300 \\ \omega_1 = -200 \\ \omega_2 = -200 \\ \omega_3 = -500 \\ \omega_4 = 200 \\ \omega_5 = 400 \\ \omega_6 = 200 \end{cases} \tag{2.41}$$

It should be noted that this behavior is only possible because we already know the approximated values of a_1 for the described areas, and it's change in those areas is null (it's derivative is ≈ 0), which means the inequalities will hold true for the assumptions of a_1's value on any point. The only places where this doesn't hold true is when the change isn't null, which means on the areas where the sigmoid changes from 0 to 1: in the lines described by $x_2 = -x_1 + 1.5$ and $x_2 = -x_1 + 0.5$.

x_1	x_2	s_1	$\sigma(s_1)$	s_2	y
0	0	300	≈ 1	≈ -100	≈ 0
0	1	100	≈ 1	≈ 100	≈ 1
1	0	100	≈ 1	≈ 100	≈ 1
1	1	-100	≈ 0	≈ -100	≈ 0

Table 2.5: Resulting outputs from the weights in 2.41.

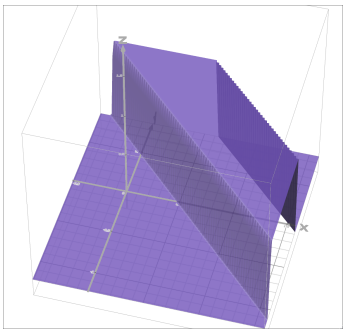

Figure 2.5: a_1 plotted in 3D space, with $x = x_1$, $y = x_2$, $z = a_1$.

2.6.4 Results

When plotting the graph of a_2 in respect to x_1 and x_2, we can see that indeed it has the intended behavior. For bigger images and more perspectives consult B.1

2.6.5 Implementation

the neuron and weights obtained can be expressed in the code implementation shown in A.2. Giving us the following output

```
n1 -> n2 : 400.0
n1 : 300.0
n2 : -500.0
x1 -> n1 : -200.0
x1 -> n2 : 200.0
x2 -> n1 : -200.0
x2 -> n2 : 200.0
(x1, x2) = (n2)
(0.000, 0.000) = (0.000)
(0.000, 1.000) = (1.000)
(1.000, 0.000) = (1.000)
(1.000, 1.000) = (0.000)
```

which we can confirm to match the numbers obtained in 2.5.

2.7 Obtaining the weights through training

For each training data present in Ch. 3 there will be a corresponding entry in appendix A showcasing the code used. All of the code has mostly the same pattern, in order:

1. define training set and target output.
2. define neurons
3. connect neurons
4. train network

Chapter 3

Results and Discussion

3.1 Obtaining the weights of the two-neuron network to act as an XOR through training

The general XOR training implementation is described in appendix A.3. As this model applies the base gradient descent, instead of something like the stochastic gradient descent, it's convergence is quite slow, thus usually needing a large number of iterations to show any results.

3.1.1 Selecting the training rate

Our goal is to minimize the number of iterations to reach a point where the output is < 0.5 when the output should be 0, and ≥ 0.5 when the output should be 1. For this network in specific, it is not of our concern if the network starts overfitting, as we aren't aiming to generalize a binary function. However we should be wary of setting the training rate too high, which could cause it to never converge at all. To get the average number of iterations for convergence, random initial weights were generated between -1 and 1, and an average of 1000 independent trainings were taken as samples for number of iterations performed. The results are in appendix C.1 and the resulting graph lies below.

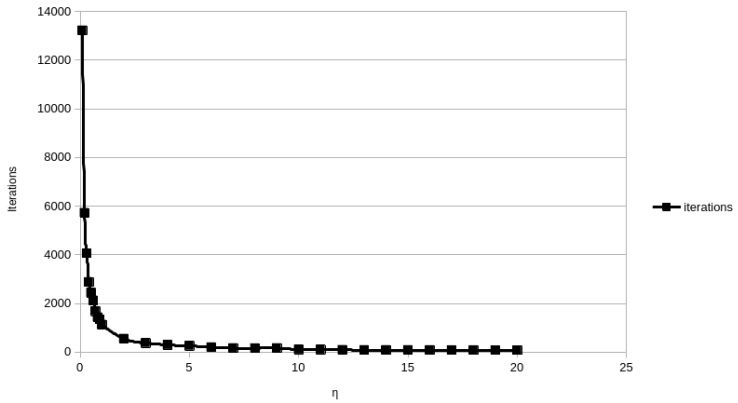

Figure 3.1: Graph of number of average training iterations for convergence per learning rate η.

Looking at the graph and analyzing the data for this specific case, we clearly see a log-

arithmic **improvement** of the number of iterations required as the learning rate goes up (a "$\frac{1}{x}$" behavior). However using a high learning rate can be dangerous as we **risk divergence**, or even increased number of iterations. In our case divergence came first, which is the reason learning rates above 20 were not graphed.

It was observed experimentally that the odds of divergence above 23 are very high, often not converging (especially beginning at 25).

Although divergence under a learning rate of 23 was never observed for this network, a learning rate of 1 will be used to ensure divergence is never an issue, combined with the fact it is the learning rate with most promising results that is ≤ 1.

3.1.2 Selecting the initial weights

Upon experimental observation of the learning rate, it was observed that a particular seed (3207636386306947792) which has the following initial weights:

$$\begin{cases} \omega_0 \approx -0.013 \\ \omega_1 \approx -0.985 \\ \omega_2 \approx -0.882 \\ \omega_3 \approx 0.410 \\ \omega_4 \approx -0.931 \\ \omega_5 \approx -0.899 \\ \omega_6 \approx 0.651 \end{cases} \tag{3.1}$$

seemed to reach a fast convergence. Which with a learning rate of 1.0 reached the following results in 103 iterations:

```
seed: 3207636386306947792
merr: 0.23
rate: 1.0
iterations: 103
n1 -> n2 : -1.6224516894580938
n1 : -0.12976251877949785
n2 : 0.5554389010453477
x1 -> n1 : -1.7514306232617134
x1 -> n2 : -0.3105018225688449
x2 -> n1 : -1.5318349260246862
x2 -> n2 : -0.1982592915969022
(x1, x2) = (n2)
(0.000, 0.000) = (0.449)
(1.000, 0.000) = (0.508)
(0.000, 1.000) = (0.525)
(1.000, 1.000) = (0.499)

error: 0.22979410735774947
```

Giving us the desired output.

