How does Different activation functions affect the performance and accuracy of a neural network

Word Count: 5000 words

Contents

1 Abstract **3**

2 Background knowledge **3**

3 Aim **4**

4 What is machine learning **4**

5 How does neural networks work **4**
5.1 Layers of a neural network 4
5.2 Neurons 6
5.3 Cost function 6
5.4 How does cost function in this paper work 7
5.5 Gradient descent 7
5.6 Backwards propagation 9
5.7 How does backwards propagation work 9

6 Activation functions **11**
6.1 How do activation functions work 11
6.2 Common problems caused by different activation functions 11
6.3 How to compare different activation functions based on their performance and accuracy 12
6.4 ReLU 12
6.5 Logistic function / Sigmoid function 13
6.6 Arctan 13
6.7 Tanh 13

7 Brief explanation of a neural network using a real example **14**

8 Data collected from the experiment **16**
8.1 ReLU 16
8.2 Logistic function 17
8.3 Arctan 18
8.4 Tanh 19

9 Conclusion Applications and Limitations **19**
9.1 limitations 20

10 References **21**

11 Appendix **22**
11.1 ReLU Graphs 22
11.2 Logistic function Graphs 24
11.3 Arctan 26
11.4 Tanh 28

1 Abstract

In this paper the effects of different activation functions on neural networks are argued

2 Background knowledge

1. vanishing gradient problem: vanishing gradient problem is caused by the derivatives of functions going to 0 as the input goes to infinity. In order to teach a neural network we use a back propagation algorithm. In the back-propagation algorithm we use the first derivative of the activation function. This situation doesn't become a problem if the input of the function doesn't reach too high of a value this is the case in shallow networks but if the network is a deep neural network the input value can reach a very low value therefore the value of the derivative goes to 0.

2. Perceptrons: perceptrons can be thought of as little functions or calculators. The perceptrons are found in layers. A perceptron is connected to each and every perceptron in the previous layer. Every connection has a weight associated with it. To calculate the value of a single perceptron we need to multiply the weight-which can be negative of positive- with the value of the perceptrons -in the previous layer- if the value we get is a lower than a set value decided -threshold- the perceptron outputs a 0 and if it is higher than said threshold the perceptron will output a 1. This property of the perceptrons makes them similar to boolean logic since both can only have 2 states either fully open (1) or fully closed (0). This feature -while it eases the calculations limits what can be achieved with perceptrons.

3. Matrix: Matrixes are sets of numbers that are arranged in columns and lines. Each value has the name "element"

4. Vector:Vectors are mathematical objects that both have a value and a direction. They are made up from components. Each component has one direction and the vectorial sum of these components make up our vector

5. Epoch: Epochs are small subsets of the training data. They are used in order to lower the computational power required. Rather than running the learning algorithm for each individual data point we can run it for the average of an epoch thus reducing the number of calculations drastically.

6. Training and Test sample: In neural networks a given data set is usually divided into 2 parts: one for training the network and the other for testing the networks accuracy on a never before seen data group-in order to be unbiased- These sets are called training sample and test sample respectively

7. Hadamard product: Hadamard product is used when we have 2 vectors that are in the same dimensions. The operator multiplies each component of a vector with its equivalent in the other vector and when it does this process for each component the resultant numbers create a vector of same dimension
8. Transpose operator: Transpose operator is a operator that turns a list of values into a vector which's elements are the elements of that set

3 Aim

In this paper I will briefly explain neural networks and how they work then i will explain different types of activation functions that can be used in these neural networks and finally i will compare different activation functions using the same data set so that i can see which activation function is the most effective and efficient

4 What is machine learning

Machine learning is the general term referring to the ways machines(computers can learn). The most common way is neural networks and deep neural networks(these two methods are essential the same but deep neural network utilises more hidden layers so that it can process data which can be overwhelming for a shallow neural network) All of the ways a machine learn depends on the data its fed because of this machine learning is directly connected to mathematics especially statistics since what computers actually do is interpolating based on the available data

5 How does neural networks work

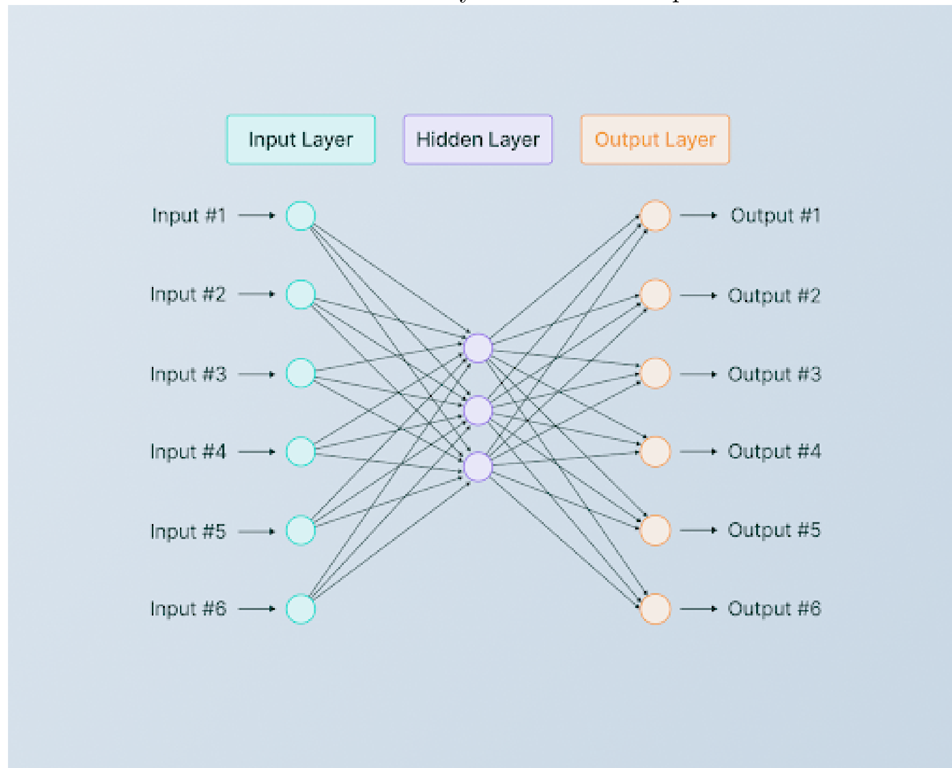
5.1 Layers of a neural network

Neural networks consist of 3 main layers, these are called input layer, hidden layer(s) and output layer. What first and last layer does can be inferred directly from their name

Input layer: input layer is where the data available is fed to the neural network. This data can range from the population of a city to the number of car crashes happening in a junction. The key thing about this layer is that it needs a numerical value and not a word. For some applications words and adjectives can be turned into numerical values for example colour can be represented by relative contributions of red green and blue. This layer can have as many neurons as there are parameters.
Output layer: Output layer is the final layer in the network. This layer tells the output of the neural network to the user. This layer has the same number of

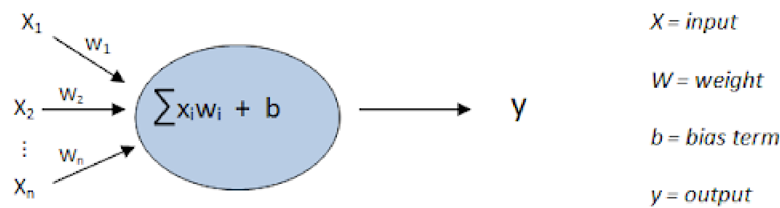
neurons as the number of possible outcomes. This layer also outputs numerical values just like the input layer but in order to get useful data we could associate the neurons to possible outcomes. For example we can say that neuron 1 represents cars and train our network to output a 1 in the first neuron and 0 in all the other neurons when it sees a car. When this network sees a new image which probably has a car in it; it will output a value in near proximity to 1 in the first neuron in the output layer. So what we did was assign a non numerical word to a numerical value

Hidden layer(s): hidden layer is the place where magic happens. Hidden layers are the layers where the calculations and backpropagation algorithm happen. The term hidden comes from the fact that the user does not directly have control over these layers and thus usually does not see these layers. There can be more than one hidden layer in a network. The number of layers depends on the complexity of the problem and computational limitations. Usually neural networks with more than 1-2 hidden layers are called deep neural networks.



