Cybenko's Theorem

and the capability of a neural network as function approximator

Andrea Lörke
Fabian Schneider
Johannes Heck
Patrick Nitter

24. September 2019

Contents

1 Introduction

Nowadays, neural networks are quite popular. Whether it's voice-, image-recognition, or other assistance, this "new" technology advances the standard of life for various people. But it's not that new, rather close to 50 years old. In the present-day, its development reached a new peak, which further increased its popularity. Consequently, one question appears, what is a neural network in simple terms? Regarding supervised learning, a neural network calculates, in correlation to the in- and output data, a model or a pattern. An example would be a cat-picture for an image-recognition, that displays the pattern "cat" over so-called machine learning. As the name indicates, the machine (computer) learns to differentiate among several representations of cats. As a result, it establishes various identifying characteristics - like eyes, ears, pelt, and so forth - which create the pattern cat. With this, the concept of AI and particularly deep learning distinguishes itself from the recognized algorithms, mainly because the others couldn't solve similar sets of obstacles before.

In 1989 George Cybenko confirmed, that a neural network with solely one hidden-layer is capable of always approximating a multi-variant continuous function. Although the neural network in this paper operates on regression, nevertheless, this document won't address the necessity of neural networks in regression and classification of data.

The following chapters discuss shallow networks, which are neural networks with only one of the so-called hidden-layer, and therefore, satisfies Cybenko's theorem for function approximation. There are several techniques, for the learning process, in this paper back-propagation will be applied and explained from a mathematical perspective in chapter two. Then a visual representation and explanation for the effects of Cybenko's theorem are illustrated and lastly, the practical implementation in Matlab, and its results.

2 Mathematical - Backpropagation

A node receives one or more inputs $x_1 \dots x_n$, multiplies them with weights $w_1 \dots w_n$ and adds a bias b to estimate a value v.

$$v = \sum_{i=1}^{n} w_i x_i + b$$

Using matrices, this equation can be written as $v = Wx + b$, where $W = [w_1 \dots, w_n]$ $x = [x_1 \dots x_n]^t$. In most cases, there will be a function σ applied to this value to determine an output $y = \sigma(v)$. The nature of this so-called activation function will be described later.
Utilizing multiple of these nodes a single-layer neural network with the inputs $x = [x_1 \dots x_n]^t$ and the outputs $y = [y_1 \dots y_m]$ is produced. The output y_i will now be calculated by $y_i = \sum_{j=1}^{n} w_{ij} x_j + b_i$.

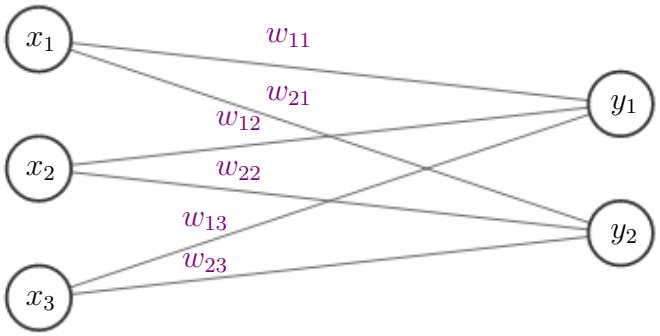


Figure 1: A single layer neural network

Again using matrices will simplify the equation for the output to $y = \sigma(Wx + b)$, where $W = \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \end{pmatrix}$ and $b = [b_1 \dots b_m]$. However, this expression can be reduced even further by appending the number one to the input $x = [x_1 \dots x_n \ 1]^t$ and the bias to the weights $W = \begin{pmatrix} w_{11} & \cdots & w_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ w_{m1} & \cdots & w_{mn} & b_m \end{pmatrix}$. This will not change the value of y but end in the formula $y = \sigma(Wx)$ with the new dimensions for W and x.
Assume now there is concrete data d, that some neural network is supposed to learn, and some explaining variables x, which describe the input- data. Beginning with random values for the weights and values one for biases, the discrepancy linking the output of our network and the supplied data decreases gradually through the gradient descent method.

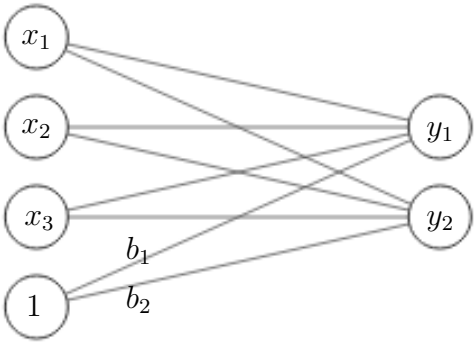


Figure 2: Another implementation for the bias

To accomplish such, a loss-function must be defined as a measurement on how large the value of the error is. A common loss-function would be the squared error: $E = \frac{1}{2}(d-y)^2$. Here y is the output from the network and d is the target-solution to the input. Gradient descent is employed to minimize the error. Therefore its derivative to the matrix W is acquired and can be multiplied with a positive parameter α and subtracted from the preceding value to complete a step. This α is called the learning rate. Picking a high learning rate produces a faster learning network. Nevertheless, if it is too high, the iterations will never converge, and consequently, the neural network won't learn at all. But it shouldn't be too low as well, because it may result in a long training process that could become stuck.

$$W_{new} = W_{old} - \alpha \frac{\partial E}{\partial W} = W_{old} - \alpha \begin{pmatrix} \frac{\partial E}{\partial W_{11}} & \cdots & \frac{\partial E}{\partial W_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial W_{m1}} & \cdots & \frac{\partial E}{\partial W_{mn}} \end{pmatrix}$$

After repeating this process as often as expedient, the procedure decreases the error to an acceptable value, but depending on the complexity of the problem this might also fail. For instance, single-layer neural networks are not capable of solving the XOR-Problem. This is why the coming section proposes a more complicated neural network with more reliable approximation capacities.
A shallow neural network adds a second layer of nodes, the hidden nodes, before the output.

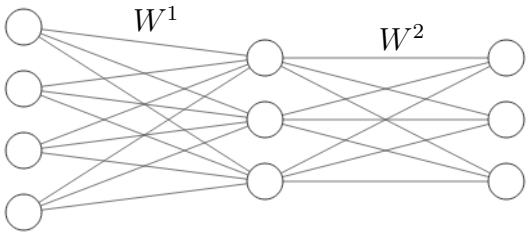


Figure 3: A shallow neural network

The output y will now depend on the matrices W^1 and W^2:

$$y = \sigma_2(W^2\sigma_1(W^1x))$$

Keep in mind, that the implementation of the bias is accomplished by adding a one to the input and attaching another column to the matrices containing the weights. This shallow neural network uses the following activation-function σ_1: $x \mapsto \frac{1}{1+e^{-x}}$ and $\sigma_2 : x \mapsto x$.

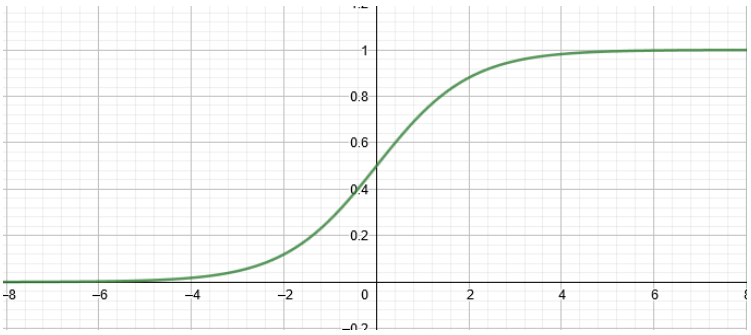


Figure 4: The graph of the sigmoid function σ_1

Gradient descent is used to update W^2 and therefore minimize the error E, described by a loss-function, in this case: $E(y) = \frac{1}{2}(d-y)^2$. To minimize E, the formula $y = \sigma_2(W^2\sigma_1(W^1x))$ is used and embedded into E. To shorten the expressions $y_1 := \sigma_1(W^1x)$ is defined. Now the derivative is equal to

$$\frac{\partial}{\partial W^2}E(\sigma_2(W^2y_1)) = \frac{\partial}{\partial y}E(\sigma_2(W^2y_1)) \cdot \frac{\partial}{\partial W^2}y(\sigma_2(W^2y_1)) = -(d-y).*\sigma_2'(W^2y_1) \cdot y_1$$

For this formula these abbreviations are chosen: $e = (d-y)$ and $\delta = e.*\sigma_2'(W^2y_1)$. Here the multiplication .* is defined as the component-wise product of two vectors with the same length. According to gradient descent W^2 is updated:

$$W^2 = W^2 + \alpha \cdot \delta \cdot y_1^t$$

For W^1 backpropagation is used. Therefore, the values of the vector δ are assigned to the output y, as shown in the following figure.

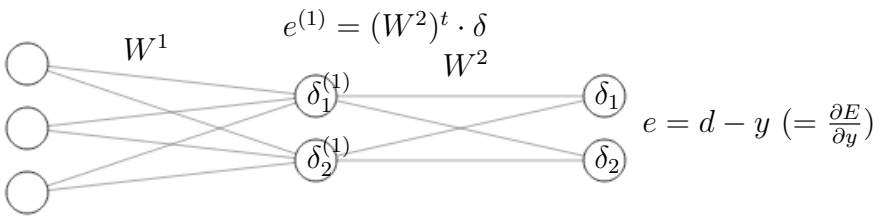


Figure 5: How backpropagation works

These values will now be back-propagated to the hidden layer, to define their error, using the weights in W^2.

$$e^{(1)} = (W^2)^t \delta$$

This error is used to compute the update for W_1 in a similar fashion:

$$W^1 = W^1 + \alpha \cdot \delta^{(1)} \cdot x^t$$

In this formula, $\delta^{(1)} = \sigma_1'(W^1 x) . * e^{(1)}$, which is essentially the same as the formula for δ, except σ_2 being replaced with σ_1 and y_1 with x (based on its dependents on the input of the layer).
Replicating this process of adding layers to the neural network produces a deep neural network, which approximates functions notably better. The output is therefore:

$$y = \sigma_n(W^n(\cdots(\sigma_1 W^1 x)))$$

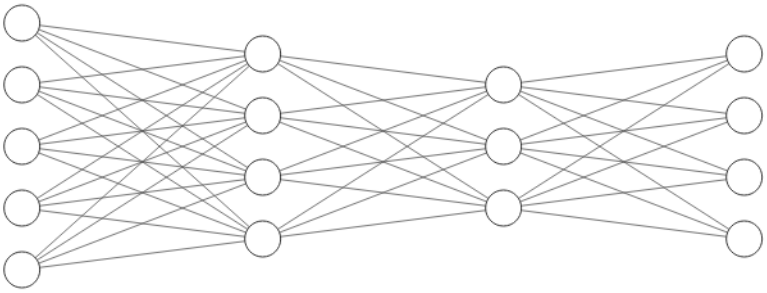


Figure 6: A Deep Neural Network

The weights will be updated as described above, using gradient descent for W^n and backpropagation for $W^{n-1}, \cdots, W^1$.

3 Visual Proof of the Cybenko-Theorem

In this chapter, the *Cybenko-Theorem*, predecessor to the *Universal Approximation Theorem*, is introduced. A visualization of the proof illustrates, how neural networks approximate a function in its fundamental steps. A "formal" mathematical proof shall not be presented here, as it can be found in the original paper of G. Cybenko.

Theorem (G. Cybenko):

Let σ be a sigmoidal function:

$$\sigma(x) = \begin{cases} 1 & \text{for } x \to +\infty \\ 0 & \text{for } x \to -\infty. \end{cases}$$

Then finite sums of the form

$$g(x) = \sum_{j=1}^{N} w_j^2 \sigma((w_j^1)^T x + b_j^1)$$

are dense in $C(I_n)$ with respect to the supremum norm. Where $x \in \mathbb{R}^n$, $w_j^2 \in \mathbb{R}^n$, $w_j^1 \in \mathbb{R}$, $b_j^1 \in \mathbb{R}$. I_n is the $n-$dimensional unit cube $[0,1]^n$ and $C(I_n)$ is noted for the space of continuous functions on I_n.
In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum, $g(x)$, of the above form, for which

$$|g(x) - f(x)| < \epsilon$$

for all $x \in I_n$.