3.1.3 Training the network

Firstly our goal is **not** to reach weights such that an output < 0.5 is interpreted as a binary 0, and an output ≥ 0.5 interpreted as a binary 1. Our goal is to reach an approximation of the binary output, just like it was done analytically in section 2.6 (in a way we are overfitting intentionally).

When training the network with the established learning rate (1.0) and the initial weights described in section 3.1.2 we get a very good approximation after 9800 iterations, getting the following outputs (including an iterative demonstration using the NeuralNetwork.iterativePropagation())

```
seed: 3207636386306947792
iter: 9800
rate: 1.0
n1 -> n2 : -14.668082602497808
n1 : 3.101367094120633
n2 : 10.248195502687466
x1 -> n1 : -7.986011299959047
x1 -> n2 : -6.7606625933298945
x2 -> n1 : -7.985263571017975
x2 -> n2 : -6.7605747857540575
(x1, x2) = (n2)
(0.000, 0.000) = (0.022)
(1.000, 0.000) = (0.967)
(0.000, 1.000) = (0.967)
(1.000, 1.000) = (0.037)

error: 0.0010003723788809262
« 0 0
|    n2 » 0.0
« 0 1
|    n2 » 1.0
« 1 0
|    n2 » 1.0
« 1 1
|    n2 » 0.0
« 0.75 0.75
|    n2 » 0.5
```

3.1.4 Result analysis

The resulting weights converted to the weight notation used are

$$\begin{cases} \omega_0 \approx 3.101 \\ \omega_1 \approx -7.986 \\ \omega_2 \approx -7.985 \\ \omega_3 \approx 10.25 \\ \omega_4 \approx -6.761 \\ \omega_5 \approx -14.67 \\ \omega_6 \approx -6.761 \end{cases} \tag{3.2}$$

and when comparing them to the weights computed analytically in section 2.6.3, we can conclude that the main relations still apply, that is

$$\omega_1 \approx \omega_2$$

$$\omega_4 \approx \omega_6$$

Other than that there are some other relations that still apply such as

$$\omega_5 \approx 2\omega_6$$

and $\omega 3$ is of opposite sign as ω_4, ω_5 and ω_6. Just like ω_0 is of opposite sign as ω_1 and ω_2. Disregarding these relations we can see an obvious difference in the magnitude of the values, if given enough training, these values would increase eventually surpassing the magnitude of the ones computed analytically, continuing to respect the relations that make the minimization of the loss function possible (or the mathematical relations that make the XOR function with two neurons rather).

Furthermore we can plot the MSE for each epoch of the network training, allowing us to better understand how much the network is learning over the iterations.

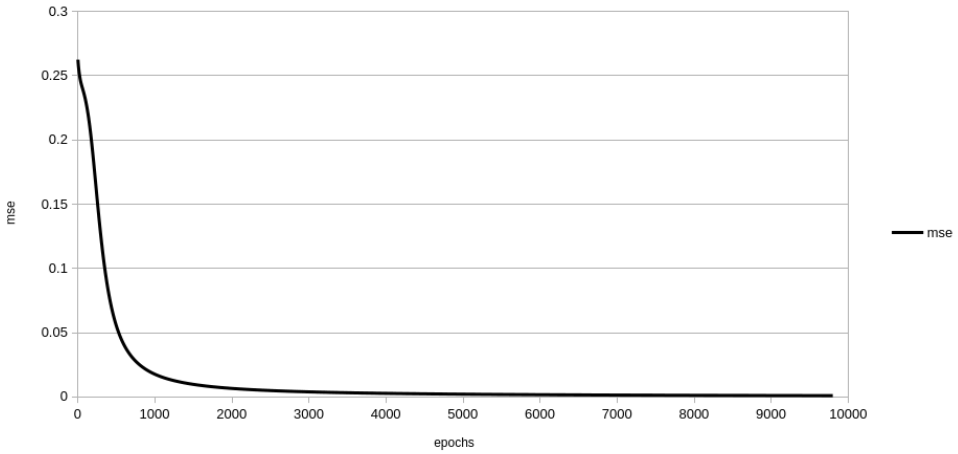

Figure 3.2: Graph of the MSE of each epoch until the two-neuron network approximates the XOR function.

Looking at the graph we can clearly see an asymptote near 0, having diminishing returns as the epochs go towards infinity. Between the first and the 1000th epoch however, there

seems to be a more interesting behavior forming, so when plotting only the first 1000 epochs, we get the following graph

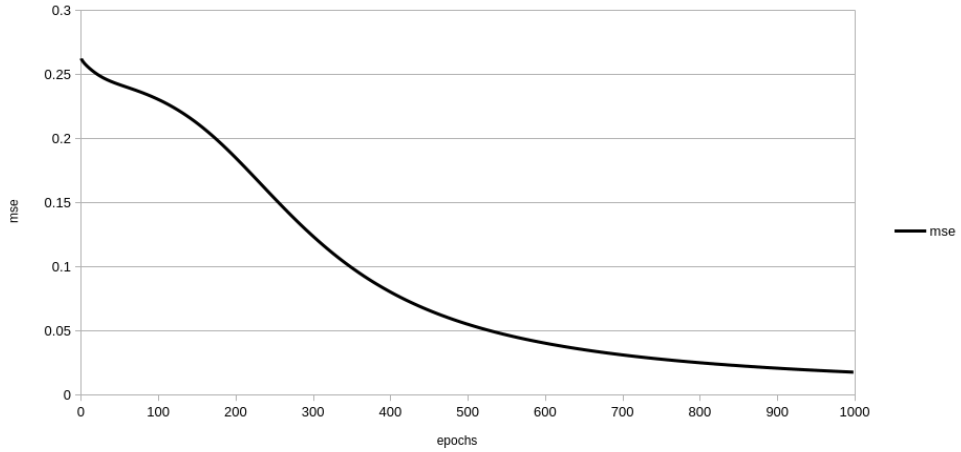

Figure 3.3: Graph of the MSE of each epoch for the first 1000 epochs.