5.2 Neurons

Neurons can be taught as little calculators. Each has a connection to the other neurons in the previous layer and a weight associated with them.In order to calculate the value of a certain neuron first we need to multiply the weight of a connection with the value of the neuron corresponding to that connection. Secondly we need to sum all the connection and value multiplications and add bias to that sum. After we calculate this sum (for simplicity this sum will be called z from now on) we need a function to map the numbers we reach(real numbers) to a defined range so that numbers don't get exponentially big or small. For that mapping we use activation functions. Unlike previously discussed perceptrons neurons -with the help of activation functions- can get intermediate values which enable them to be partially open in this sense they are more like the fuzzy logic



$$y = \sum(weight * input) + bias$$

These operations can be done using matrices and vectors
We can write activations of the neurons in l st layer as a^l which is a vector we can also write weights of the neurons in the $l + 1th$ layer as a vector b^{l+1} and when we use dot product to multiply these 2 vectors we get a single value that is equal to z. After we calculate z for every neuron in the $l + 1th$ layer we can create a vector consisting of these values and call it $z^{l+1}$and if we put this vector in our activation function we would get an vector consisting of the activations of neurons in the $l+1th$ layer. We might want to chose this method over the normal way of calculating activations because many neural network optimised programming languages/libraries such as python have very efficient linear algebra libraries that accelerates our neural networks

5.3 Cost function

Cost function is a function that measures how accurate the network was in the last training data. It takes the weights and biases of the network as an input and it uses them to assess whether the value calculated by the neuron is close to the desired value or not. It does this by first assessing the final output of the network and comparing every output neuron's activations to what they should

be(this process is done during the training/learning phase of the network so the output that is desired is known.) When the cost function decides what each neuron should have as its activation then it goes a layer back and repeats the same process

5.4 How does cost function in this paper work

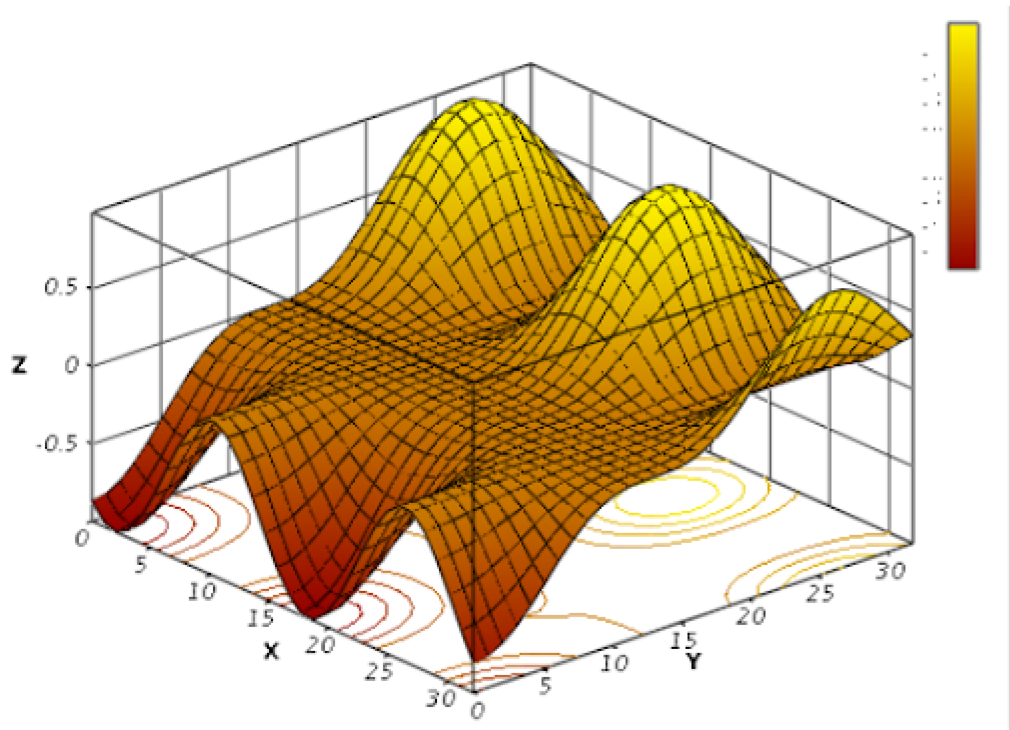
In this paper I have used a quadratic cost function. This function is not a must so other networks created for different purposes can use different cost functions which will have different results compared to this paper.

$$(c(w,b)) \equiv \frac{1}{2n}\Sigma_x|||y(x)-a|||^2$$

In this equation $y(x)$ represents the desired activation and a represents the value our network got represent the length of the vector (as discussed previously all the variables are in the form of vectors since it is easier for the computer to compute) and we square the length of the vector so that we always get a non zero term. This property of our cost function will help us better understand how gradient descent works.

5.5 Gradient descent

To understand gradient descent better we can use a simpler case(our small ie shallow network has 13000 variables rather than imagining a 13000 dimensional plane we would consider a case for which we have 2 inputs(x and y) and a single output (z)which will give us a total of 3 dimensions which we can easily imagine) We can think that there is a ball on our 3 dimensional plane. Common sense and physics tells us that a ball would fall to the lowest place it can reach without exceeding its initial height. In other words it will always go in the $-z$ direction.



3 dimensional x,y,z plane as an example

The movement of our ball based on small change in x and y can be approximately equal to

$$\Delta z \approx \frac{\partial z}{\partial x} \times \Delta x + \frac{\partial z}{\partial y} \times \Delta y$$

This expression can be written in vector form as -which will simplify notation-

$$\Delta z \approx \nabla z \cdot \Delta x$$

Where Δz is the vector which components are the partial derivatives of the cost function and Δx is the vector consisting of small increases to each variable. We want to make Δz negative in order to achieve this goal we must decide a Δx which will always make Δz always negative. A good option is to make $\Delta x = -\alpha \nabla z$ where α is a small positive number which is called learning rate. When we use $-\alpha \nabla$ as Δx we get $\Delta z \equiv -\alpha \nabla z^2$ which is always negative since the square of a number is always non negative and if we multiply it by a negative number we should either get a negative number or 0 so we have achieved our goal. (It is important to say that must be sufficiently small since Δz is approximately equal to the partial derivatives times change in variable. It also must be sufficiently big so that the learning rate of our network is not

too low. In complicated networks this value is changed based on the current gradient vector but in this paper -for simplicity- it is kept constant)

While Δz is said to be dependent on 2 variables it doesn't have to be; all the equations remain same for m number of variables so this method called gradient descent can be used in our neural network with 13000+ variables

The following equation tells us that our network learns by changing the weights and biases. How we change these values are as follows

$$X \rightarrow X' = X - \alpha \nabla C$$

The values we should assign to the weights and biases are old values minus learning rate times partial derivative of the cost function with respect to the variable we are changing

In order to change every weight and bias we must calculate its partial derivative which is a demanding task therefore a method called stochastic gradient descent is used. For this method a big enough(with m variables) "mini-batch" is chosen. The average gradient of this mini batch is approximately equal to the gradient of the whole batch(with n variables) therefore we can use that value instead of calculating the gradient for each weight and bias. This is described in the equation below.

$$\frac{\Sigma_{j=1}^{m} \nabla zj}{m} \approx \frac{\Sigma_{j=1}^{n} \nabla zj}{n}$$

5.6 Backwards propagation

Backwards propagation is a fundamental part of a neural network. As explained neural networks work by forwarding information to layers ahead. This is called feed forward. When the network is in the learning phase it also needs to evaluate the results it got with the actual results. For this phase we use an algorithm called backwards propagation. This algorithm helps us to back propagate the error in the output layer to the hidden layers.

We said that the gradient descent algorithm adjusts our weights and biases. As said this algorithm uses 2 parameters a learning rate $\alpha$and the gradient vector of the cost function ∇C. Back propagation algorithm allows us to calculate the gradient vector.