This means, that any continuous function on a compact subset of $[0,1]^n$ can be approximated by a feed forward neural network with only one single hidden layer and a finite number of neurons.
The neural network computes any continuous function. It is important to explain this last statement since this doesn't imply, that **one** neural network can accurately approximate any arbitrary function with unspecified complexity. It requires, that the neural network and its parameters (number of hidden nodes, number of learning iterations etc.) must be adjusted for each unique function. Under the right circumstances, it is possible to achieve any desired accuracy ϵ.
Additionally, the class of functions, that can be approximated, does not always need to be continuous. For some noncontinuous function, a continuous approximation works too and therefore could be approximated by the neural network (for example a noncontinuous function with lift-able positions).

In the following, the Cybenko-Theorem is verified by a shallow neural network with one- and two-dimensional inputs, that is, one or two input nodes and one output node.

3.1 One-dimensional shallow-neural-network

The neural network, used to demonstrate the *Cybenko's Theorem* on the basis of some examples in section 4 of this paper, has the following form:

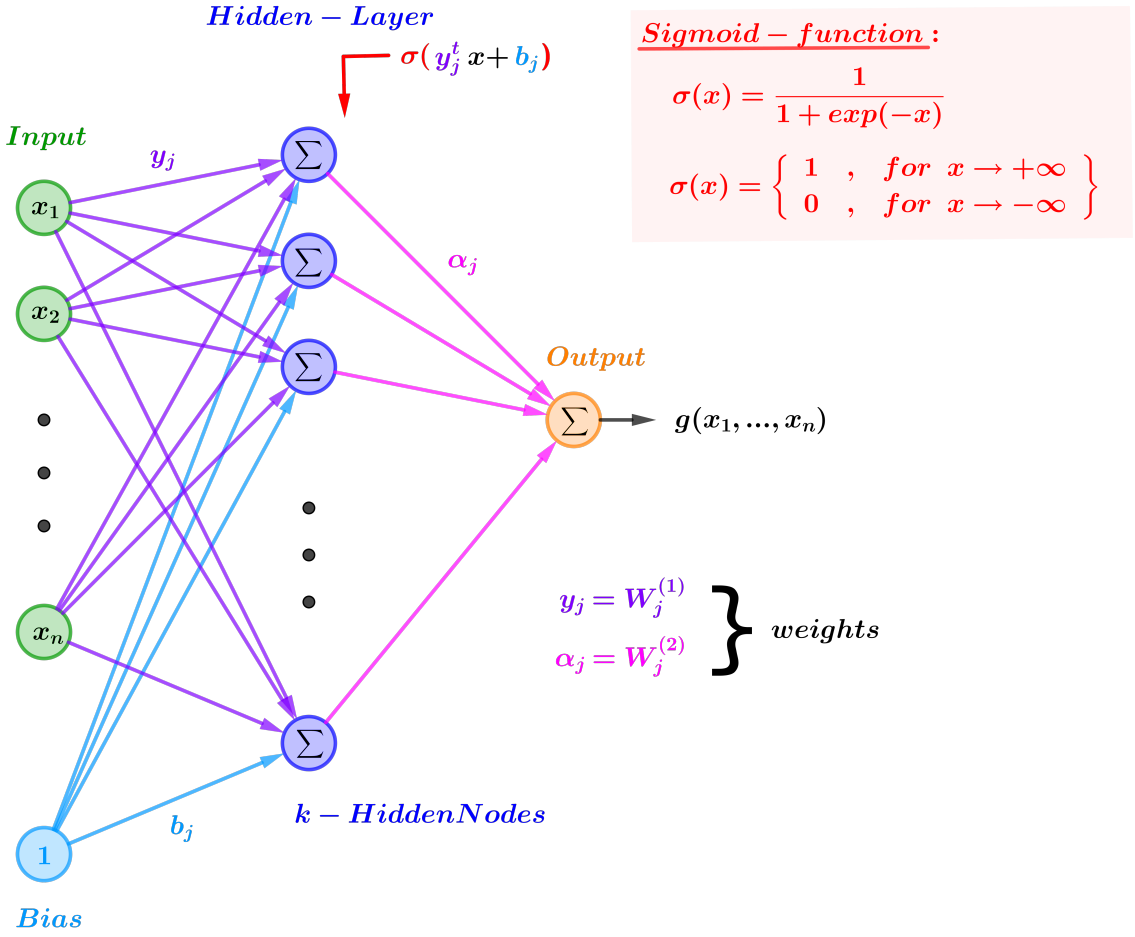


Figure 7: n-dimensional shallow-network with bias

In the one-dimensional case of the visual proof, the assumption is taken, that the neural network just has one input node, as illustrated in the following image:

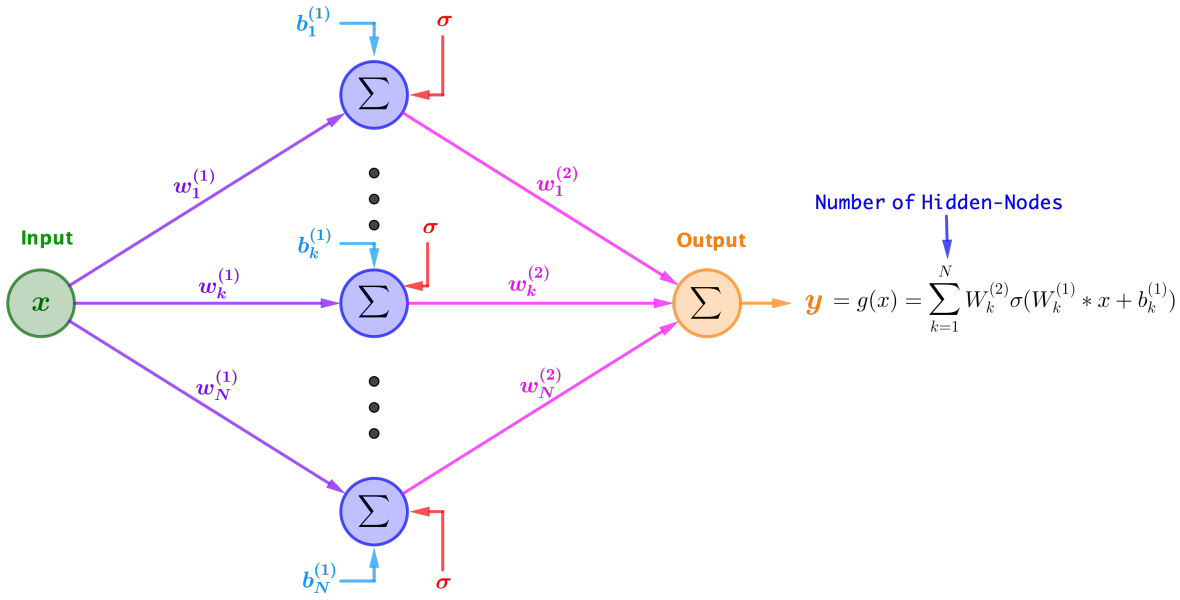


Figure 8: One-dimensional shallow-network with N-Hidden-Nodes

Similarly defined as in the first chapter and portrayed in the figure above, the neural network computes the output y. For the sake of simplicity within the visual proof, imagine a neural network with only two hidden nodes, a weight value $w_1^{(1)}$ and the corresponding bias $b_1^{(1)}$:

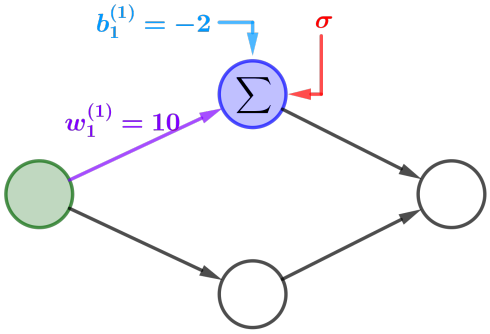


Figure 9: Computing of the top node .

As a representation for the other nodes, the top node is taken into focus. The output of the node is

$$\begin{aligned} \sigma(w_1^{(1)}x + b_1^{(1)}) &= \sigma(10x - 2) \\ &= \frac{1}{1 + e^{-(10x-2)}} \end{aligned}$$

The graph of the output of the top hidden node is visualized in the next figure:

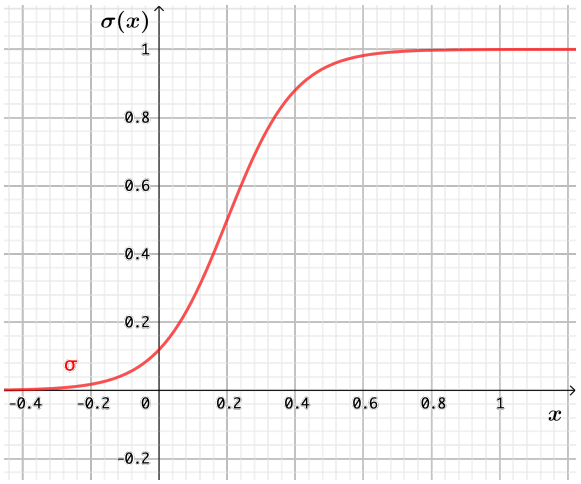


Figure 10: Sigmoid function

By increasing the weight value $w_1^{(1)}$, the graph becomes similar to a step function. If $w_1^{(1)} = 1000$ is chosen as an example, that behaviour can be seen in the following figure:

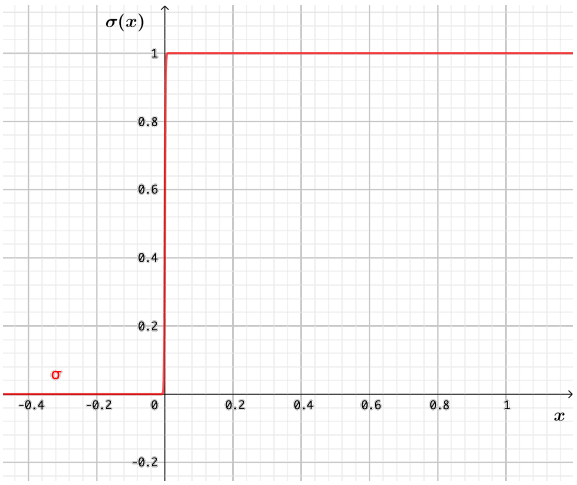


Figure 11: Increased weight value $w^{(1)}$

To simplify the analysis, a large weight value $w_1^{(1)} = 1000$ is utilized for further description. By increasing the bias $b_1^{(1)}$, the step moves to the right and by decreasing the bias value, it moves to the left. As a consequence, the bias determines the position of the step, which is also a threshold point, and that this step is at the position $\boxed{s = \frac{-b}{w}}$.

For example, for $b_1^{(1)} = -200$ the position of the step is at $s = 0,2$:

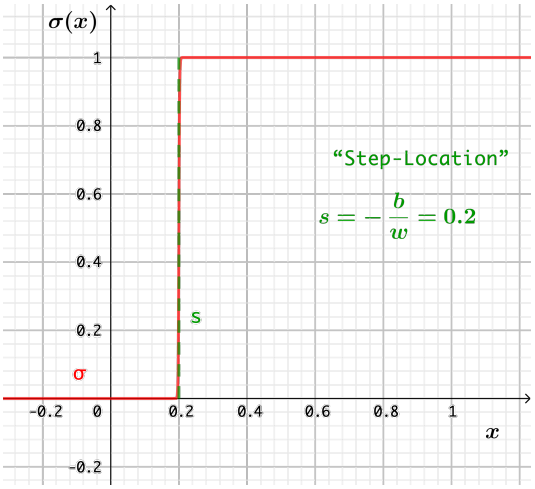


Figure 12: Step Sigmoid function

In this case, by adding another weight $w_1^{(2)} = 1.5$ to the output of the top hidden node, the output of the step function is scaled. This "height" of the step-sigmoid-function is determined by the second weight, which is illustrated in the right figure.

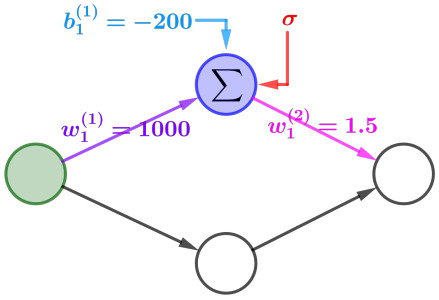


Figure 13: Addition of w_1^2

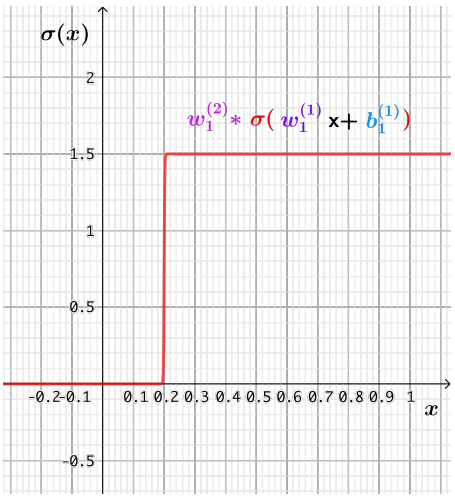


Figure 14: Scaled step-function

To complete the neural network from above, the missing two weights and another bias for the second hidden node are added with the example values shown in the next graph.