Which is truly an interesting behavior as it seems in the beginning the MSE tends to decrease very slowly, at which point (epoch ∼100) it decreases in a linear fashion. After 300 or so epochs of "linear" decrement (at epoch ∼400) it starts decreasing slowly once again.

3.2 Training a neural network to act as a 2-bit adder

This small demo was made as a demonstration to show the capabilities of the machine learning implementation developed.

The neural network's setup code is shown in appendix A.4 and can be represented graphically as such

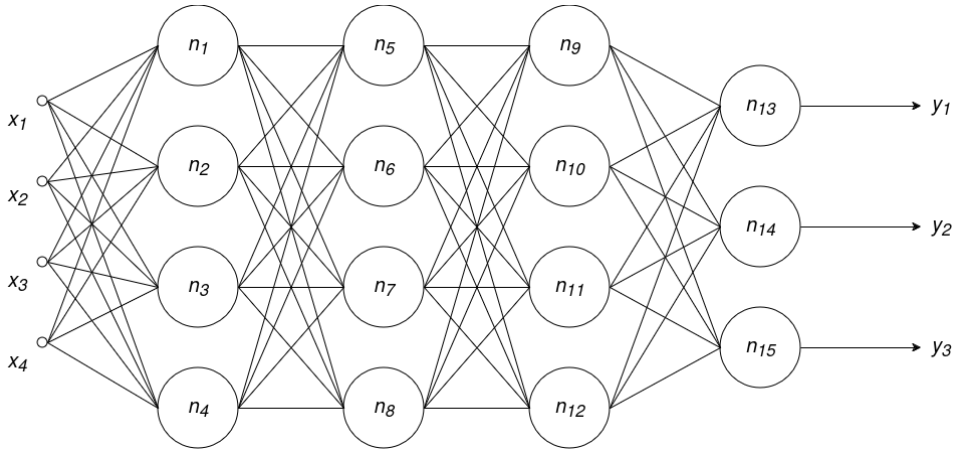

Figure 3.4: Neural network used to train a 2-bit adder.

x_1 and x_2 represent the digits of the first binary number (between 0 and 3), x_3 and x_4 represent the digits of second binary number. y_1, y_2 and y_3 represent the binary output (between 0 and 7). y_3 can be interpreted as the carry bit.

3.2.1 Initial parameters

The number of hidden layers and the number of neurons per hidden layer was a completely arbitrary choice with no backing, as was thought to be more than enough to perform a 2-bit sum. A heuristic for choosing the lower bound of neurons was to think of them as logical operators, and around 4 XOR operations, 4 AND/OR operations would have to be needed to compute the result, so given that we need 2 neurons to compute the result, and the result should propagate from one expression to the next when using logic units, there should be at least 4 layers (2 layers for two XORs in parallel and 2 layers for another 2 XORs in parallel). The learning was rate chosen by experimenting with different values, having good results with $\eta = 0.9$. The number of epochs chosen was 100000. This number comes from experimenting, it's a number that makes the output close to it's binary counterpart (0 or 1), meanwhile a number higher than this makes the network lose an interesting property talked about in section 3.2.3. The initial weight values, when generating at random usually keep all of these properties, but for more deterministic results, the seed used in appendix A.4 can be used.

3.2.2 Result analysis

The resulting output can be found in appendix C.2. Note that the output is backwards, from the least significant digit to the most. We can also verify that all the input-output pairs at the end are correct.

When plotting the MSE over the epochs for this network, we get the following graph which has a similar overall behavior as the plotted MSE for the XOR network in section 3.1.4.

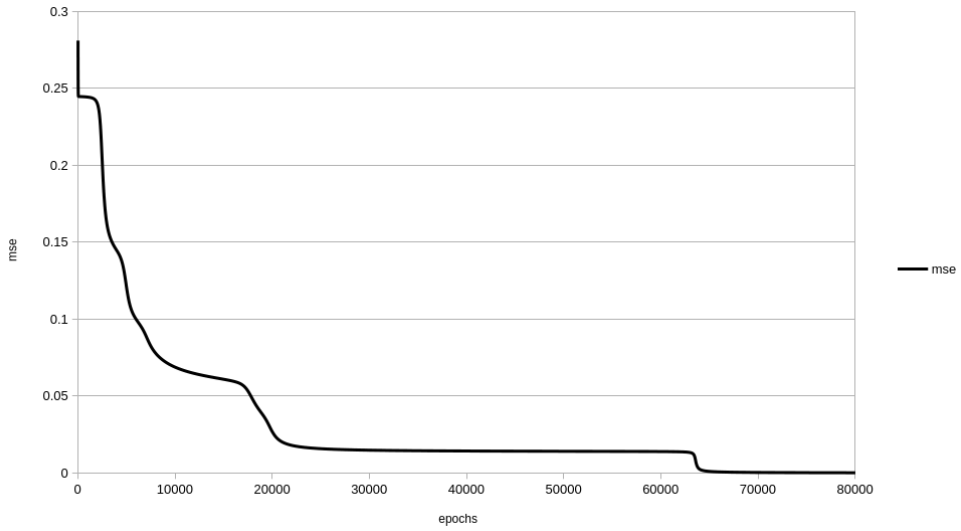

Figure 3.5: MSE over epochs for 2-bit neural network between the first and 80000th epoch.

Furthermore it appears as if it has micro-behaviors similar to the overall behavior described in the analysis of the XOR networking (i.e. from 10000 to 25000, or from 60000 to 70000). It's as if there's points in which the neural networks has a "breakthrough", and rapidly decreases it's error.