5.7 How does backwards propagation work

Back propagation is basically calculating partial derivatives of the cost function with respect to weights and biases ($\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$). Calculating these values requires an intermediary value that we call error(this error is not the same thing as the network's whole error. This error is each neuron's ability to affect the

network performance while the other error refers to the percentage of the test images that were incorrectly classified by the network).

We can say that a change in a neuron will affect the network by partial derivative of the cost function with respect to that neuron activation times change in that neuron activation. This is described in the equation below.

$$\frac{\partial C}{\partial z}\Delta z \approx \Delta C$$

From above we can deduce that if the partial derivative of a neuron is big it will have a bigger effect on the cost function and vice versa. Based on this we can call this partial derivative the error of that neuron (δ)

$$\delta \equiv \frac{\partial C}{\partial z}$$

Back propagation will allow us to compute δ for each layer and relate those to $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$. For this purpose we have a total of 4 equations. For descriptive purposes we assume that the sigmoid function is used as the activation function. This is not a requirement since none of the sigmoid specific features of the function is used. So where sigmoid σ function and its derivative $\sigma^{'}$ any other function can be used.

First equation: Calculating error in the (output) final layer:

$$\sigma = \frac{\partial C}{\partial a}\sigma^{'}(Z)$$

a is the activation of the function

σ is the error

$\sigma^{'}(Z)$is the derivative of the activation function evaluated at the neurons value

This equation uses one of the basic properties of calculus: the chain rule. We defined δas the partial derivative of the cost function with respect to the value of that neuron. But the neuron's value doesn't directly affect the cost function; rather it affects the activation of the neuron(via sigmoid function) and that affects the cost function so we multiply their derivatives in order to get the effect of the value of the neuron in the cost function. Essentially what we do is

$$\frac{\partial C}{\partial a}\frac{\partial a}{\partial z} = \frac{\partial C}{\partial Z}$$

Calculating the error in the last layer is not demanding since the network has already calculated z while processing the image. Also the partial derivative of the cost function can be easily computed While it dependents on the activation function calculating its derivative is not -comparatively- that much work

This equation is using the derivative of the activation function. This can lead to the problem called the "vanishing gradient problem". Further discussion about this can be found at the section: Common problems caused by different activation functions.

his equation can be written using vectors in the form
$\delta = \nabla C \odot \sigma'(Z)$
Second equation: Calculating error in the nth layer using the error in the $n+1th$ layer

$$\delta^n = ((w^{n+1})\delta^{n+1} \odot \sigma'(Z)$$
$$w^{n+1} \text{ is the weight matrix}$$
$$\delta^{n+1} \text{ error of the } n+1th \text{ layer}$$
$$\sigma'(Z) \text{ derivative of the activation function calculated at } Z$$

In this equation we are propagating the error we found in the nth layer to the $n-1th$ layer hence the name backwards propagation. This equation first takes the error calculated at the $n+1th$ layer as its input it also take the weights for the $n+1th$ layer this equation applies the weight matrix to the errors and after that it takes the Hadamard product of the vector calculated and the derivative of the activation function calculated at Z. This gives us the error in the nth layer

As we can now find out the error in the last layer and we can propagate the error in any. We can now calculate the error in every layer by first calculating the error in the output layer and then calculating the error in the layer before that using the second equation and doing this process of back propagation until we reach the first layer.

While the other 2 equations are just as important as the first and second one. Since the scope of this paper is activation functions and the first 2 equations concerts activation functions (and also because of word limit issues) other 2 will not be discuss

6 Activation functions

6.1 How do activation functions work

Activation functions are functions that map the output calculated value of a neuron(z) to a real number. They usually have a range of [-1,1] or [0,1] but some activation functions have ranges extending to $\pm$ infinity. We use these functions to standardise our output and prevent our numbers going to infinity in a deep neural network.

6.2 Common problems caused by different activation functions

Different activation functions have different properties. In a neural network we need to calculate the derivative of the activation function at a lot of different values. Because of this some activation functions use a linear activation value for positive numbers and a negative value for negative numbers. While this

makes calculating the derivative trivial since it is a constant it also means that the numbers exponentially get larger as the network gets deeper. On the other hand of the spectrum we have activation functions that limit their output to a range of [-1,1] while the maximum value these functions have is 1 they suffer from a different problem(this one is much more common since the computing power of modern computers allows them to deal with very big numbers). As said these activation functions have a maximum value of 1 but take all real numbers as input thus they must have a horizontal asymptote at -1 and 1. As the input gets bigger and the output approaches 1 the gradient of the function goes to 0. As said in back-propagation equation 1 and 2 we use the derivative of the activation function in our algorithm and it affects the learning rate. But since the value of the derivative goes to 0 the learning rate goes to 0 (the network is called saturated at this point) and the network stops learning. This problem is seen mostly in deep networks with thousands of inputs and hundreds of layers and it is called vanishing gradient problem

6.3 How to compare different activation functions based on their performance and accuracy

In order to compare different activation functions we need a numerical value assigned to their performance. We can calculate this value using either by dividing their total number of errors by the total number of examples in the test group which gives us the error percentage. Another way we can use to calculate the efficiency of an activation function is the time required to complete and learn a set number of training samples. This time will give us an insight about the complexity of the activation function. This time being long means that the activation function is comparatively more complex. One final thing we can do is divide the train data set into little groups called epochs and after a network finishes training in a given epoch we could try it using a group of test samples. We can do this process repeatedly until we run out of training samples. While doing this we can plot the error percentage over epoch numbers. Finally we can get the gradient of the graph we make. This gradient will leave us with the learning rate of our network

6.4 ReLU

Rectified linear unit or ReLU is a piece-wise defined activation function commonly used in deep neural networks. It maps $R \rightarrow R^+ + \{0\}$. It can be represented as $f(x) = max(0, x)$. It outputs 0 when the input is - and it outputs the input value when the input is positive it can be written explicitly as $f(x) = \{0, x < 0; x, x > 0\}$. ReLU is relatively a new invention in the world of machine learning but its benefits are worthwhile. First of all it eases the computation required by a lot since in the interval where it is continuous(R^+) its first derivative is 1. It also solves the "vanishing gradient problem" since

it does not have an asymptotic upper and lower bound meaning the derivative doesn't go to 0 as the input goes to 0 or ∞

6.5 Logistic function / Sigmoid function

The Sigmoid (logistic) function is a continuous function that maps $R \to (0, 1)$. It is continuously differentiable. It's derivative is always non negative but as the input gets larger the derivative goes to 0. $\lim_{x\to\infty} \frac{dy}{dx} \to 0$ Because of this learning can stop(vanishing gradient problem). Logistic function can be written as $\frac{1}{1+e^{-x}}$ it has a derivative of$\frac{e^{-x}}{(1+e^{-x})^2}$ It has asymptotes at y=0 and y=1

6.6 Arctan

Arctan is a continuous function which maps $R \to (-\frac{\pi}{2}, \frac{\pi}{2})$ It is commonly used in trigonometry to evaluate an angle whichs tangent value is known. It does not have a vertical asymptote but unlike ReLU it has horizontal asymptotes which makes it susceptible to "vanishing gradient problem" it can be represented as $f(x) = arctan(x)$ It has a derivative of $\frac{1}{1+x^2}$ which is not as easy to calculate as that of ReLU which is a constant and since as $\lim_{x\to\infty} \frac{dy}{dx} \to 0$ which causes the vanishing gradient problem

6.7 Tanh

Hyperbolic tangent or tanh is a trigonometric function defined in hyperbolic geometry it is defined $\frac{\sinh}{\cosh}$ It maps $R \to (-1, 1)$ It is continuously differentiable and just like logistic function. its derivative goes to 0 as $x \to \pm\infty$ $\lim_{x\to\pm\infty} \frac{dy}{dx} \to 0$ Its advantage over logistic function is that it maps low values close to -1 which in turn change those values from indifferent to negatively affecting. Tanh can be represented as $\frac{e^{2x}-1}{e^{2x}+1}$ and it has a derivative of $1 - tanh^2$