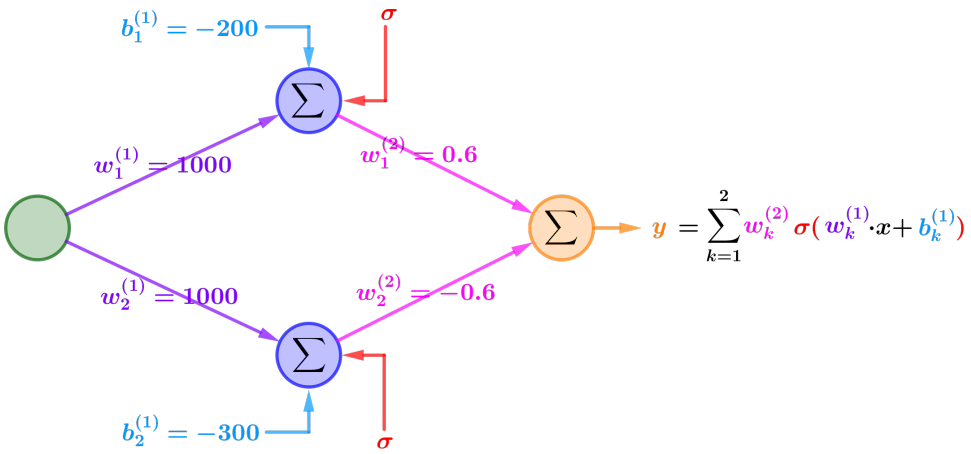


Figure 15: Neural network with 2 hidden nodes

Consequently, the weighted output y of the hidden layer is calculated as follows:

$$y := g(x) = \sum_{k=1}^{2} w_k^2 \sigma(w_k^1 x + b_k) = 0.6 * \sigma(1000x - 200) + (-0.6) * \sigma(1000x - 300) = 0.6$$

for $0.2 \leq x \leq 0.3$, and $y = 0$ elsewhere.

In conclusion, the graph of the output of the neural network is similar to a tower-function, as the two hidden nodes are "coupled" by setting $w_1^{(2)} = 0.6$ and $w_2^{(2)} = -0.6$:

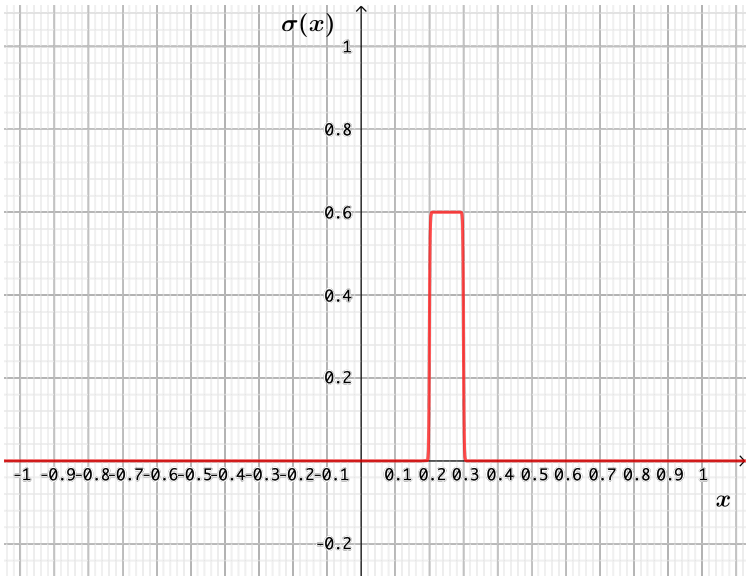


Figure 16: Coupled nodes

On the basis of the *mean value theorem for integrals*, any continuous function of one variable can be approximated by dividing the interval $[0, 1]$ in N infinitesimal sub-intervals. Thus by constructing a shallow neural network with $2N$ hidden nodes - which means N coupled nodes - it is possible to construct as many towers as desired. Therefore, any of these functions can be approximated by the neural network.

The structure of the one-dimensional neural network with $2N$ hidden nodes can be visualized similar to figure 8 but with $2N$ hidden nodes instead of N.

Therefore, the output of the neural network y is calculated as before:

$$y = \sum_{k=1}^{2N} w_k^{(2)} \sigma(w_k^{(1)} x + b_k).$$

To visualize the applied method more accurately, the following function is used as an example of a continuous function that is non-trivial:

$$f(x) = 0.2x + 0.4x^2 + 0.3xsin(15x) + 0.05cos(50x) + 0.1.$$

For this example, the interval $[0, 1]$ is divided into $N = 4$ sub-intervals. Which means, that 8 hidden nodes must be used for coupling them into 4 pairs:

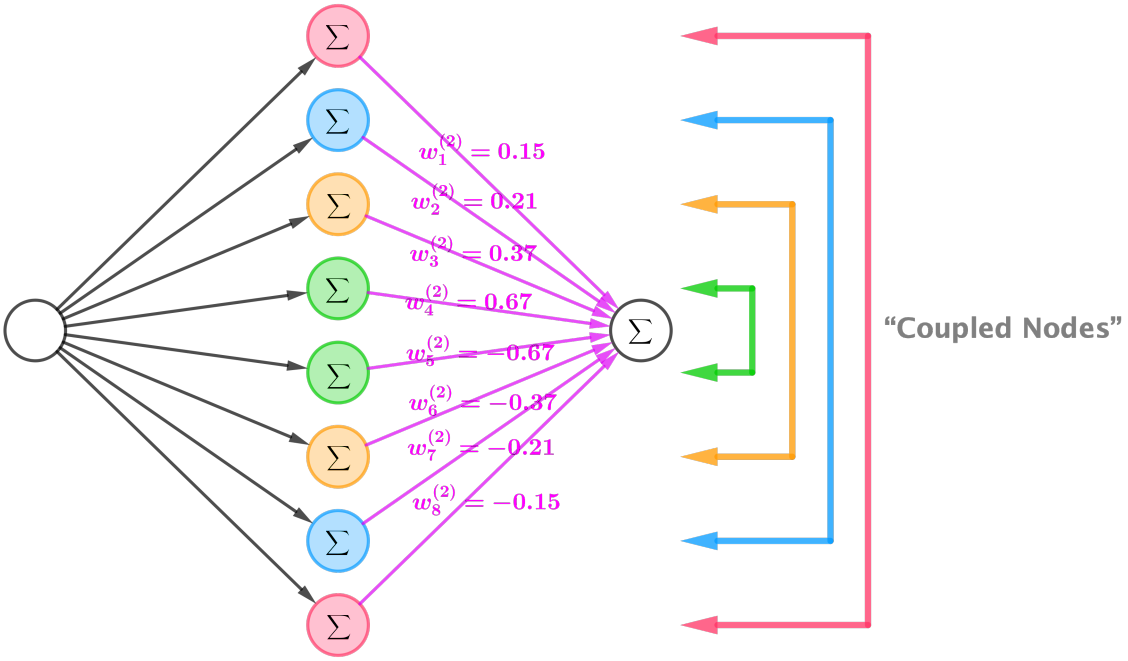


Figure 17: neural network with coupled nodes

With this amount of hidden nodes, the following approximation of the example function f can be achieved:

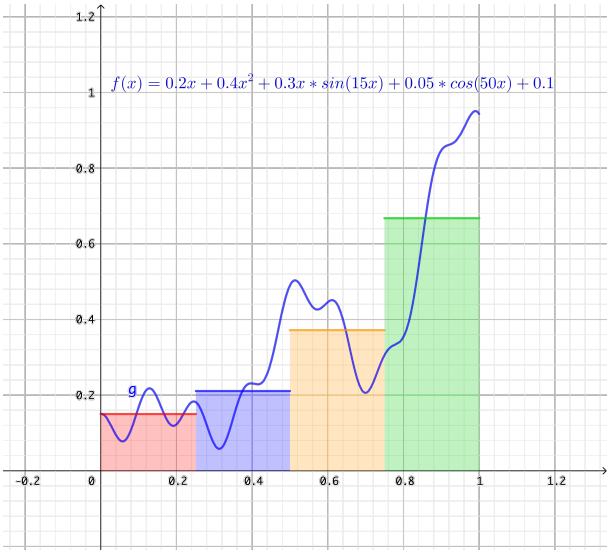


Figure 18: Approximation (N=4)

By increasing the number of hidden node pairs to i. e. $N = 100$, f can be approximated with higher accuracy:

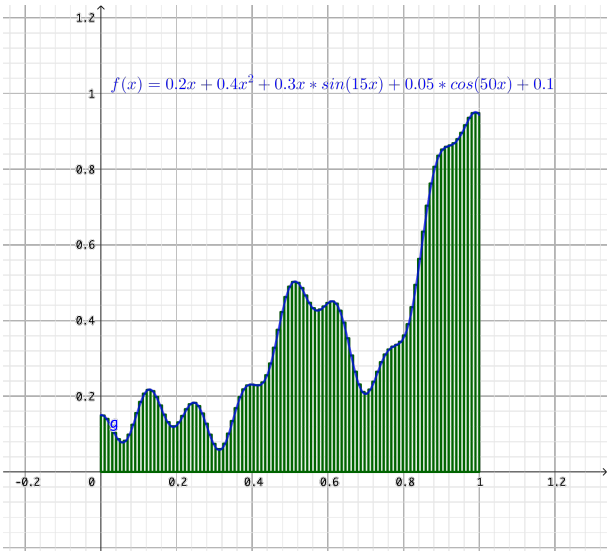


Figure 19: Approximation (N=100)

The figure above indicates, that by increasing the number of coupled nodes and therefore increasing the number sub-intervals, the function f can approximate with any desired accuracy.

3.2 Two-dimensional shallow-neural-network

For functions with more than one variable, added input-nodes are required. For an additional example, the task of a function of the form $f(x, y)$ will be explained by describing the fundamental steps of a 2-dimensional input for neural networks:

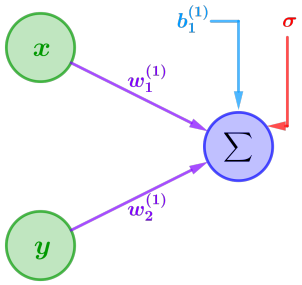
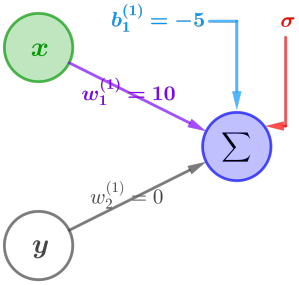


Figure 20: 2-dimensional input

Therefore several parameters, which impact the output of the hidden-nodes are displayed. For the beginning, the path of the top node is examined:



The output of the top hidden-node gives the shifted "3D-Sigmoid-Function" in x-direction, which is illustrated in the figure below:

Figure 21: 2-dimensional input

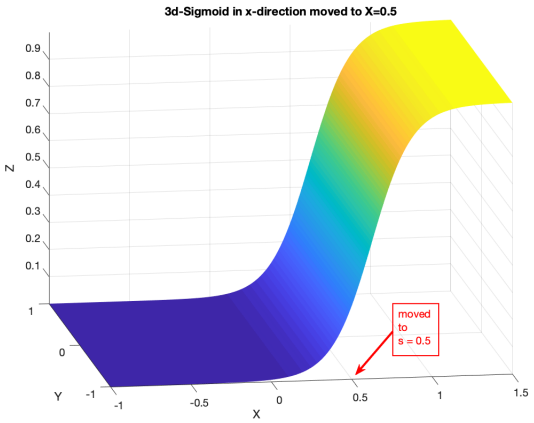


Figure 22: Sigmoid 3D Graph

Similar to the one-dimensional case a step-function is generated by increasing the weight $w_1^{(1)}$ and increasing the weight $w_2^{(1)}$ afterwards and varying the bias $b_k^{(1)}$ by the formula $\boxed{s = \frac{-b}{w}}$. For this example the step-location occurs at $s = 0.2$, which is illustrated in the next figures:

x-direction:

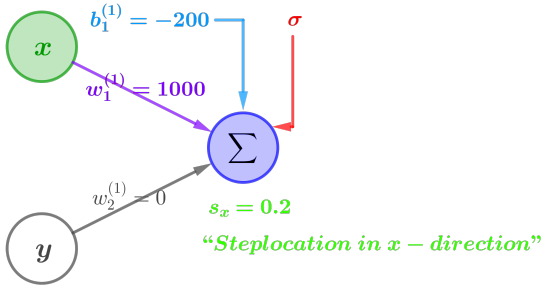


Figure 23: Nodes for x-direction

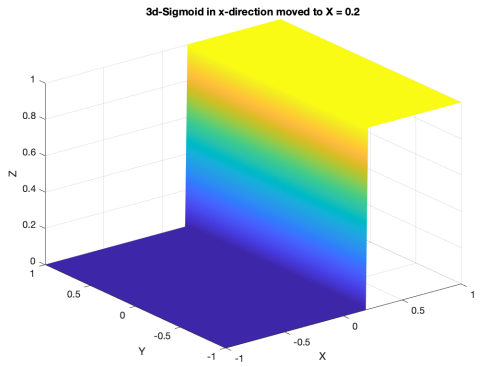


Figure 24: Step-function in x-direction

y-direction:

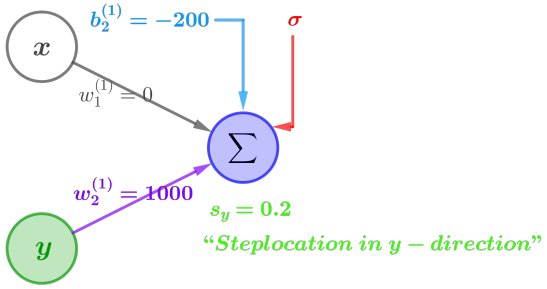


Figure 25: Nodes for y-direction

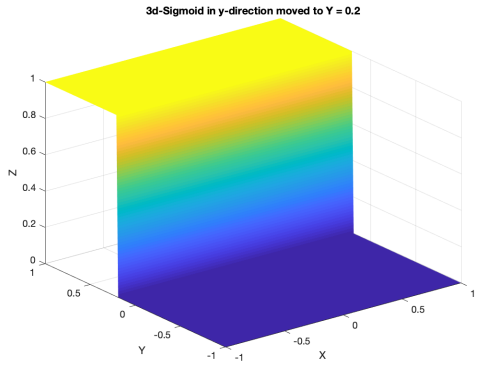


Figure 26: Step-function in y-direction

Now the *wave* is generated, relating to the one-dimensional case, by adding one more hidden-node to the first hidden-layer.

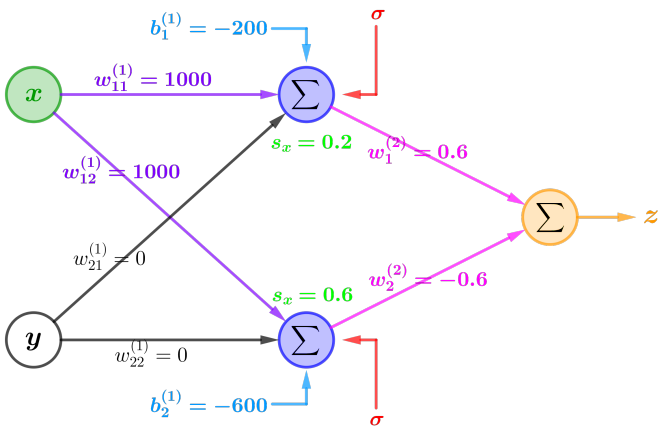


Figure 27: Generates wave in x-direction

Therefore the output of the hidden-node is scaled with the weight $w_k^{(2)}$, which is given on several step-points (in this example from $s_1 = 0.2$ to $s_2 = 0.6$).
For these cases the overall output scales with the weight $w_k^{(2)} = \pm 0.6$ (height of wave) and in similar manner in y-direction. The following two figures show both waves in each direction:

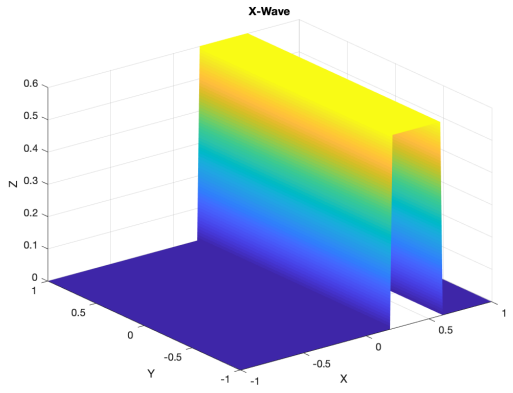


Figure 28: "Wave" 3D graph x-direction

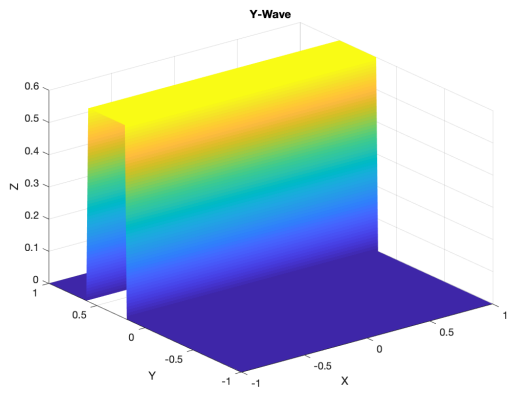


Figure 29: "Wave" 3D graph y-direction

Subsequently the two waves are added up to get a construct, that is similar to a tower-function with surrounding plateaus:

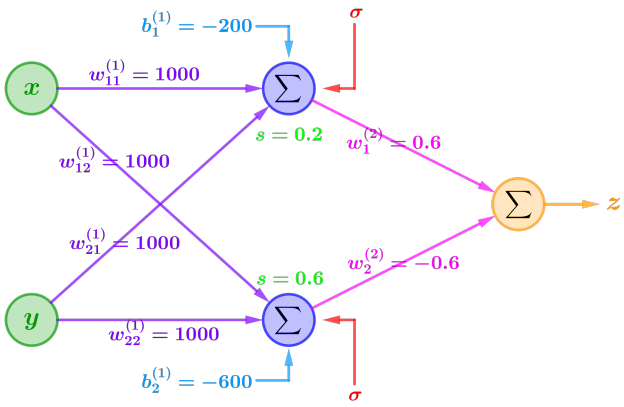


Figure 30: Nodes for both directions with same step-locations for each direction

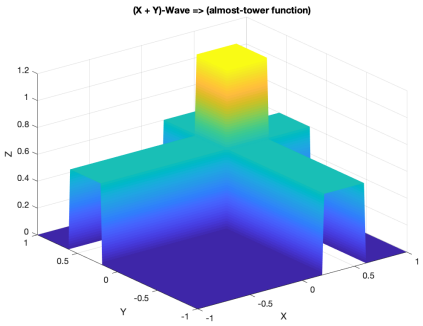


Figure 31: "Tower"-function with surrounding plateaus

To really get a tower-function, the weights $w_k^{(2)}$ have to be increased. Also another bias $b_k^{(2)}$ and activation-function (in this case sigmoid-function) must be added to the output-layer to achieve this goal. In this example $w_k^{(2)} = 10$ and the bias $b_k^{(2)}$ varies according to the formula $\boxed{b = \frac{-3w}{2}}$. Therefore it amounts to $b_k^{(2)} = -15$. This causes the surrounding plateau to almost reach zero.
The neural network with all noted parameters to get a tower-function in any position is shown in the next two figures.

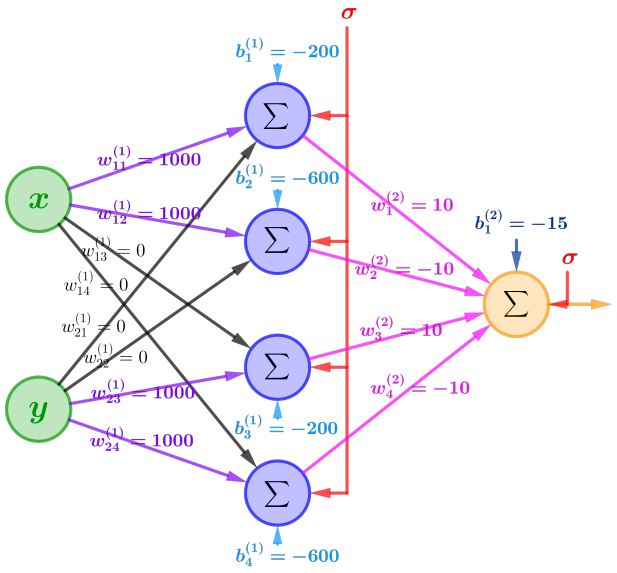


Figure 32: Nodes to compute a Tower-function at any Position

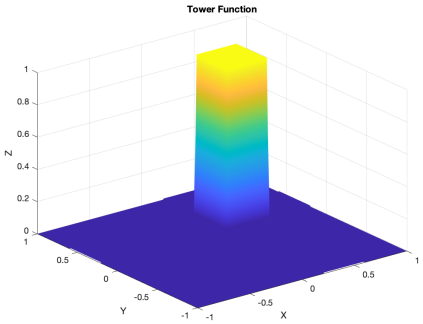


Figure 33: "Tower"-function

To arrange the tower functions hight, one more hidden-layer must be added. According to Cybenko's theorem, this isn't required, since one hidden-layer should be sufficient. Regardless of this, the neural network must be extended for an appropriate approximation of the towers. But it is important to remember, that it is also possible to acquire this with a shallow-network.
By attaching an additional hidden-layer, the height of the tower-function can be scaled by the weights $w_k^{(3)}$. For simplicity, the hidden-nodes of the first hidden-layer and the corresponding biases are not considered and only the several step-locations of each hidden-node in the first hidden-layer are displayed. Nevertheless, the weights $w_k^{(2)}$ are increased to a high value of positive and negative *coupled nodes* and the corresponding bias $b_k(2)$ of the second hidden-layer is varied, so that the formula $b = \frac{-3w_k^{(2)}}{2}$ holds. In this instance, $w_k^{(2)} = \pm 10$ and $b_k^{(3)} = -15$ should be enough to generate a tower-function. Furthermore one more tower-function is added, to show that there can be various of these towers in any desired position.