3.2.3 Extrapolating values outside of the training range

By far the most interesting thing to note about this neural network, is that it *somehow* is able to perform a sum of two decimal numbers despite not having any training data for such an operation, some samples of input-output pairs of the iterative mode after training showcasing this behavior lie below

```
« 0 2 0 0
|   n13 » 0.0
|   n14 » 1.0
|   n15 » 0.0
```

Showing the network correctly identifying 2 as it's binary counterpart (010).

```
« 0 3 0 0
|   n13 » 1.0
|   n14 » 1.0
|   n15 » 0.0
```

Showing the network correctly identifying 3 as it's binary counterpart (011).

```
« 0 2 0 1
|   n13 » 1.0
|   n14 » 1.0
|   n15 » 0.0
```

Interpreted as $0\times2^1+2\times2^0+0\times2^1+1\times2^0=3$, and the output, $0\times2^2+1\times2^1+1\times2^0=3$. Showing the network correctly summing these numbers.

```
« 2 0 0 0
|   n13 » 1.0
|   n14 » 1.0
|   n15 » 0.0
```

Interpreted as $2\times2^1+0\times2^0+0\times2^1+0\times2^0=4$, and the output, $1\times2^2+0\times2^1+0\times2^0=4$. Showing the network correctly squaring the most significant term.

```
« 2 0 0 1
|   n13 » 1.0
|   n14 » 1.0
|   n15 » 0.0
```

Interpreted as $2\times2^1+0\times2^0+0\times2^1+1\times2^0=5$, and the output, $1\times2^2+0\times2^1+1\times2^0=5$. Showing the network correctly squaring the most significant term and performing the sum.

Again it should be emphasized that the network **did not** have any training samples containing non-binary numbers.

Chapter 4

Conclusions

This project aimed to produce a code base capable of representing non-layered neural networks as well as training them using backpropagation with the delta rule. While showcasing the underlying math. Kahn's algorithm paired with synchronous non-optimized matrix operations proved sufficient for handling efficiently training even a network with a total of 19 nodes for 100000 iterations (showcased in section 3.2 in a relatively short amount of time.

Even with the base unaltered gradient descent, which is a relatively slow machine learning technique, the code implementation was efficient enough to train any small experimental network that was introduced to it.

Besides the math demonstrations and the implementation efficiency, it was possible to analyze the MSE of two networks. It was concluded that the networks go through a process of improvement stagnation, before a big breakthrough after which they will stagnate once again, after one or more of these cycles there will be extreme diminishing gains in continuing training.

For future improvement, parallelizing some aspects of the graph traversal and matrix operations could prove to extremely improve performance, especially considering modern neural network sizes. Performance wise there are matrix operations that could be simplified or cached better. Furthermore, some aspects of the code design could be cleaned up and generalized to allow for a greater modularity, such that changing the loss function or the training algorithm as a whole becomes even easier than it already is.

It should be noted that these optimizations would be for a non-layered network, which isn't a popular choice, and the applications for such a network might be sparse enough not to be worth the performance overhead.

Chapter 5

Reflection

This project was vastly informative to make. A lot of areas of math and computation were researched to make this project from the ground up, such as automatic differentiation, gradient descent, stochastic gradient descent, machine learning theory as a whole, such as neurons and neural networks. Upon learning about all the required subjects all the math was worked from the ground up, and passed to matrix form in such a way to allow for matrix operations of non-layered networks, which in retrospective is somewhat tricky (and not very efficient). Then algorithms were devised to make these operations as efficient as possible.

A way that could help greatly improve performance up to the same level of performance per epoch as a layered network could perhaps be to pre-compute the topological sorting of the network, and adapt the matrix operations in such a way as to take advantage of this, this was ultimately not made a reality in the current implementation as it pushed the difficulty even further, and mainly because it was out of the project's scope.

Overall the experience and knowledge gained with this project, especially in the AI field was *extremely* high, but also in the numeric analysis and slightly in calculus fields.

References

Ari Seff (2020), 'What is automatic differentiation? - video explanation of the paper "automatic differentiation in machine learning: a survey"'. (accessed November 2024).
URL: *https://www.youtube.com/watch?v=wG$_n$F1awSSY*

Atilim G. Baydin Barak A. Pearlmutter Alexey A. Radul Jeffrey M. Siskind (2018), 'Automatic differentiation in machine learning: a survey'. (accessed November 2024).
URL: *https://www.jmlr.org/papers/volume18/17-468/17-468.pdf*

Grant Sanderson (2017), 'Neural networks'. (accessed November 2024).
URL: *https://www.3blue1brown.com/topics/neural-networks*

José V. Oliveira (2024), 'Engenharia informática ualg slides'. (accessed November 2024).
URL: *https://tutoria.ualg.pt/2024/course/view.php?id=2354*

Sebastian Ruder (2017), 'An overview of gradient descent optimization algorithms'. (accessed November 2024).
URL: *https://arxiv.org/pdf/1609.04747*

Wikipedia (2012), 'Delta rule'. (accessed November 2024).
URL: *https://en.wikipedia.org/wiki/Delta$_r$ule*

Wikipedia (2024*a*), 'Automatic differentiation'. (accessed November 2024).
URL: *https://en.wikipedia.org/wiki/Automatic$_d$ifferentiation*

Wikipedia (2024*b*), 'Backpropagation'. (accessed November 2024).
URL: *https://en.wikipedia.org/wiki/Backpropagation*

Wikipedia (2024*c*), 'Gradient descent'. (accessed November 2024).
URL: *https://en.wikipedia.org/wiki/Gradient$_d$escent*

Wikipedia (2024*d*), 'Stochastic gradient descent'. (accessed November 2024).
URL: *https://en.wikipedia.org/wiki/Stochastic$_g$radient$_d$escent*

Appendix A

Code implementations

A.1 AND function

```
Matrix trainingSet = new Matrix(new double[][] {
    { 0, 1, 0, 1 },
    { 0, 0, 1, 1 },
});
Matrix targetOutput = new Matrix(new double[][] {
    { 0, 0, 0, 1 },
});

InputNode x1 = new InputNode(trainingSet.getRow(0));
InputNode x2 = new InputNode(trainingSet.getRow(1));
Neuron n1 = new Neuron(-1.5);

ArrayList<InputNode> inputNodes = new ArrayList<>();
inputNodes.add(x1);
inputNodes.add(x2);
ArrayList<Neuron> outputNeurons = new ArrayList<>();
outputNeurons.add(n1);

NeuralNetwork network = new NeuralNetwork(inputNodes, outputNeurons,
    trainingSet, targetOutput);

x1.connect(n1, 1);
x2.connect(n1, 1);

network.setPrettyPrint(true);

network.propagate();

network.printWeights();
network.printOutputs();
```