7 Brief explanation of a neural network using a real example

For simplicity in this part I will define a very basic network and using that network I will explain how feedforward and backpropagation works also I will explain what is loss and accuracy based on real numerical values for this network
First we need to define how many neurons and layers our network has. Since this is a demonstrative model I chose an input layer, a hidden layer and an output layer. All of these layers consist of 1 neuron. This shallow of a network surely won't work as good as deep network but since all the calculations will remain same this is a good demonstrative model
After we decide what our network's architecture would look like we need to define initial weights and biases of the network. Initial weights and biases can be randomly chosen so I wrote a little code that chose these values randomly for us. Also we need to choose a learning rate which can also be randomly chosen
Initial Weights Biases and learning rate:
learning rate = 0.1
$Weight_1$: [[0.4703668]]
$Weight_2$: [[0.02151528]]
$Bias_1$: [[0.08962905]]
$Bias_2$: [[0.14936365]]
While in normal networks we change the input and therefore the output with each iteration this type of network won't be able to adapt to new inputs and outputs due to its limited number of neurons in the hidden layer therefore In this example I kept the Input and output value the same. In normal applications of neural networks this will lead to overfitting which would not be a desirable consequence.
Input value(X): [[0.89259312]]
Desired output value ($y = X^2$):[[0.79672248]]
Feed forward
In this section of the network we will -as the name suggests- feed the values we get in the input layer forward till we reach the output layer. As said this is a demonstrative model so I will be using a simple sigmoid function as my activation function but with any other function the procedure will remain unchanged.
a_2 the activation of the hidden layer Activation function$(X \times Weight_1 + bias_1)$
a_3 the activation of the output layer Activation function$(X \times Weight_2 + bias_2)$
Numerical values from this example
$a_2 = \sigma(0.89259312 \times 0.4703668 + 0.08962905) = 0.6246834456269417$
$a_3 = \sigma(a_2 \times 0.02151528 + 0.14936365) = 0.5406113113176128$

After we calculated activation of the output layer we would need to calculate the Cost of the network. In this example in order to keep calculations simple I used the difference of the calculated value and the expected value. This may cause problems in a real neural network since this can lead to negative Cost

which may lead to weights and biases being adjusted in the wrong direction.

Cost = $y - a_3$
Numerical values from this example
Cost = 0.79672248-0.54061131=0.25611117

Back-Propagation
Error In the output layer = Cost $\times\ \sigma^{'}(a_3)$
Error In the hidden layer = Error In the output layer $\times weight_2 \times \sigma^{'(a_2)}$

Numerical values from this example

Error In the output layer = 0.25611117×0.24835072139306394 = 0.06360539382632163
Error In the hidden layer = 0.06360539382632163×0.02151528×0.23445403838659348
=0.0003208475047169335

Finally we would need to update the values of the weights and biases based on the error we calculated

we would need to change the weight in n+1 st layer by a_n×Error In the a_{n+1}×learning rate

Numerical values from this example
0.02151528 → 0.02151528 + 0.6246834456269417×0.06360539382632163×0.1 = 0.0254886036575885
0.4703668 → 0.4703668+0.89259312×0.0003208475047169335×0.1 = 0.470395438627528

We do the backpropagation and changing the values until we get satisfied we the accuracy we got

Loss can be described as the difference between the value we wanted and the value we had. For example if we calculated a value of 0.4 while the value we needed was 1.0 we would have a loss of $\frac{0.6}{0.1}$= %60. Accuracy is the percentage of true outputs. Test accuracy is the percentage of correct guesses the network got in the test sample.

8 Data collected from the experiment

This paper is written in latex Due to its limitations graphs need to be small but full sized graphs can be found at the appendix section

8.1 ReLU

epoch number	time taken for epoch	time per step	loss	change in loss	accuracy	change in accuracy	batch size
1	12s	24ms	20,39%	-	93,94%	-	469
2	11s	24ms	5,66%	14,73%	98,26%	4,32%	469
3	11s	24ms	4,02%	1,64%	98,79%	0,53%	469
4	11s	24ms	2,94%	1,08%	99,06%	0,27%	469
5	11s	24ms	2,23%	0,71%	99,30%	0,24%	469
6	11s	24ms	1,82%	0,41%	99,43%	0,13%	469
7	11s	24ms	1,58%	0,24%	99,50%	0,07%	469
8	11s	24ms	1,20%	0,38%	99,62%	0,12%	469
9	11s	24ms	1,15%	0,05%	99,61%	-0,01%	469
10	11s	23ms	0,79%	0,36%	99,74%	0,13%	469

Table 1: ReLU data table

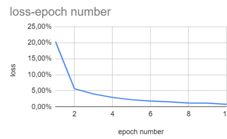


Figure 1: loss-epoch number

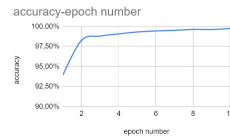


Figure 2: accuracy-epoch number

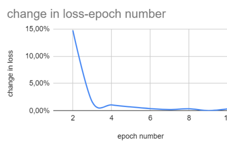


Figure 3: change in loss-epoch number

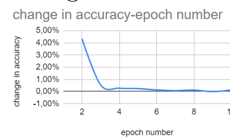


Figure 4: change in accuracy-epoch number

test accuracy :99,22%

8.2 Logistic function

epoch number	time taken for epoch	time per step	loss	change in loss	accuracy	change in accuracy	batch size
1	1s	5ms	109,46%	-	74,34%	-	128
2	1s	5ms	35,84%	73,62%	90,51%	16,17%	128
3	1s	4ms	23,01%	12,83%	92,29%	1,78%	128
4	1s	4ms	22,93%	0,08%	93,39%	1,10%	128
5	1s	4ms	19,79%	3,14%	94,28%	0,89%	128
6	1s	4ms	17,47%	2,32%	94,43%	0,15%	128
7	0s	4ms	15,54%	1,93%	95,45%	1,02%	128
8	0s	4ms	13,83%	1,71%	95,98%	0,53%	128
9	1s	4ms	12,28%	1,55%	96,46%	0,48%	128
10	0s	4ms	11,07%	1,21%	96,86%	0,40%	128

Table 2: Sigmoid data table

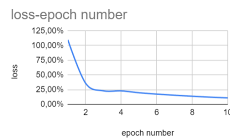


Figure 5: loss-epoch number

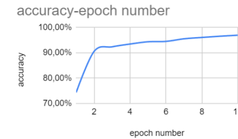


Figure 6: accuracy-epoch number

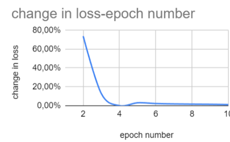


Figure 7: change in loss-epoch number

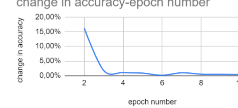


Figure 8: change in accuracy-epoch number

test accuracy :96,3%

8.3 Arctan

epoch number	time taken for epoch	time per step	loss	change in loss	accuracy	change in accuracy	batch size
1	1s	6ms	42,59%	-	87,76%	-	128
2	1s	6ms	26,11%	16,48%	92,58%	4,82%	128
3	1s	6ms	21,64%	4,47%	93,94%	1,36%	128
4	1s	7ms	17,96%	3,68%	94,94%	1,00%	128
5	1s	6ms	15,09%	2,87%	95,68%	0,74%	128
6	1s	6ms	12,99%	2,10%	96,32%	0,64%	128
7	1s	6ms	11,20%	1,79%	96,83%	0,51%	128
8	1s	6ms	9,67%	1,53%	97,26%	0,43%	128
9	1s	6ms	8,65%	1,02%	97,51%	0,25%	128
10	1s	6ms	7,63%	1,02%	97,82%	0,31%	128