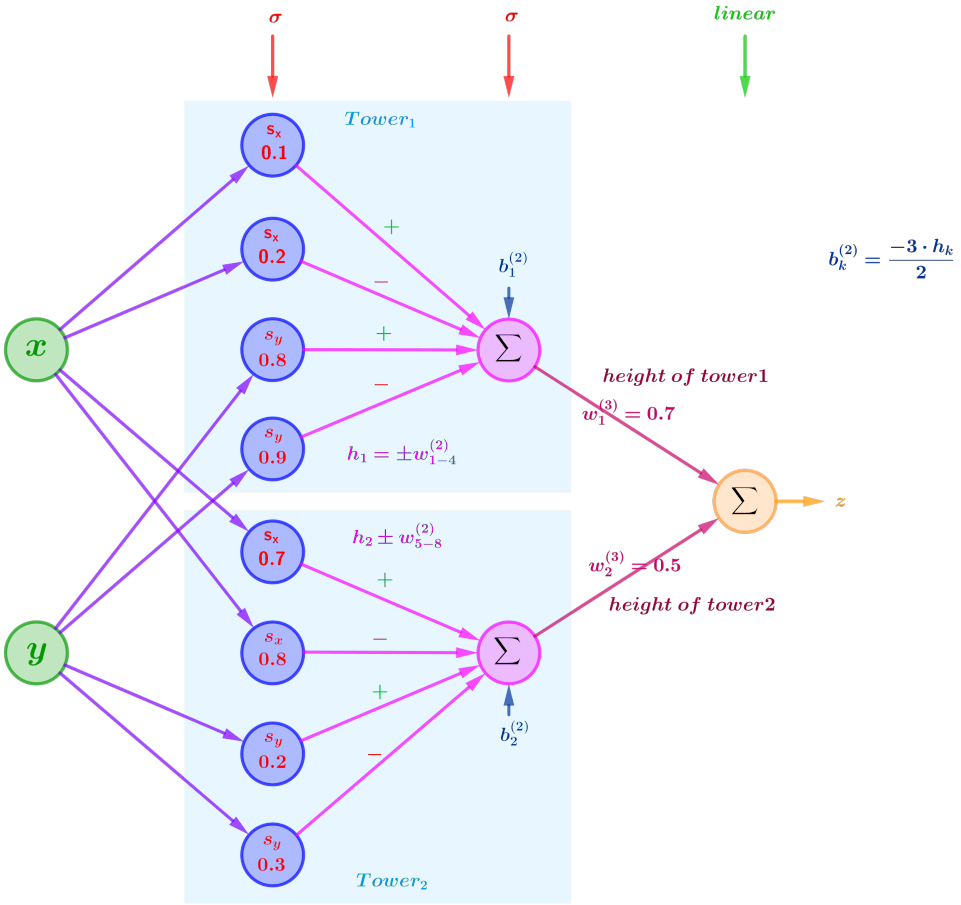


Figure 34: Neural Network with two towers in several positions

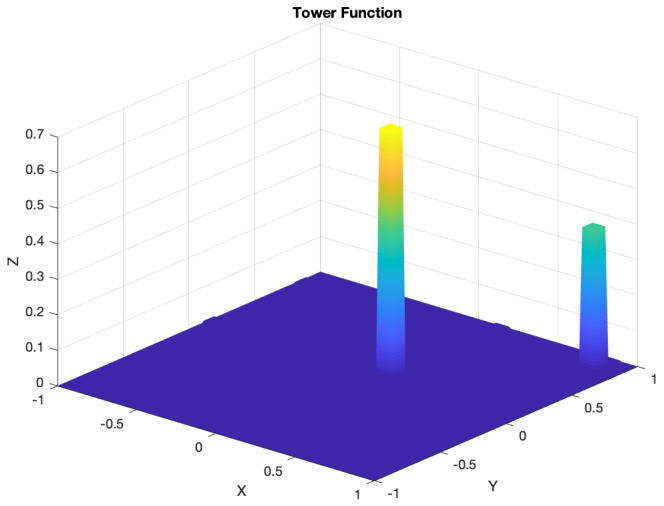


Figure 35: Two towers with several heights

In conclusion, the network can be expanded by continuously adding tower-functions, which can provide an approximation of a 3d-function of the form f(x,y)=z with two inputs. To show this principle, the following example function is reviewed:

$$f(x, y) = 0.2 * sin(10x) + 0.5 * cos(4y) + 0.5.$$

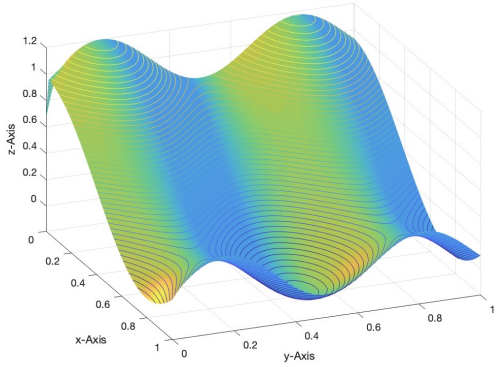


Figure 36: target-function

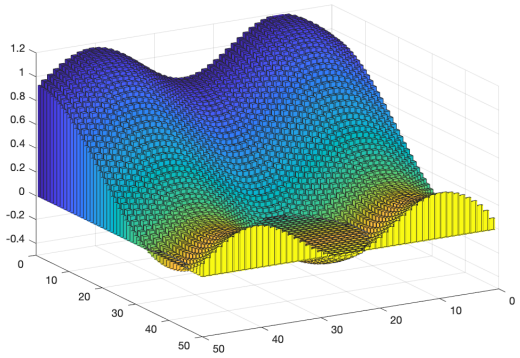


Figure 37: approximation with tower-functions

The visual proof above is not yet phrased for shallow-neural-networks with two inputs. This will be shown next.

A multi-layer-perceptron finds a "complex" decision boundary over the space of reals. To find these boundaries the preceptrons of each hidden-node is focused. A perceptron operates on real-valued vectors as shown in the next three figures.

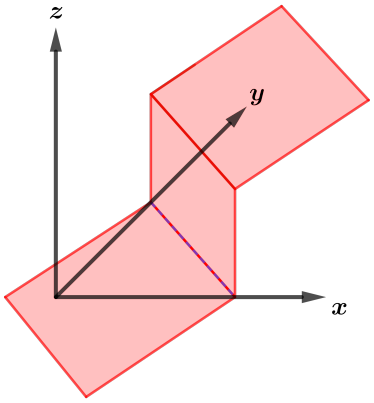


Figure 38: step-function

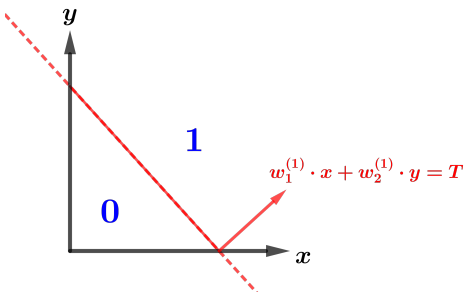


Figure 39: threshold for vector-classification

$$z = \begin{cases} 1 & if \quad \sum_k^N W_k^{(1)} x_k \geq T \\ 0 & else \end{cases}$$

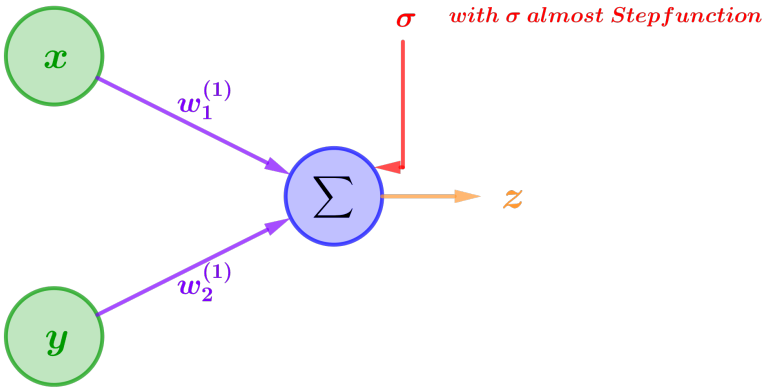


Figure 40: Multilayer perceptron "fire" for the threshold T

For example look at the network to get five of these boundaries.

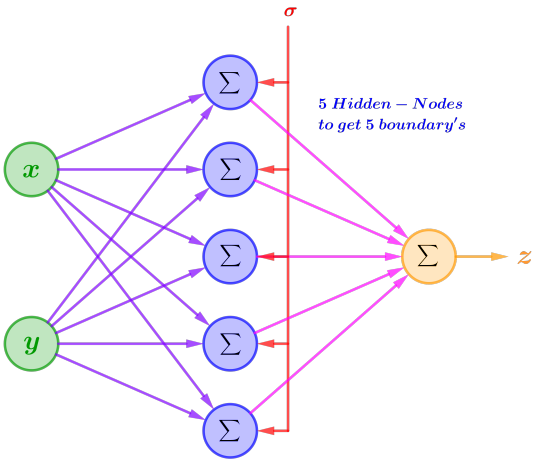


Figure 41: 5 hidden-nodes to get 5 boundaries

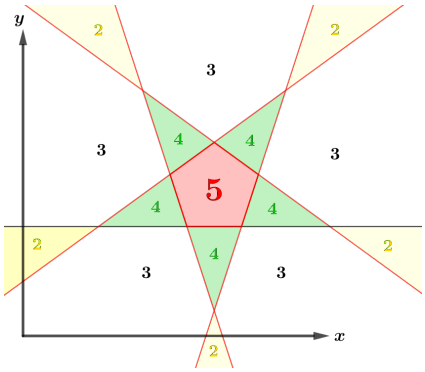


Figure 42: 5 boundaries with a pentagon in the middle

Therefore this will generate the following output y.

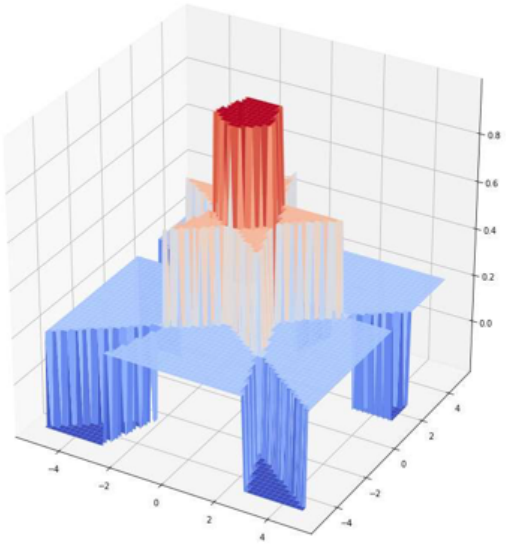


Figure 43: output with 5 boundaries - likely "pentagon-prism" with surrounding plateau [2]

The previous problems still occur, because the focus shifted to a network with boundaries, which should only "fire", if the inputs are in the area of the pentagon in the middle. But at this point the network also "fires" for the surrounding plateaus. This is why the other sums should be droped.

This can almost be achieved by increasing the number of hidden-nodes. Because by increasing the sides (boundaries), the area outside of the polygon will be reduced and therefor also the surrounding plateaus. The polygon in the middle will nearly become a circle. The output of the neural network will become likely the figure below.

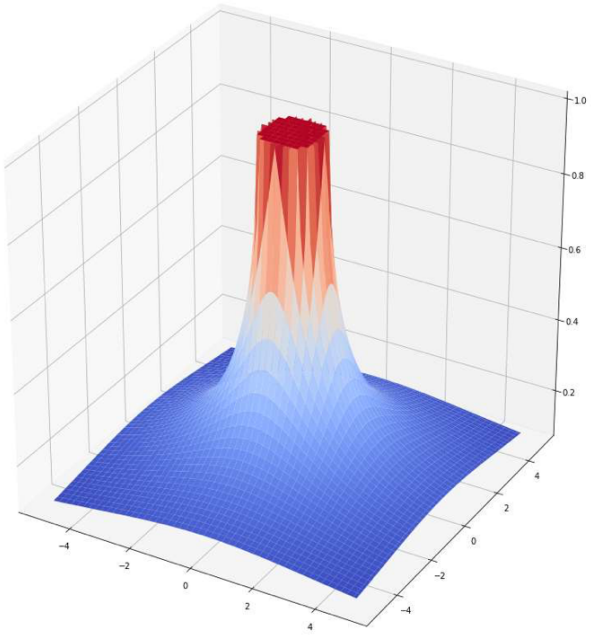


Figure 44: 1000-sides polygon [2]

For a small radius the output produces a nearly perfect cylinder at any location.

$$\sum_k y_k = N(1 - \frac{1}{\pi} arccos(min(1, \frac{radius}{|x - center|})))$$

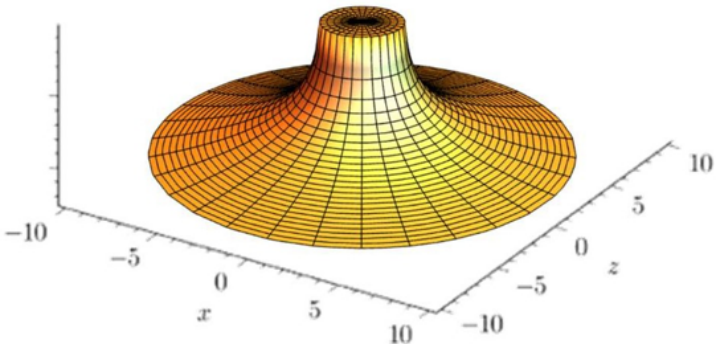


Figure 45: Cylinder Base [2]

By increasing the boundaries the output will get almost a cylinder

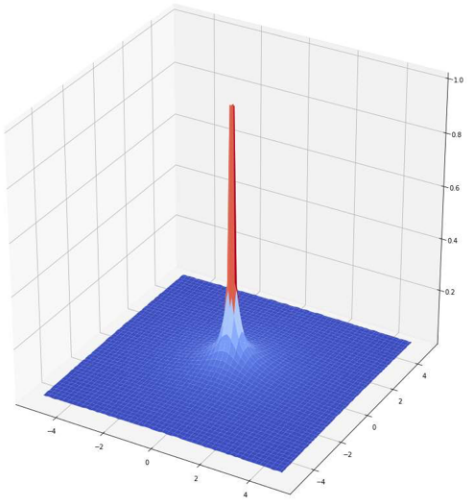


Figure 46: High approximation of the cylinder [2]

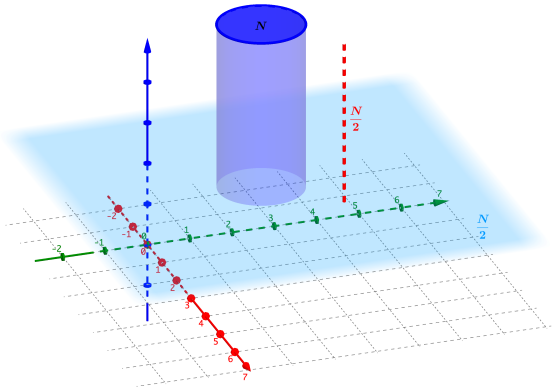


Figure 47: Not grounded

The sum of each cylinder is N inside and N/2 outside.