A.2 XOR function

```
Matrix trainingSet = new Matrix(new double[][] {
    { 0, 0, 1, 1 },
    { 0, 1, 0, 1 },
});
Matrix targetOutput = new Matrix(new double[][] {
    { 0, 1, 1, 0 },
```

```
});
// setup neurons
InputNode x1 = new InputNode(trainingSet.getRow(0));
InputNode x2 = new InputNode(trainingSet.getRow(1));

Neuron n1 = new Neuron(300.0); // w0
Neuron n2 = new Neuron(-500.0); // w3

ArrayList<InputNode> inputNodes = new ArrayList<>();
inputNodes.add(x1);
inputNodes.add(x2);
ArrayList<Neuron> outputNeurons = new ArrayList<>();
outputNeurons.add(n2);

NeuralNetwork network = new NeuralNetwork(inputNodes, outputNeurons,
    trainingSet, targetOutput);

x1.connect(n1, -200.0); // w1
x1.connect(n2, 200.0); // w4
x2.connect(n1, -200.0); // w2
x2.connect(n2, 200.0); // w6
n1.connect(n2, 400.0); // w5

network.setPrettyPrint(true);

network.propagate();

network.printWeights();
network.printOutputs();
```

A.3 XOR function training

```
int iterations = 55000;
double maxError = 0.23;
// double learningRate = 37.0;
double learningRate = 1.0;

Matrix trainingSet = new Matrix(new double[][] {
    { 0, 1, 0, 1 },
    { 0, 0, 1, 1 },
});
Matrix targetOutput = new Matrix(new double[][] {
    { 0, 1, 1, 0 },
});

InputNode x1 = new InputNode(trainingSet.getRow(0));
InputNode x2 = new InputNode(trainingSet.getRow(1));
Neuron n1 = new Neuron();
Neuron n2 = new Neuron();

ArrayList<InputNode> inputNodes = new ArrayList<>();
inputNodes.add(x1);
inputNodes.add(x2);
ArrayList<Neuron> outputNeurons = new ArrayList<>();
outputNeurons.add(n2);

NeuralNetwork network = new NeuralNetwork(inputNodes, outputNeurons,
    trainingSet, targetOutput);

```

```
// setup connections
x1.connect(n1);
x1.connect(n2);
x2.connect(n1);
x2.connect(n2);
n1.connect(n2);

// network.setPrinting(false);
network.setPrettyPrint(true);
network.setPrintWhileTraining(false);
network.setPrintWeights(true);

// network.train(iterations, learningRate);
network.train(maxError, learningRate);

network.iterativePropagation();
```

A.4 2-bit adder network

```
Neuron.setSeed(-6825448298221666661);

int iterations = 100000;
double maxError = 0.00001;
double learningRate = 0.9;

Matrix trainingSet = new Matrix(new double[][] {
    { 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1 },
    { 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1 },

    { 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1 },
    { 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1 },
});
Matrix targetOutput = new Matrix(new double[][] {
    { 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0 },
    { 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1 },
    { 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1 },
});

InputNode x1 = new InputNode(trainingSet.getRow(0));
InputNode x2 = new InputNode(trainingSet.getRow(1));
InputNode x3 = new InputNode(trainingSet.getRow(2));
InputNode x4 = new InputNode(trainingSet.getRow(3));

// hidden layer 1
Neuron n1 = new Neuron();
Neuron n2 = new Neuron();
Neuron n3 = new Neuron();
Neuron n4 = new Neuron();

// hidden layer 2
Neuron n5 = new Neuron();
Neuron n6 = new Neuron();
Neuron n7 = new Neuron();
Neuron n8 = new Neuron();

// hidden layer 3
Neuron n9 = new Neuron();
Neuron n10 = new Neuron();
Neuron n11 = new Neuron();
```

```
41 Neuron n12 = new Neuron();
42
43 // output layer
44 Neuron n13 = new Neuron();
45 Neuron n14 = new Neuron();
46 Neuron n15 = new Neuron();
47
48 ArrayList<InputNode> inputNodes = new ArrayList<>();
49 inputNodes.add(x1);
50 inputNodes.add(x2);
51 inputNodes.add(x3);
52 inputNodes.add(x4);
53 ArrayList<Neuron> outputNeurons = new ArrayList<>();
54 outputNeurons.add(n13);
55 outputNeurons.add(n14);
56 outputNeurons.add(n15);
57
58 NeuralNetwork network = new NeuralNetwork(inputNodes, outputNeurons,
       trainingSet, targetOutput);
59
60 // input layer -> layer 1
61 x1.connect(n1);
62 x1.connect(n2);
63 x1.connect(n3);
64 x1.connect(n4);
65 x2.connect(n1);
66 x2.connect(n2);
67 x2.connect(n3);
68 x2.connect(n4);
69 x3.connect(n1);
70 x3.connect(n2);
71 x3.connect(n3);
72 x3.connect(n4);
73 x4.connect(n1);
74 x4.connect(n2);
75 x4.connect(n3);
76 x4.connect(n4);
77
78 // hidden layer 1 -> hidden layer 2
79 n1.connect(n5);
80 n1.connect(n6);
81 n1.connect(n7);
82 n1.connect(n8);
83 n2.connect(n5);
84 n2.connect(n6);
85 n2.connect(n7);
86 n2.connect(n8);
87 n3.connect(n5);
88 n3.connect(n6);
89 n3.connect(n7);
90 n3.connect(n8);
91 n4.connect(n5);
92 n4.connect(n6);
93 n4.connect(n7);
94 n4.connect(n8);
95
96 // hidden layer 2 -> hidden layer 3
97 n5.connect(n9);
98 n5.connect(n10);
99 n5.connect(n11);
100 n5.connect(n12);
```

```
101 n6.connect(n9);
102 n6.connect(n10);
103 n6.connect(n11);
104 n6.connect(n12);
105 n7.connect(n9);
106 n7.connect(n10);
107 n7.connect(n11);
108 n7.connect(n12);
109 n8.connect(n9);
110 n8.connect(n10);
111 n8.connect(n11);
112 n8.connect(n12);
113
114 // hidden layer 3 -> output layer
115 n9.connect(n13);
116 n9.connect(n14);
117 n9.connect(n15);
118 n10.connect(n13);
119 n10.connect(n14);
120 n10.connect(n15);
121 n11.connect(n13);
122 n11.connect(n14);
123 n11.connect(n15);
124 n12.connect(n13);
125 n12.connect(n14);
126 n12.connect(n15);
127
128 network.setPrettyPrint(true);
129 network.setPrintWhileTraining(false);
130 network.setPrintWeights(true);
131
132 network.train(iterations, learningRate);
133 // network.train(maxError, learningRate);
134
135 network.iterativePropagation();
```