Table 3: Arctan data table

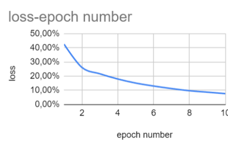


Figure 9: loss-epoch number

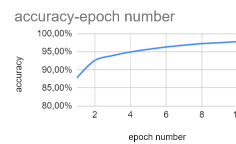


Figure 10: accuracy-epoch number

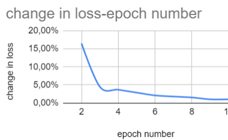


Figure 11: change in loss-epoch number

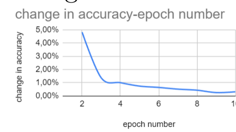


Figure 12: change in accuracy-epoch number

test accuracy :97,25%

8.4 Tanh

epoch number	time taken for epoch	time per step	loss	change in loss	accuracy	change in accuracy	batch size
1	3s	6ms	27,35%	-	91,88%	-	469
2	3s	6ms	12,31%	15,04%	96,31%	4,43%	469
3	2s	5ms	8,24%	4,07%	97,46%	1,15%	469
4	2s	5ms	5,78%	2,46%	98,27%	0,81%	469
5	2s	5ms	4,28%	1,50%	98,70%	0,43%	469
6	2s	5ms	3,32%	0,96%	99,02%	0,32%	469
7	2s	5ms	2,52%	0,80%	99,29%	0,27%	469
8	2s	5ms	1,78%	0,74%	99,53%	0,24%	469
9	2s	5ms	1,51%	0,27%	99,61%	0,08%	469
10	2s	5ms	1,18%	0,33%	99,67%	0,06%	469

Table 4: Tanh data table

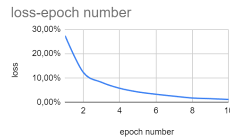


Figure 13: loss-epoch number

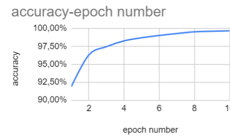


Figure 14: accuracy-epoch number

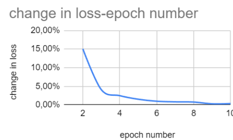


Figure 15: change in loss-epoch number

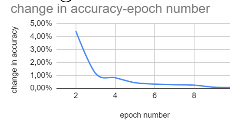


Figure 16: change in accuracy-epoch number

test accuracy :97,81%

9 Conclusion Applications and Limitations

Data collected From the experiment show that ReLU has the best overall test accuracy[99,22%]. This is followed by tanh[97,81%] arctan[97,25%] and finally sigmoid function[96,30]. Difference between these values may seem small but by the standards of computer science and neural networks ReLU is miles ahead of the competition. The reason for this difference can be stated as ReLU being a linear function and therefore not facing the vanishing gradient problem. When we look at the time taken for the network to train a different picture appears.

While ReLU and Tanh had bigger training data sets compared to arctan and sigmoid functions, ReLU needed significantly more time than all the other activation functions. This result was unexpected since one of the benefits of ReLU is its easy to compute derivatives. These unexpected results may be caused by bad programming since the code for the other functions needed to change to allow for a different activation function. Since the data set used for these models is comparatively very small any results related to time can be ignored. Another thing that was expected and can be seen from the results is that the accuracy of the first epoch is the greatest in the ReLU and follows the same trend as the accuracy and decreases as you go from Tanh to arctan to Sigmoid

As expected from the theoretical work and seen from the experimental results the best performing activation function is found to be the ReLU. This result is probably caused by it being not affected by the vanishing gradient problem. While this is true this does not mean that ReLU is the best activation function period. As with anything in life every situation is unique and may require a different activation function for example a time sensitive and error tolerant situation may deploy the sigmoid function. So while this paper agrees that vanishing gradient problem will affect even the most shallow networks, it does not say that only ReLU like functions should be used rather it says vanishing gradient problem must be considered along with other factors when choosing a activation function for a neural network

9.1 limitations

1. The network that was tested in this paper is a very shallow network that will affect both the computation time and the extent of problems caused by vanishing gradient problems. In order to be more accurate further researchers could test activation functions on networks with different depths.

2. The network used in this paper is an image classification network. Some activation functions work better than others in some use cases therefore in order to be more accurate further researchers could test different types of networks.

3. While the time taken for the network to run is not discussed the experimental values are still given. While this may not cause problems in comparative papers since the experiment is always run in the same computer if it was a qualitative analysis of the length of time required Big O notation would have been better since it does not get affected by change of hardware

10 References

1. 3D plots. (n.d.). Retrieved March 25, 2023, from https://www.qtiplot.com/doc/manual-en/x730.html

2. DeepAI. (2019, May 17). Relu. DeepAI. Retrieved March 25, 2023, from https://deepai.org/machine-learning-glossary-and-terms/relu

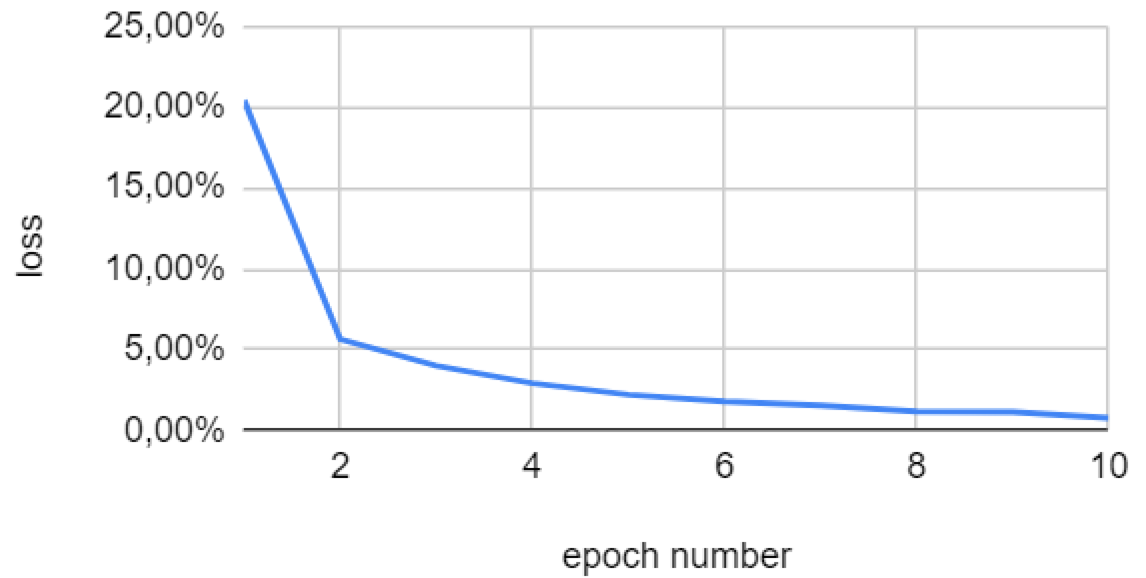
3. DeepAI. (2019, May 17). Vanishing gradient problem. DeepAI. Retrieved March 25, 2023, from https://deepai.org/machine-learning-glossary-and-terms/vanishing-gradient-problem

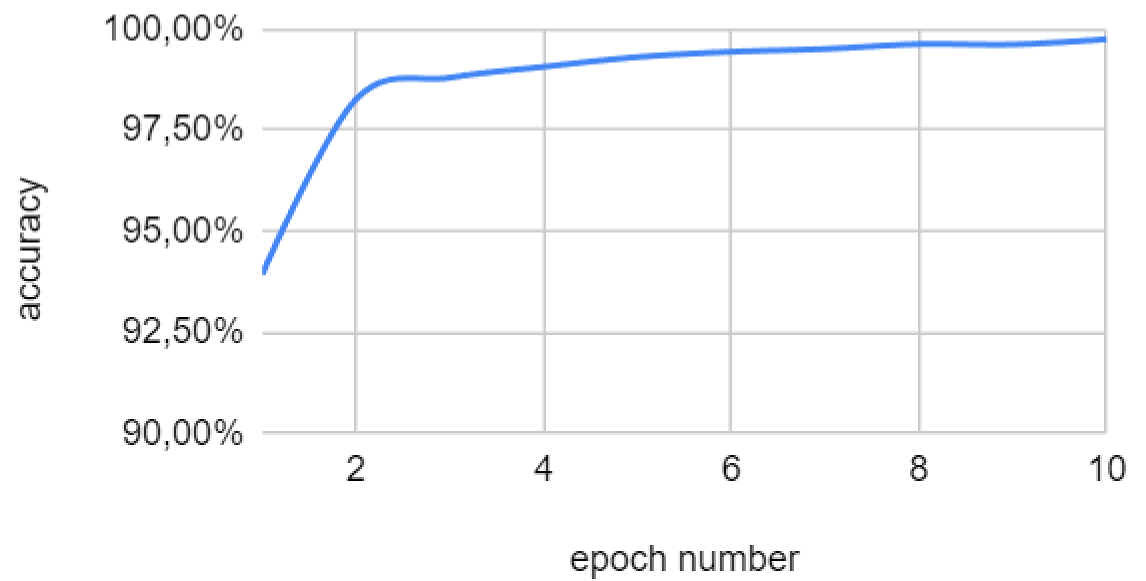
4. Nielsen, M. A. (1970, January 1). Neural networks and deep learning. Retrieved March 25, 2023, from http://neuralnetworksanddeeplearning.com/