By adding a bias $b = -\frac{N}{2}$ to the hidden-nodes, the sum of each cylinder will get to $N/2$ inside and 0 outside almost everywhere.

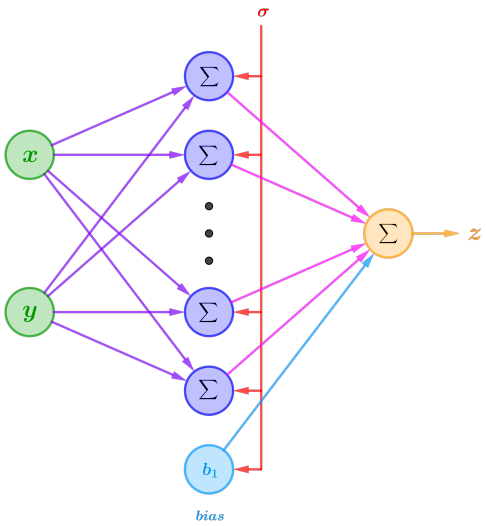


Figure 48: Adding a bias b to the Output

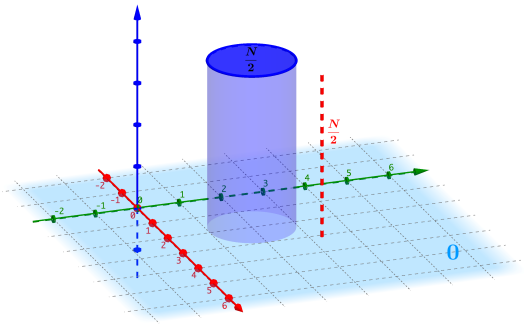


Figure 49: Grounded

Numerous of these cylinders with close height to the output value, given by $w_k^{(2)}$, will

yield almost the same result as shown before and approximate any continuous function of the form $f(x, y)$ with only one hidden-layer.

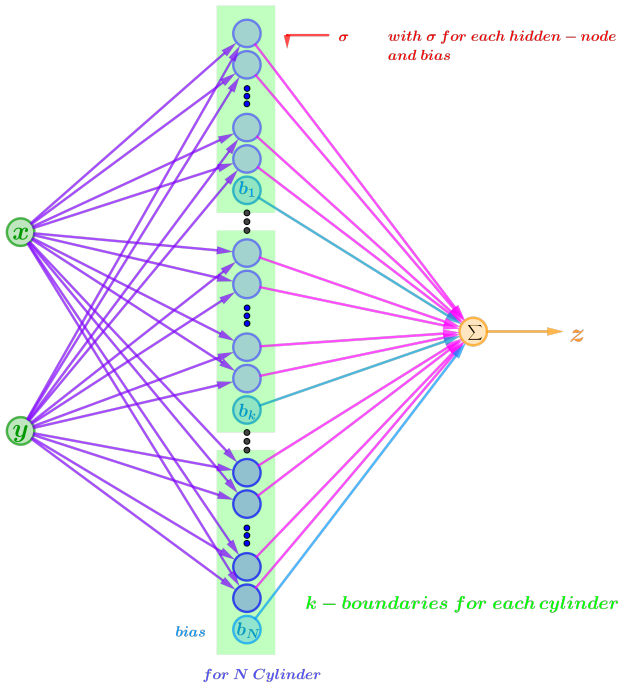


Figure 50: neural network with N cylinder

For example, the function $f(x, y) = 0.2sin(10x) + 0.5cos(4y) + 0.5$ is approximated by a large number of cylinders, which is shown in the two next figures.

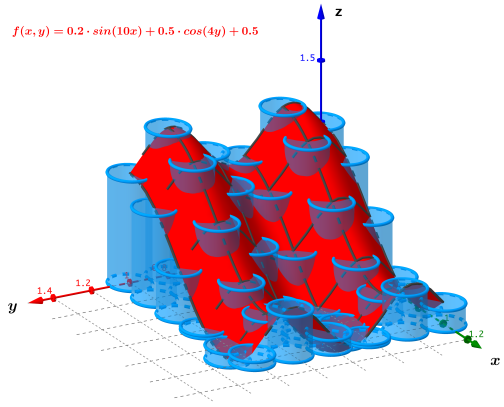


Figure 51: Cylinder-Approximation

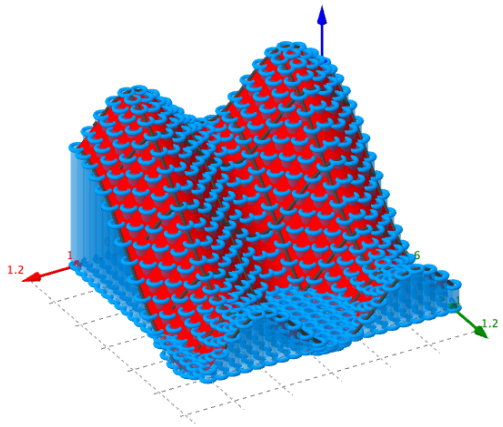


Figure 52: Increasing the number of cylinder

Overall the number of hidden-nodes for this shallow-network will increase enormously in comparison to the number of nodes for the neural network with two hidden-layer revealed earlier.
Analog to the one and two-dimensional input examples the identical approaches extend to higher-dimensional data and hence it confirms, that Cybenko's Theorem operates, for higher-dimensional functions.
Just a single-layer network may demand an exponentially large number of nodes to estimate a function with N inputs. Deeper networks may need fewer neurons than shallow to represent an identical function because the required number of nodes will only increase linearly.
Another piece of important information is that the number of nodes can reduce even more if it involves irrelevant data. This statement manifested itself in the last example for one-dimensional inputs, which describes why *coupled nodes*, as seen above, may be unnecessary to reach an acceptable accuracy for the approximation of a given function.

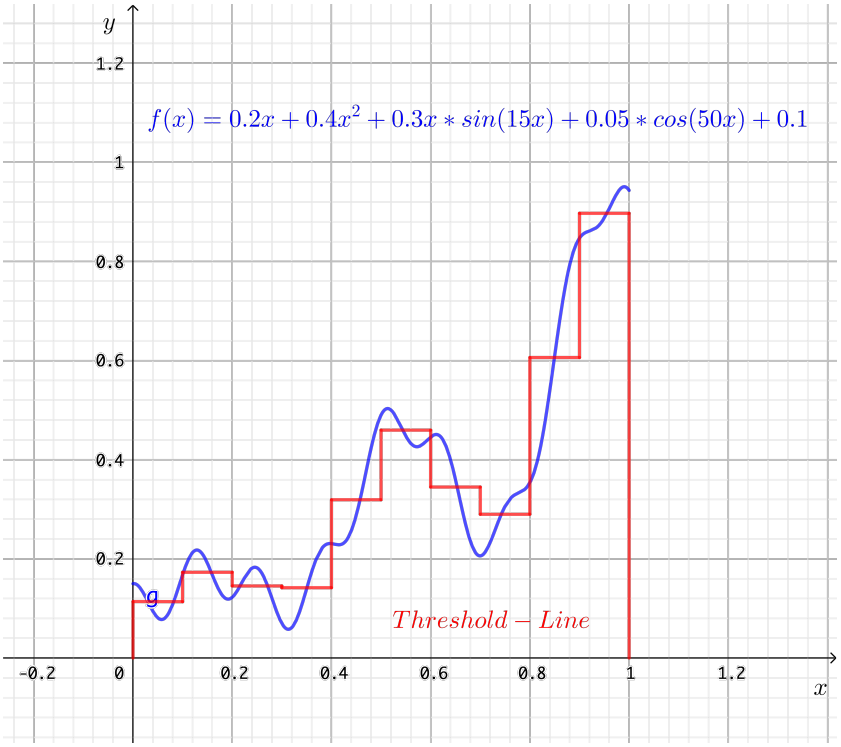


Figure 53: Threshold-line for redundant information $N = 10$

4 Implementation in Matlab

After establishing some basis for the topic of artificial neural networks, this knowledge can be applied to formulate the back-propagation explained in chapter one. During the following, Matlab's function system is utilized to generate a shallow network, that achieves the approximation described in Cybenko's theorem. The following five code-script creates this artificial neural network:

TestCybenko.m

```
% HUB for every possible Input.
clear all;
close all;

dim = 1;                                  % Function depended
alpha = 0.001;                            % Learning rate
interval = 500;                           % Iteration within the unit
i = 10000;                                % Repetition of the learning process
HiddenNodes = 2 * interval^dim;           % Nodes for Shallow-Learning


D = zeros(interval, 1);                   % Solution-Matrix
X = gridX(interval, dim);
N = interval^dim;
X(:, dim + 1) = ones(N, 1);               % Adds the bias into the weights


for  k = 1 : N
   D(k, 1) = Func(X(k, :));               % Inscribing all solutions in D
end


W1 = 2 * rand(HiddenNodes, dim) - 1;
W2 = 2 * rand(1, HiddenNodes) - 1;        % Implementing random start-weights.

W1(:, dim + 1) = ones(HiddenNodes, 1);    % Adding bias in weight-matrix W1


                                          % Learning Process
for  epoch = 1 : i
   [W1, W2, H, es1] =
Backprop(W1, W2, X, D, N, alpha);         % Backpropagation explained in
end                                       Chapter 1
```

Backprop.m

```
function [W1,W2,H,es1] = Backprop(W1,W2,X,D,N,alpha);
 H = zeros(N,1);                          % H and es1 are for hypothetical plotting
 es1 = 0;

 for k = 1:N
  x = X(k,:)';
  d = D(k,1);                             % Back-propagation
  v1 = W1*x;
  y1 = Sigmoid(v1);

  v2 = W2*y1;                             % Linear output
  y = v2;

  e = d - y;                              % Using the generalized delta-rule to correct the weights
  delta = e;                              % Cross-entropy

  e1 = W2'*delta;
  delta1 = y1.*(1-y1).*e1;

  dW1 = alpha*delta1*x';
  W1 = W1 + dW1;

  dW2 = alpha*delta*y1';
  W2 = W2 + dW2;

  H(k,:) = y;                             % Inscribing error and solution for potential plotting.
  es1 = es1 + (d-y)^2;
 end
end
```

Func.m

```
function [y]=Func(x)
y=exp(x(1)-1);                            % The function that will be approximated
end
```

Sigmoid.m

```
function [y] = Sigmoid(x)
y = 1./(1+exp(-x));
end
```

gridX.m

```
function [X] = gridX(interval,n)
H = zeros(interval^n + 1, n);
H(1,:) = eye(1,n);
X = zeros(interval^n, n);
X1 =
(1.0/interval:1.0/interval:1.0)';
for k = 2:interval^n + 1
H(k,1) = H(k-1,1)+1;
for m = 2:n
if mod(H(k-
1,1),interval^(m - 1)) == 1
H(k,m) = H(k-1,m)+1;
else
H(k,m) = H(k-1,m);
end
end
end
for k = 1:interval^n + 1
H(k,1) = H(k,1)-1;
end
for k = 2:interval^n + 1
X(k-1,:) = H(k,:);
end
for k = 1:interval^n
for m = 1:n
if mod(X(k,m),interval)==0
X(k,m) = interval;
X(k,m) = X1(X(k,m),1);
else
X(k,m) = mod(X(k,m),interval);
X(k,m) = X1(X(k,m),1);
end
end
end
end
```

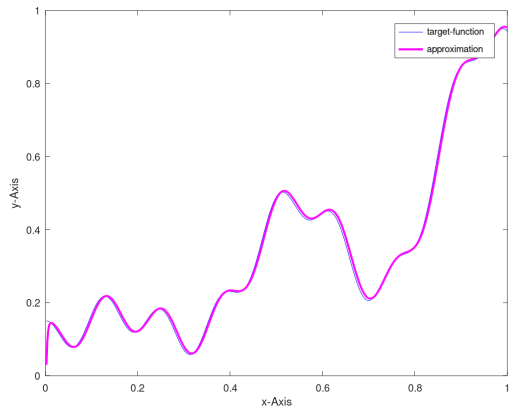
Instead of commenting gridX, it's easier to show an example with dim=2 and interval=4:

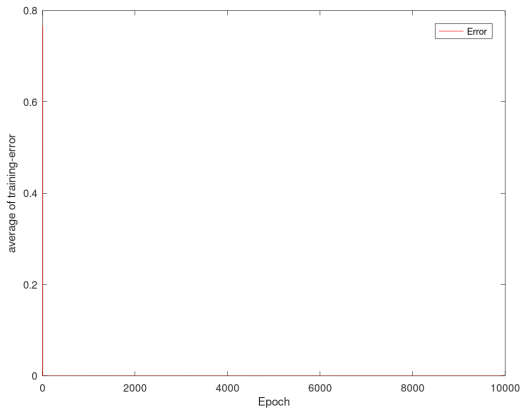
$$\text{gridX(4,2)=}\begin{pmatrix} 0.25 & 0.25 \\ 0.5 & 0.25 \\ 0.75 & 0.25 \\ 1.0 & 0.25 \\ 0.25 & 0.5 \\ \vdots & \vdots \\ 1.0 & 0.5 \\ 0.25 & 0.75 \\ \vdots & \vdots \\ 1.0 & 1.0 \end{pmatrix}$$

This code creates an artificial neural network, which approximates multi-variant functions to an error-tolerance between 10^{-3} and 10^{-4}.
The opening segment uses the following one-dimensional function and explains the occurrences within the first 10000 learning steps:

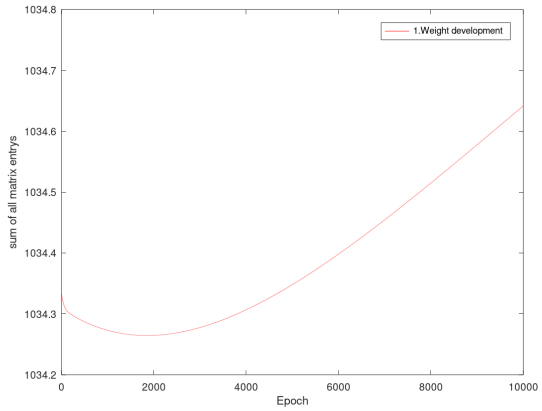
$$y = 0.2 \cdot x + 0.4 \cdot x^2 + 0.3 \cdot x \cdot sin(15 \cdot x) + 0.05 \cdot cos(50 \cdot x) + 0.1$$

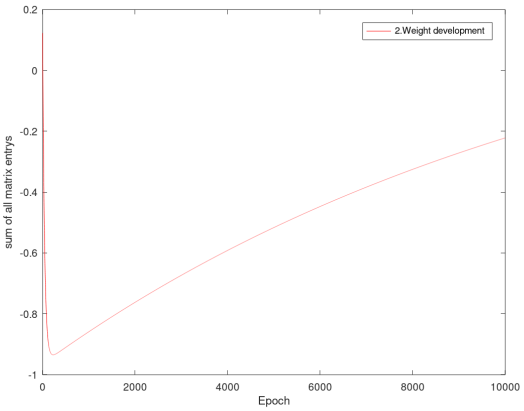
The left figure represents the learned function with $alpha = 0.001$ and $interval = 500$ for $epoch = 10000$. The fundamental difference to the target occurs within direct surrounding to $x = 0$, where the approximation fails, to visually overlap the target-function. With more learning steps, this error gets marginally smaller but has its limits for accuracy. In the right picture is the behavior of the mean error. The fastest learning part is within the first 100 learning iterations, whereas after $epoch = 1000$, detecting a visual change is almost impossible.





The following graphs display the sum of entries for the weight-matrices, which for the first weight includes the bias on the hidden layer. In the second diagram, the weight-matrices make a rapid change within the first 100 learning steps that sometimes occur even during later stages (as seen in the two-dimensional function later in this chapter). The error stays between 10^{-3} and $5 \cdot 10^{-4}$ within the last 1000 learning iterations. The entire change in the sum of weight-matrix entries amounts to 0.1 for 500-1000 matrix entries. Thus there is a small mean difference per weight-entry during the last 1000 learning steps.

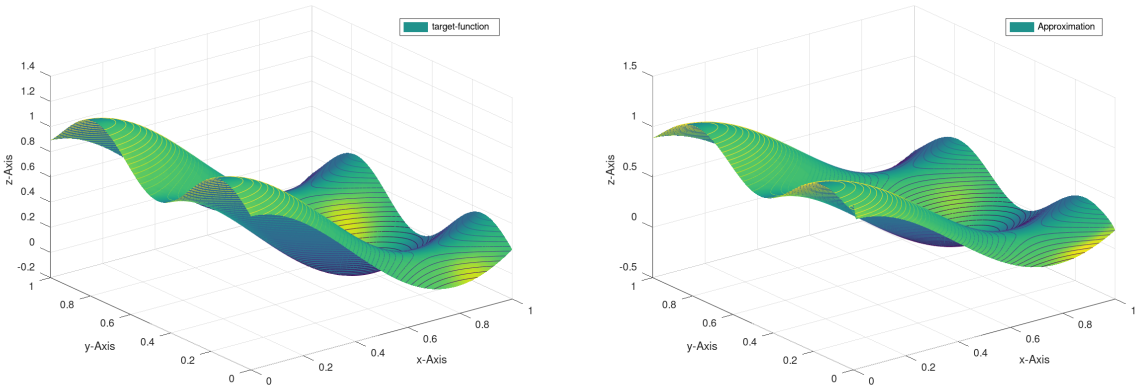




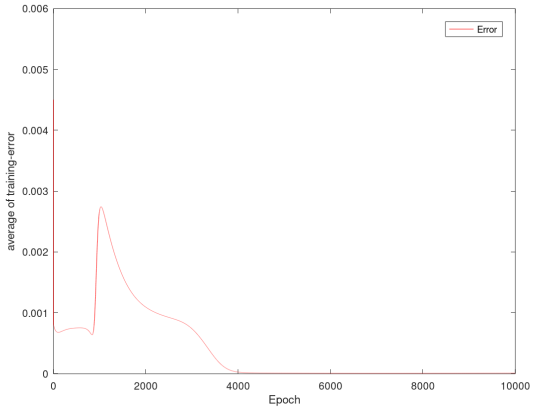
The real difficulty occurs for higher-dimensional approximations and therefore, already at two dimensions, which will be further analyzed using the following function:

$$z = 0.2 \cdot sin(10 \cdot x) + 0.5 \cdot cos(4 \cdot y) + 0.5$$

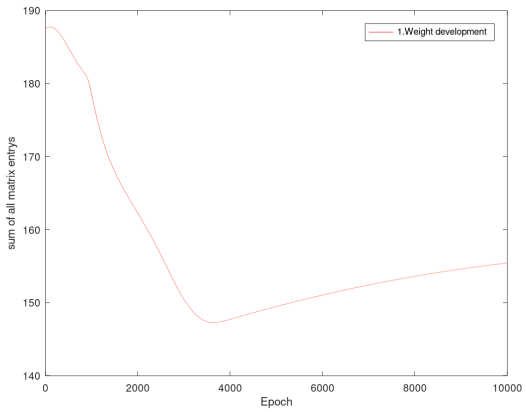
On the left is the target function and on the right the approximation with $alpha = 0.005$ and $interval = 100$ for $epoch = 10000$. The reoccurring issue in proximity to $(x, y) = (0, 0)$ remains unchanged, as expected, but the extrema won't converge as efficiently as for the one dimensional input.

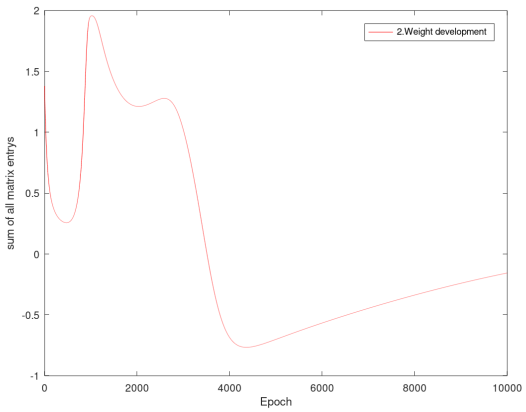


The mean error - although reaching 10^{-4} - only very slowly declines in time and even jumps to higher errors within the first 2000 learning steps. This behavior of the error should not happen, but there exist counter-measurements, that would need more learning steps and maybe a change in the learning rate. Adding more nodes or implementing deep learning could also improve the approximation. Although this shows a feasible 2-dimensional-functions approximation with this shallow network, its flaw lies within the increase in time needed to achieve visually appealing results and the low accuracy.



The following two pictures describe the weights, but this time it's obvious, they are converging to a fixed point. The waves created in the error-function are visible as well.

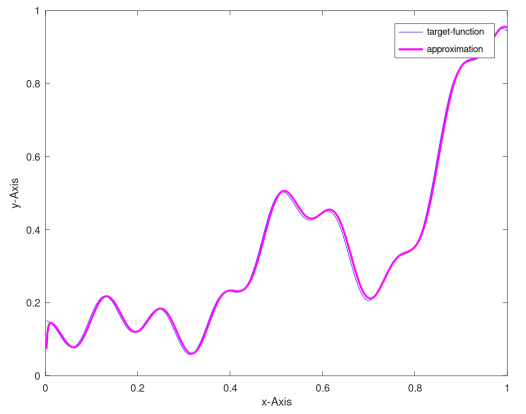


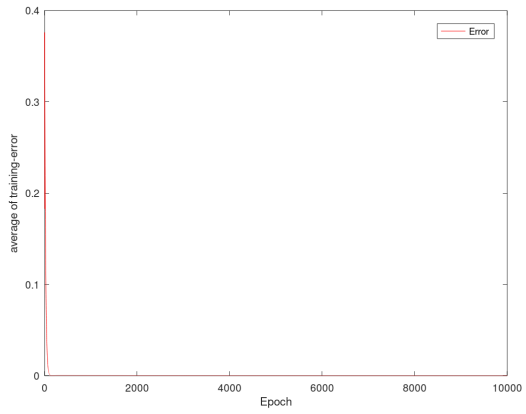


Although this code is fast, it represents one of the least accurate results, because the node-count for the calculated functions is too low. As explained in chapter 3, the optimal number of nodes grows almost exponentially with the dimensions of the input data. The "cylinder-tower-functions" - in the 2-dimensional case - will only work to a specific accuracy for the given parameters and adding more nodes will slow down the process, but achieve same results with a smaller amount of learning steps. The following part reviews the previous functions with slight changes in settings, to demonstrate the effects of a higher node-count with changed learning-rate:

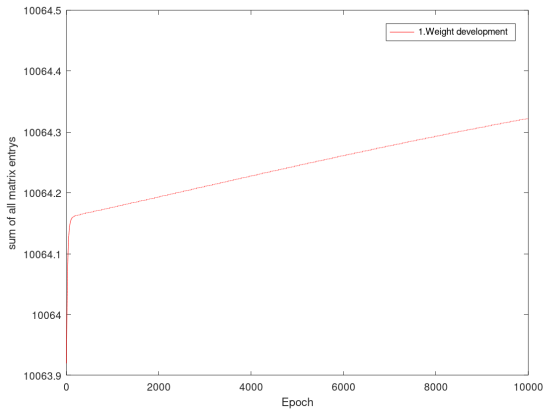
One-Dimensional-Function:

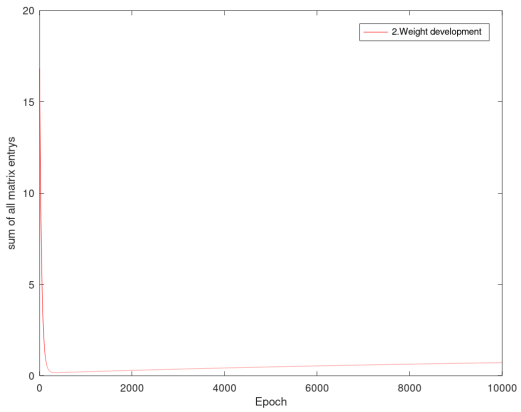
The nodes are increased by the factor 10, and the replaced parameter is $alpha = 0.0001$. For a higher node-count, the results are better, and the mean error reduced by a factor 0.1.