Appendix B

Graphics

B.1 XOR neuron graphs

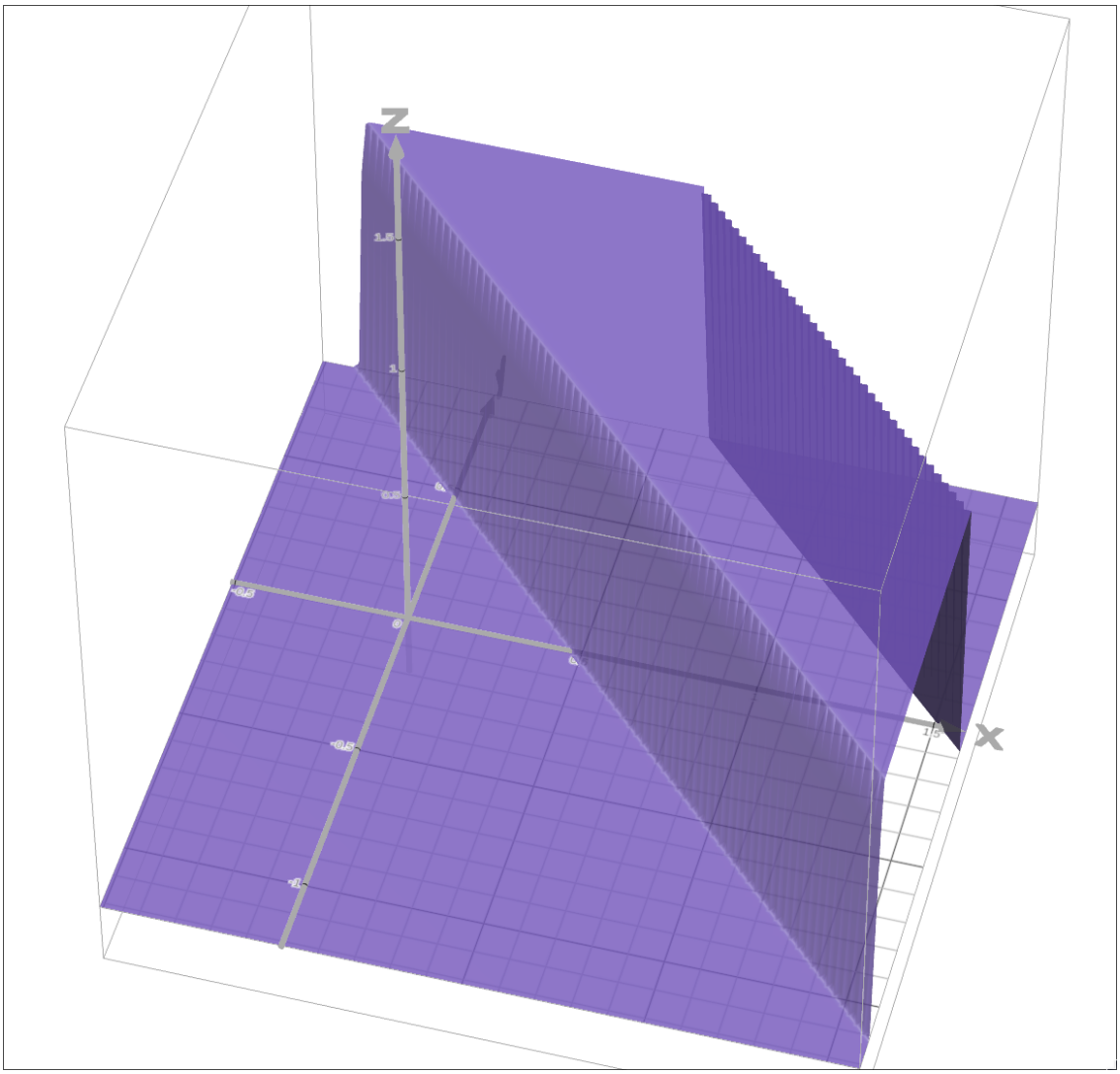

Figure B.1: a_1 plotted in 3D space, with $x = x_1$, $y = x_2$, $z = a_1$.