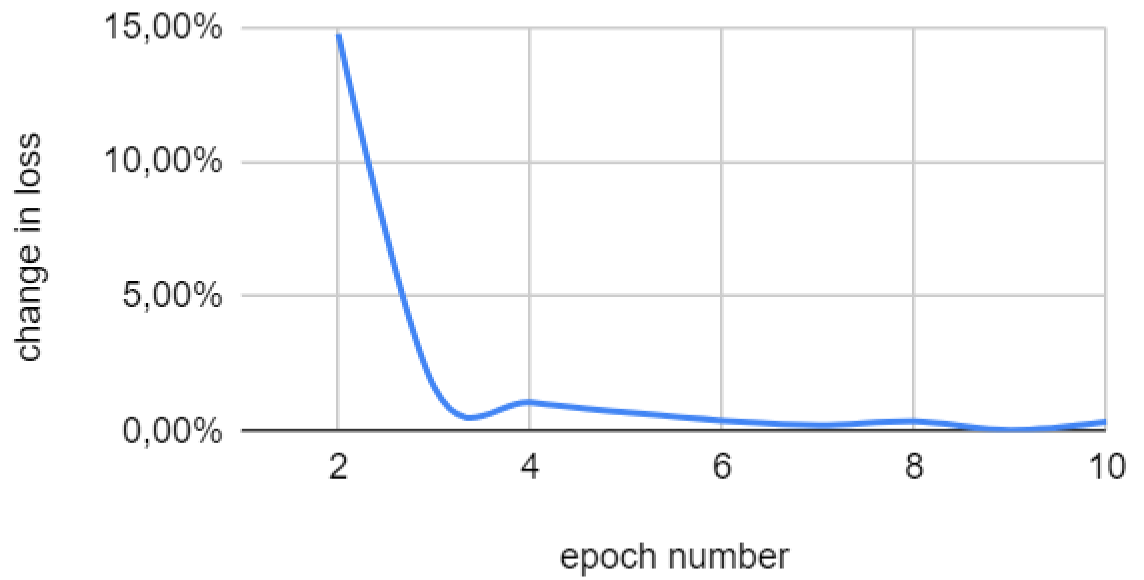
5. Sharma, S. (2022, November 20). Activation functions in neural networks. Medium. Retrieved March 25, 2023, from https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6: :text=tanh%20is%20also%20sigmoida l%20(s%20%2D%20shaped). text=The%20advantage%20is%20that%20the,its%20deriv ative%20is%20not%20monotonic.