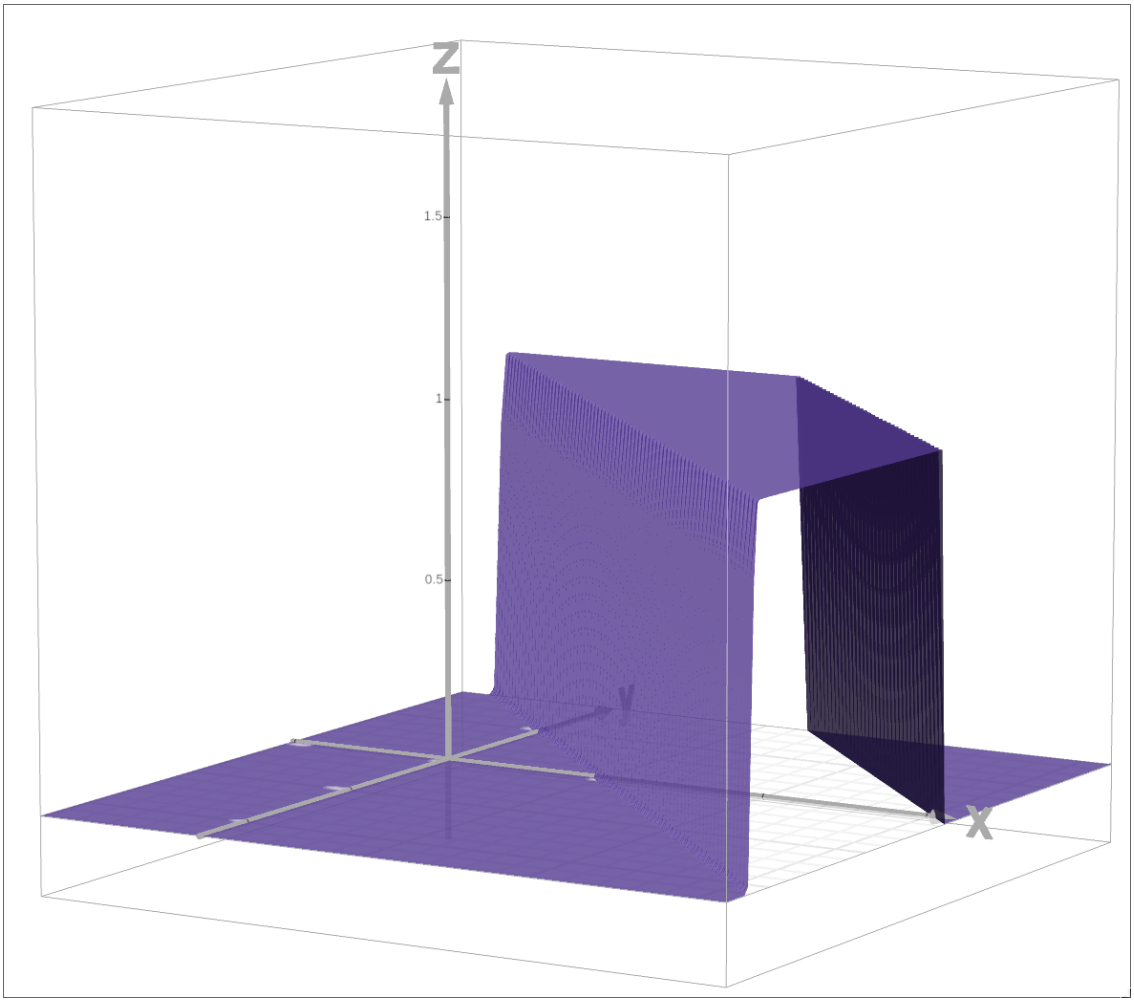

Figure B.2: a_1 plotted in 3D space, with $x = x_1$, $y = x_2$, $z = a_1$.

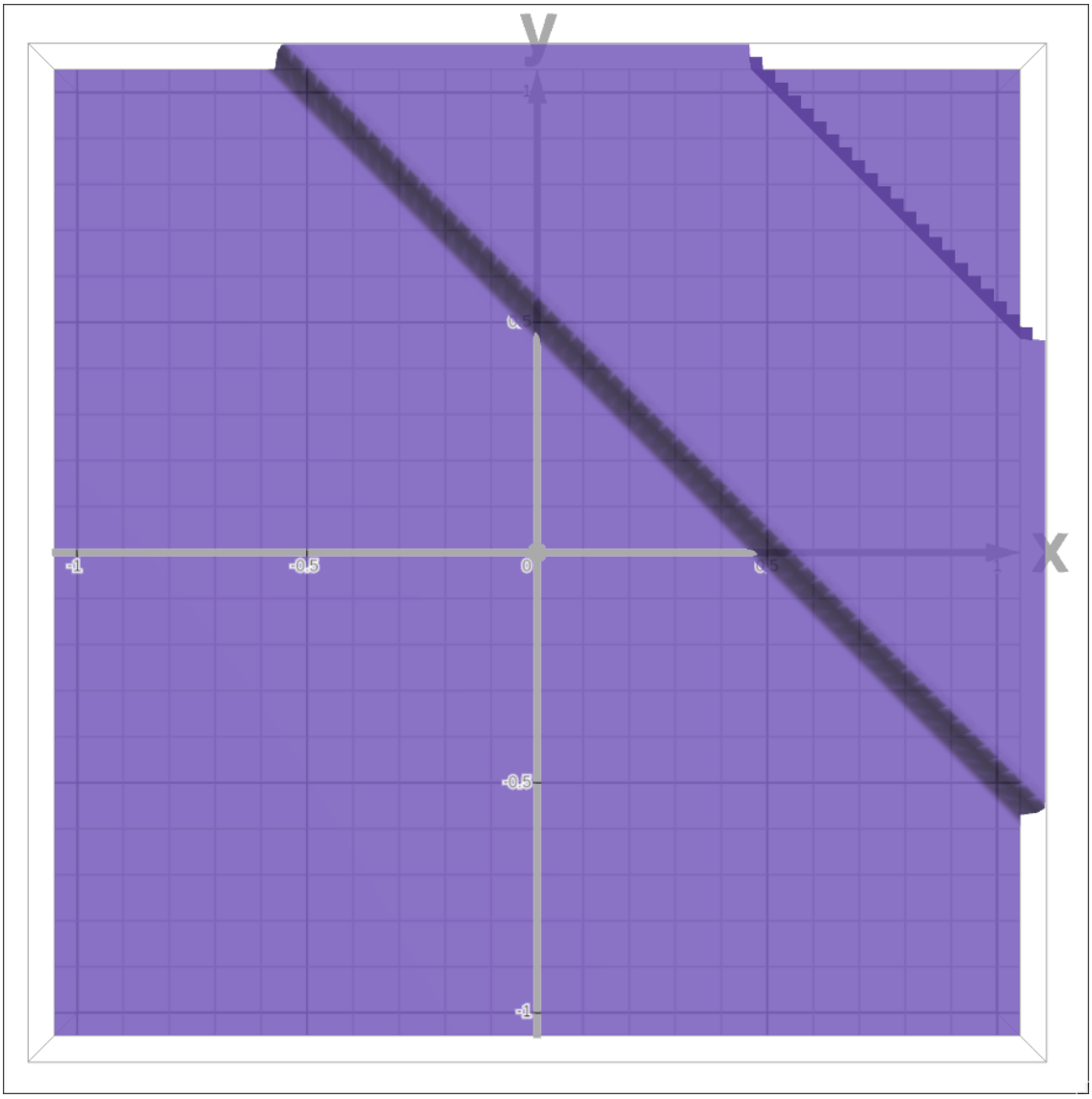

Figure B.3: a_1 plotted in 3D space, with $x = x_1$, $y = x_2$, $z = a_1$.