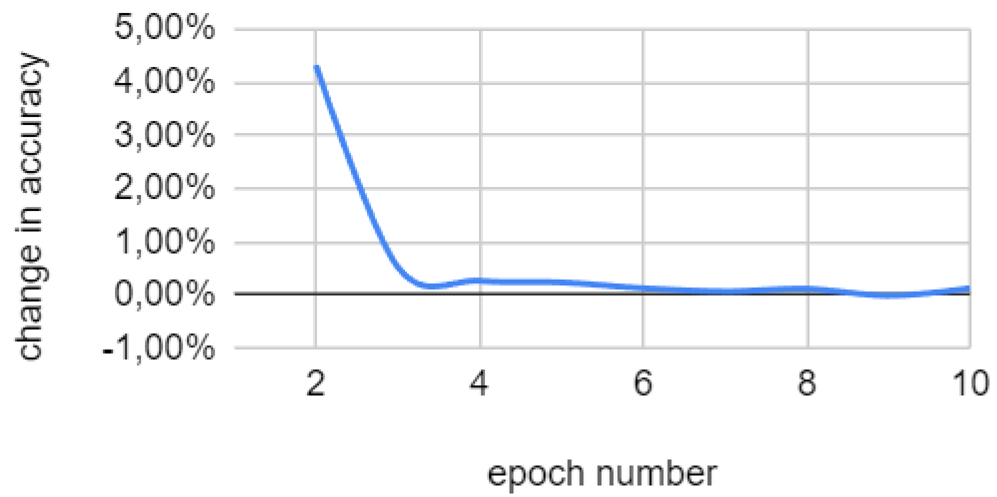
11 Appendix

11.1 ReLU Graphs

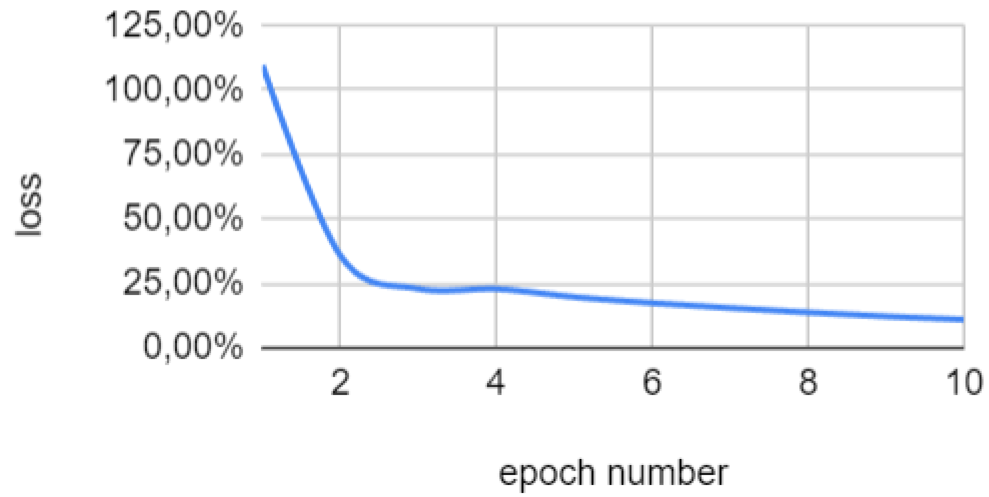


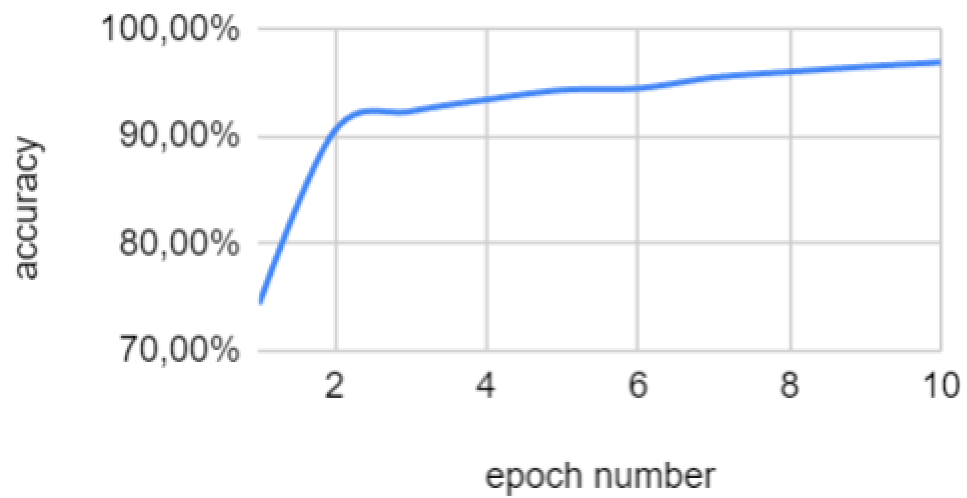


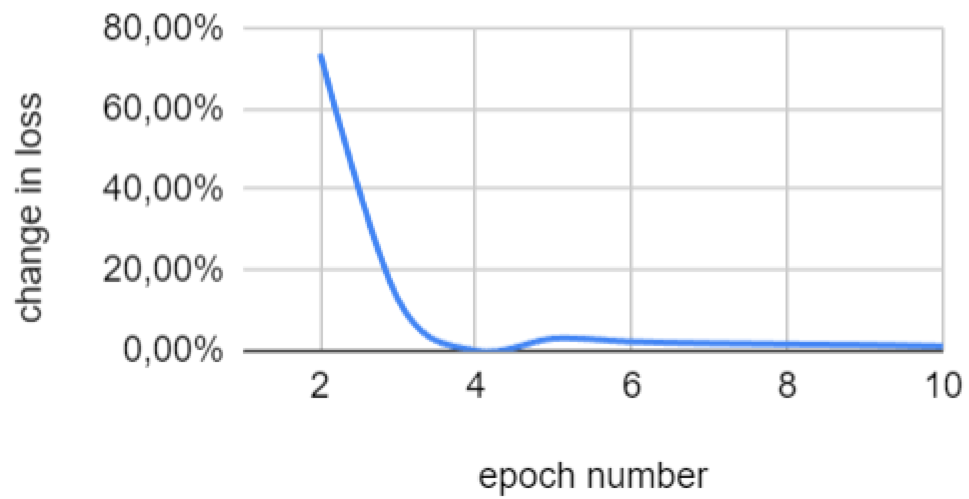

change in loss-epoch number
15,00%
10,00%
5,00%
0,00%
change in loss
2
4
6
8
10
epoch number

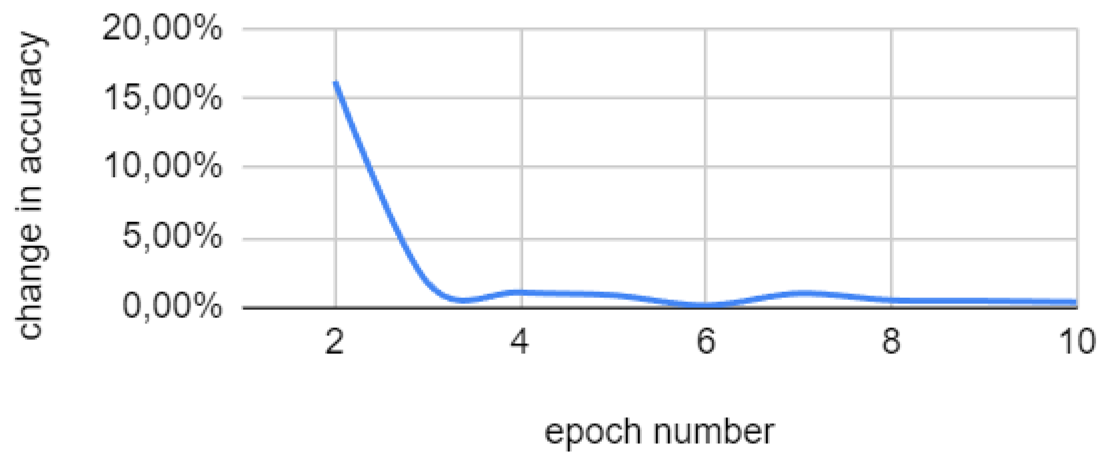

change in accuracy-epoch number
5,00%
4,00%
3,00%
2,00%
1,00%
0,00%
-1,00%
change in accuracy
2
4
6
8
10
epoch number

11.2 Logistic function Graphs

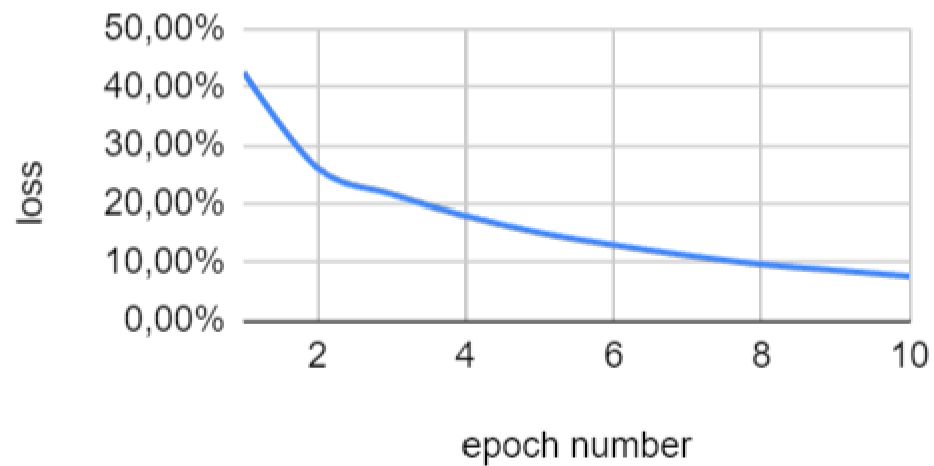


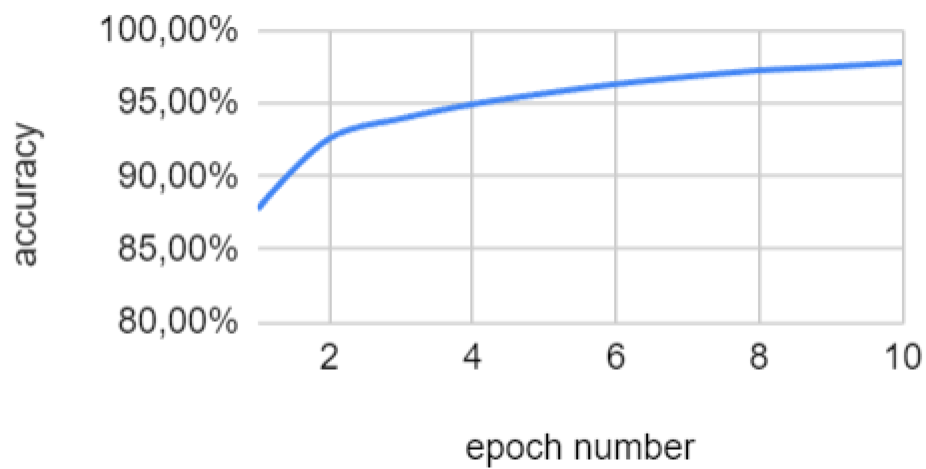


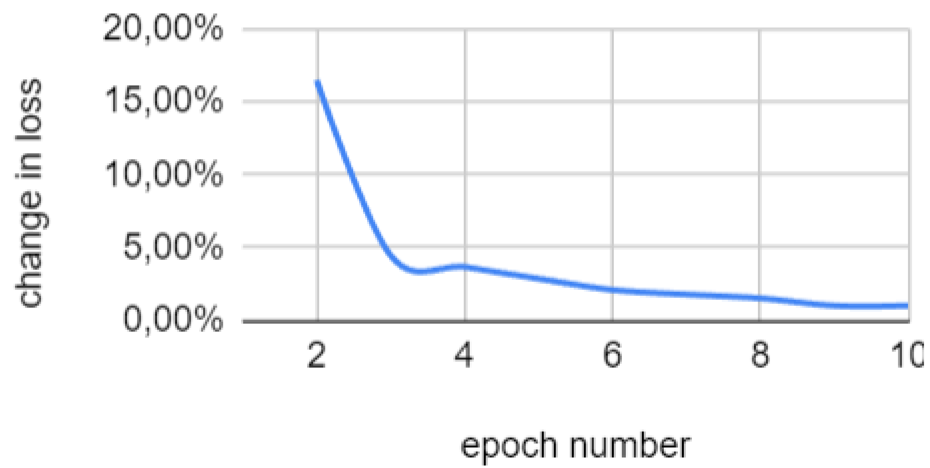

change in loss-epoch number
80,00%
60,00%
40,00%
20,00%
0,00%
change in loss
2
4
6
8
10
epoch number

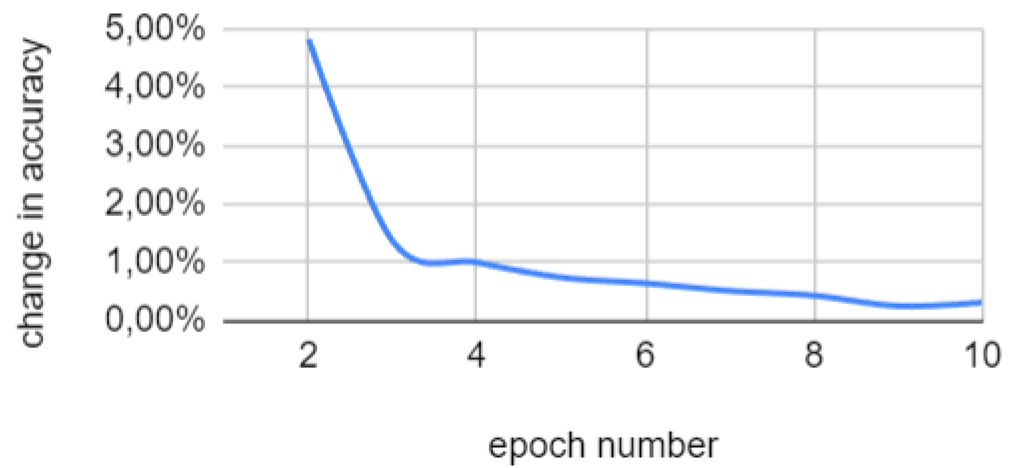

change in accuracy-epoch number
20,00%
15,00%
10,00%
5,00%
0,00%
change in accuracy
2
4
6
8
10
epoch number

11.3 Arctan

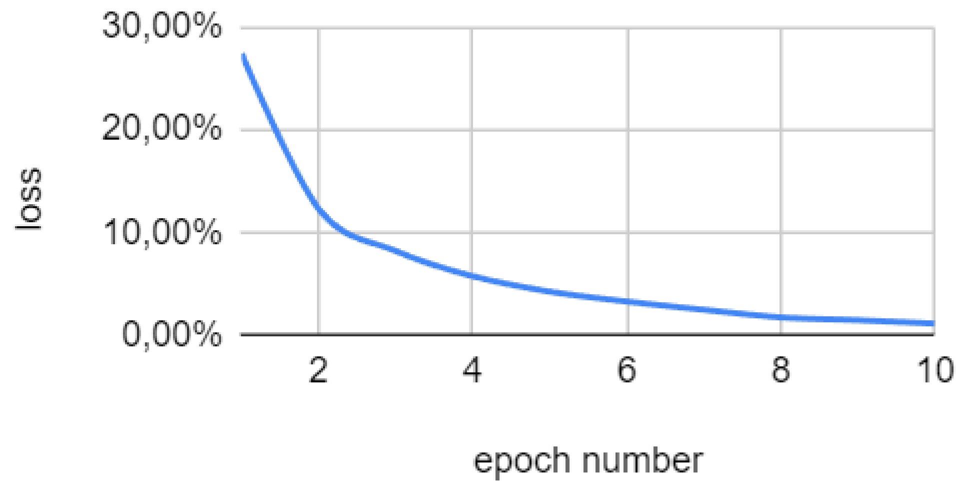


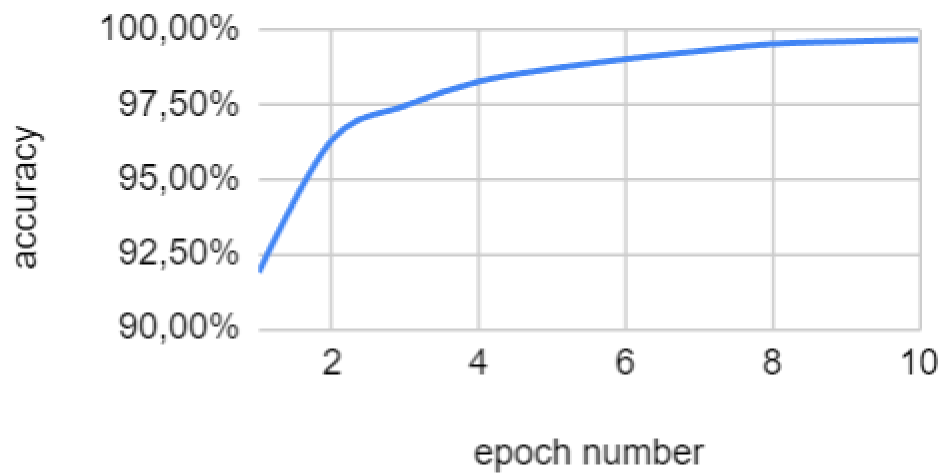


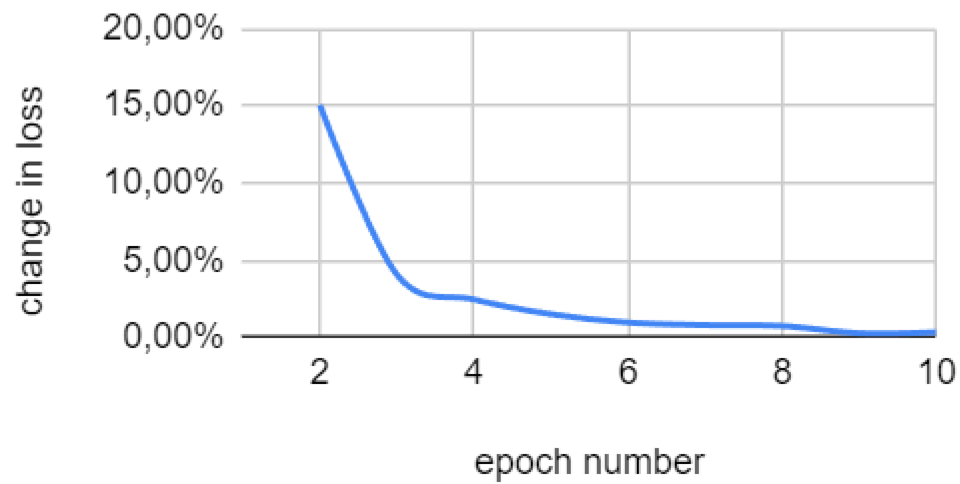

change in loss-epoch number
change in loss
20,00%
15,00%
10,00%
5,00%
0,00%
2
4
6
8
10
epoch number

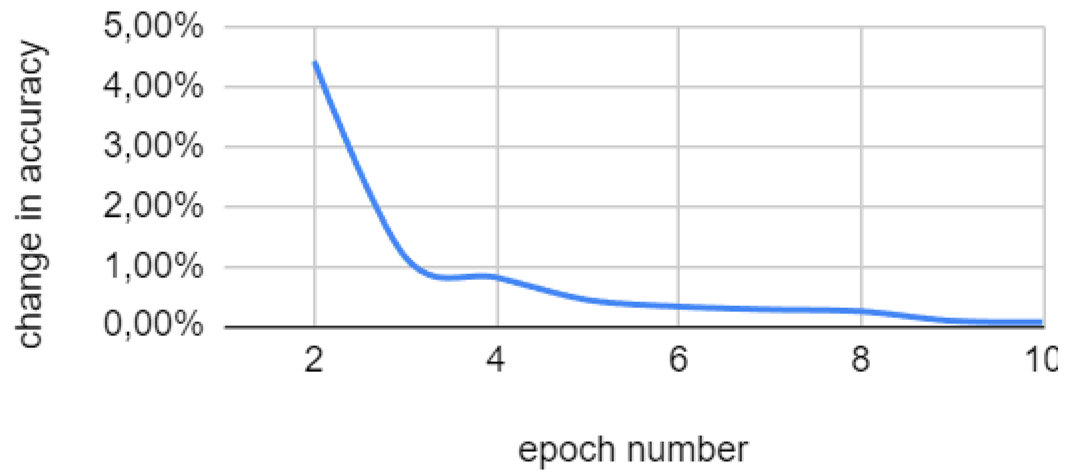

change in accuracy-epoch number
change in accuracy
5,00%
4,00%
3,00%
2,00%
1,00%
0,00%
2
4
6
8
10
epoch number

11.4 Tanh






change in loss-epoch number
20,00%
15,00%
10,00%
5,00%
0,00%
change in loss
2
4
6
8
10
epoch number


change in accuracy-epoch number
5,00%
4,00%
3,00%
2,00%
1,00%
0,00%
change in accuracy
2
4
6
8
epoch number