Appendix C

Data points

C.1 XOR learning rate iterations data

η	iterations
0.1	13218
0.2	5723
0.3	4063
0.4	2881
0.5	2439
0.6	2127
0.7	1685
0.8	1435
0.9	1344
1.0	1132
2.0	554
3.0	377
4.0	297
5.0	262
6.0	204
7.0	169
8.0	164
9.0	168
10.0	107
11.0	103
12.0	94
13.0	90
14.0	90
15.0	91
16.0	89
17.0	86
18.0	86
19.0	82
20.0	82

Table C.1: average number of iterations over a sample of 1000 independent trainings per learning rate η

C.2 2-bit network output after training

seed: -682544829822166666
iter: 100000
rate: 0.9
n1 -> n5 : -2.626163071607487
n1 -> n6 : -1.0918361575453854
n1 -> n7 : 6.890816287316236
n1 -> n8 : 9.049902192119283
n1 : -1.979943068925635
n10 -> n13 : 11.016527064044421
n10 -> n14 : 18.01469423190476
n10 -> n15 : 0.33118819699995283
n10 : 0.08406389109571413
n11 -> n13 : -6.184599779772065
n11 -> n14 : -7.5834250817658155
n11 -> n15 : 5.6648316813903925
n11 : -0.5997674400562134
n12 -> n13 : 13.402366535888799
n12 -> n14 : -6.583324391626635
n12 -> n15 : 0.9114776850602515
n12 : -1.2796539137421505
n13 : -8.2555534459278
n14 : -4.713490296244223
n15 : -0.8336182532253162
n2 -> n5 : -2.9814662806639602
n2 -> n6 : 0.6209949206470863
n2 -> n7 : -3.8564021288761916
n2 -> n8 : -6.774813429162239
n2 : 3.167121543604468
n3 -> n5 : 5.411934912536479
n3 -> n6 : 8.158794774207792
n3 -> n7 : -9.824064406918238
n3 -> n8 : -4.487078607883731
n3 : 9.040924741667324
n4 -> n5 : -3.8583317998792643
n4 -> n6 : -2.0930068356167877
n4 -> n7 : 7.279803514276952
n4 -> n8 : 4.326662320968116
n4 : -5.437137045784667
n5 -> n10 : -6.971803191255802
n5 -> n11 : 1.0192725029825582
n5 -> n12 : -8.505339292566877
n5 -> n9 : 2.7231435147065777
n5 : -0.5823615870964653
n6 -> n10 : -2.10216944299786
n6 -> n11 : -0.1647437866603822
n6 -> n12 : 7.727335188230168
n6 -> n9 : 1.0194310424505557
n6 : 0.9603846715778972

n7 -> n10 : -8.051416082214715
n7 -> n11 : 6.945032460418737
n7 -> n12 : 4.57983454855983
n7 -> n9 : -8.513096139185702
n7 : -0.6069298838907332
n8 -> n10 : 10.85839414396671
n8 -> n11 : -2.751190431566648
n8 -> n12 : -8.47633017109018
n8 -> n9 : -0.8478128835746189
n8 : -1.5562592526268297
n9 -> n13 : 2.8692494414718683
n9 -> n14 : 7.0691765810759115
n9 -> n15 : -6.911581575097018
n9 : 2.4853684073331777
x1 -> n1 : 5.0421480934195735
x1 -> n2 : -3.4366643170970184
x1 -> n3 : -3.297933153804258
x1 -> n4 : 2.9038538550064033
x2 -> n1 : 4.489474496267876
x2 -> n2 : -1.6237573130236391
x2 -> n3 : -2.5046169212709057
x2 -> n4 : 0.8279900806668822
x3 -> n1 : 4.580604106687328
x3 -> n2 : -3.5728361767243437
x3 -> n3 : -3.167680585471552
x3 -> n4 : 3.0853203541345806
x4 -> n1 : 4.49112273085994
x4 -> n2 : -1.6242842525177454
x4 -> n3 : -2.510836178237913
x4 -> n4 : 0.8165272600205128
(x1, x2, x3, x4) = (n13, n14, n15)
(0.000, 0.000, 0.000, 0.000) = (0.017, 0.021, 0.011)
(0.000, 0.000, 0.000, 1.000) = (0.983, 0.001, 0.010)
(0.000, 0.000, 1.000, 0.000) = (0.017, 0.988, 0.001)
(0.000, 0.000, 1.000, 1.000) = (0.984, 1.000, 0.001)
(0.000, 1.000, 0.000, 0.000) = (0.983, 0.001, 0.010)
(0.000, 1.000, 0.000, 1.000) = (0.022, 0.985, 0.002)
(0.000, 1.000, 1.000, 0.000) = (0.984, 1.000, 0.001)
(0.000, 1.000, 1.000, 1.000) = (0.006, 0.004, 0.991)
(1.000, 0.000, 0.000, 0.000) = (0.019, 0.987, 0.002)
(1.000, 0.000, 0.000, 1.000) = (0.985, 1.000, 0.001)
(1.000, 0.000, 1.000, 0.000) = (0.003, 0.003, 0.991)
(1.000, 0.000, 1.000, 1.000) = (0.989, 0.005, 0.996)
(1.000, 1.000, 0.000, 0.000) = (0.985, 1.000, 0.001)
(1.000, 1.000, 0.000, 1.000) = (0.006, 0.005, 0.991)
(1.000, 1.000, 1.000, 0.000) = (0.989, 0.005, 0.996)
(1.000, 1.000, 1.000, 1.000) = (0.022, 0.987, 0.993)

error: 1.1748376639913018E-4