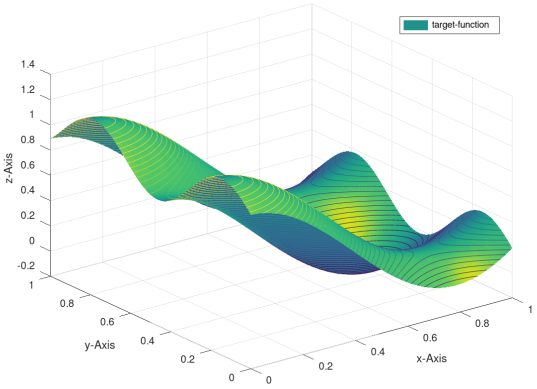


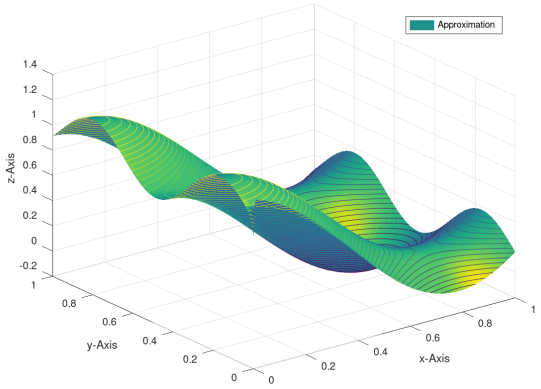
The real changes appear in the weights. Instead of the slow approximations over time, the graphs change directions within the first 100 learning steps and continue to improve marginally.



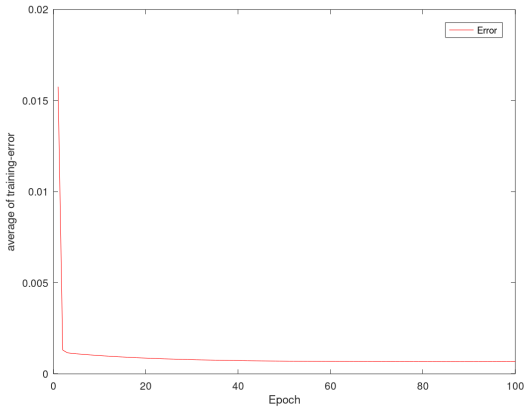


Taking the same configurations as for the one-dimensional function, this code achieves a similar result with 1/100 of the learning steps.

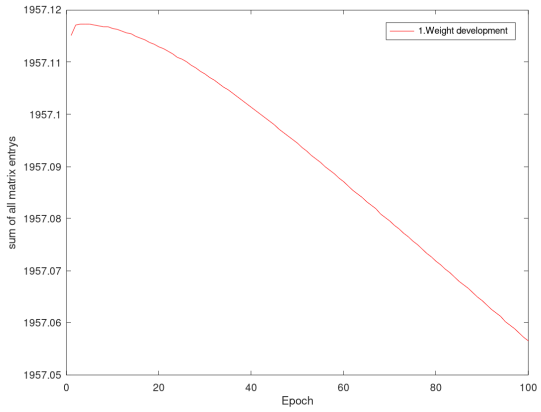


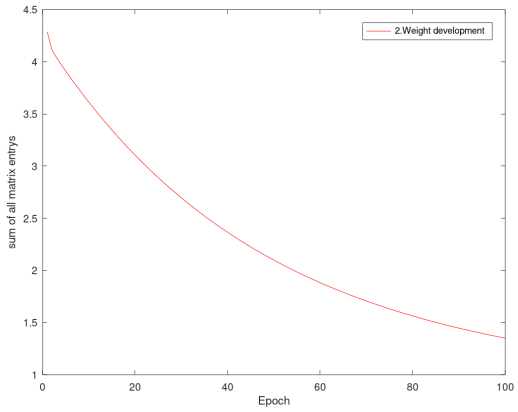


The error also converges faster than before within the same amount of learning steps.



The weights show similar behavior as the first weights did in the last 100 steps.





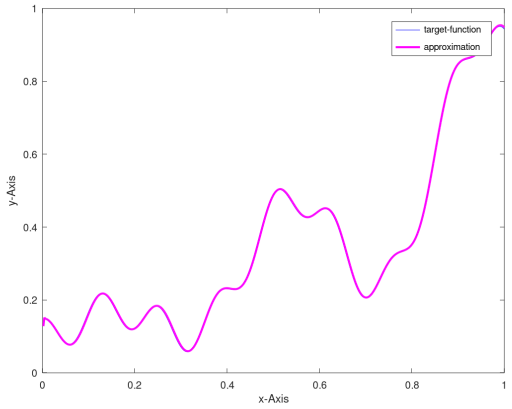
Although the improvement of the results for one-dimensional functions is rather slim, the effects of more nodes are remarkable in higher dimensions. Nevertheless, the computation time increases exponentially related to the number of input data, and it is still not viable for complex systems with multiple inputs.

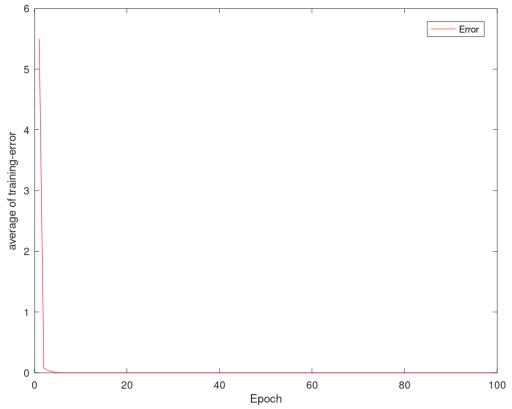
5 Excursion/Outlook: Deep Learning

The changes in node-count increased the run-time of the code, thus having higher dimensions then 2 proposes a new problem, not about the accuracy, but run-time. While preserving the small mean error, the hidden layers must expand, and therefore deep-learning replaces shallow-learning. Although deep-learning is not the main topic of this paper, its importance in artificial neural networks is remarkable. As the realization of one of numerous follows up on Cybenko's theorem, this is manageable through manipulation of the former program.
The first change is the addition of another weight-matrix - with matching dimensions - between the input and output on the same base of chapter 1 (back-propagation) and applying the sigmoid function to the created second layer. The new weight-matrix demands an adjustment of the back-propagation algorithm. For simplicity, the addition of the new bias vector happens separately from the weight and needs consideration in the back-propagation as well.
For the approximation of the one- and two-dimensional function from the previous chapter, two hidden-layers are sufficient to achieve satisfying results:

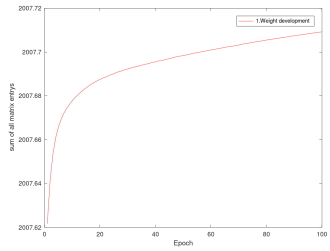
One Dimensional Function:

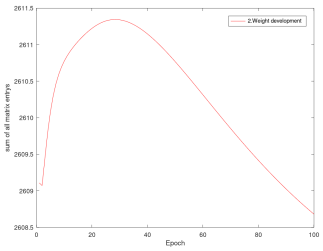
The following pictures show the results of the deep-learning algorithm with said changes. The new error is 0.000012168 after only 100 learning iterations. As the left figure shows, the approximation close to $x = 0$ has improved drastically.

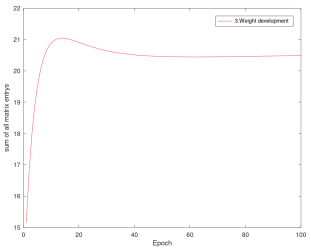




The next pictures show the weights with their bias, as explained in the previous chapter. Even though there are only 100 learning iterations, the graphs indicate convergence.

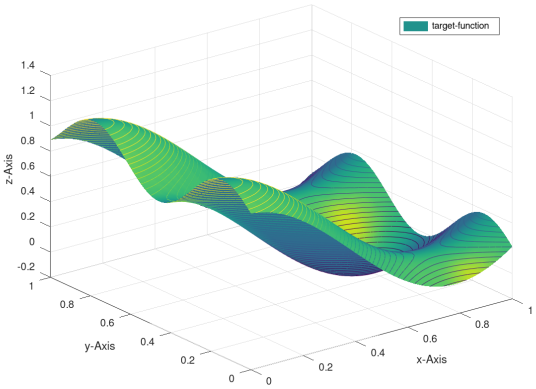


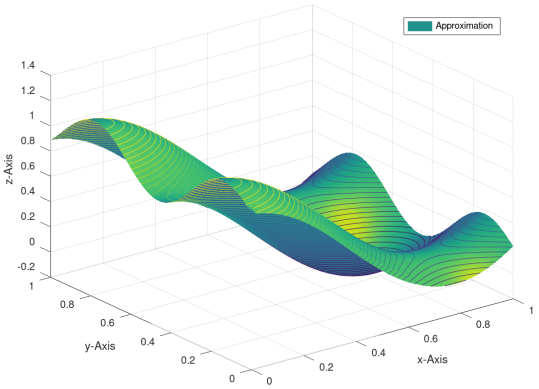




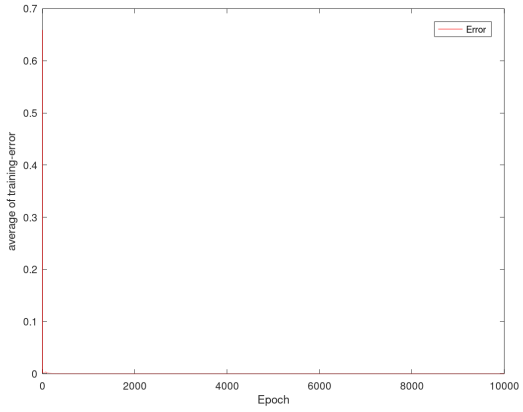
Two Dimensional Function:

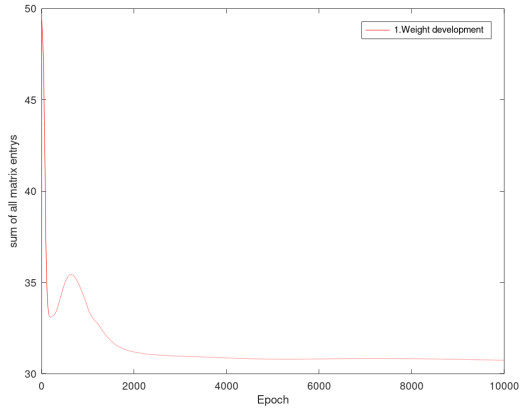
With the appropriate settings the code achieves an error of 0.0000033636 for 10000 learning iterations and the following graphs.

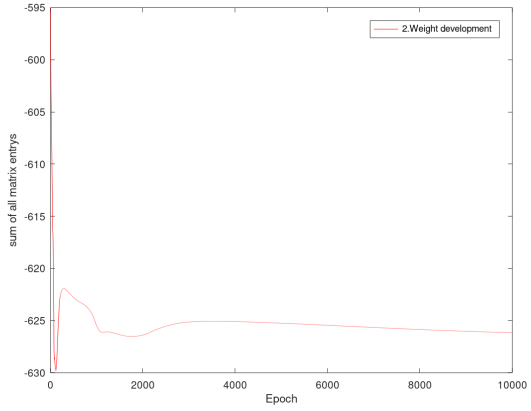


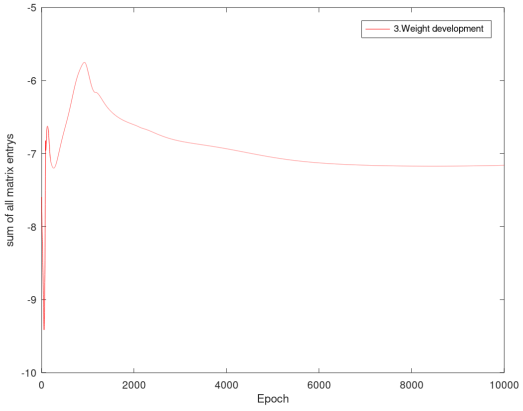


The error, as well as the weights, converge after 5000 learning iterations.









6 Conclusion

This paper demonstrated the basics for back-propagation, illustrated a visual proof of Cybenko's Theorem, and concluded with the implementation of a functioning system in Matlab. The question asked within this paper was, whether a shallow network satisfying Cybenko's-Theorem, can approximate any multi-variant continuous function to a certain error-tolerance. The answer - as shown in chapter 3 - is theoretically yes, but since with higher precision, the required node-count, especially for more than two inputs, grows exponentially. Therefore it becomes practically inefficient to reach a small error within a reasonable run-time.
Later developments in the field of machine-learning solved this problem, as pointed out in chapter 4 by the improvements of deep learning. Shallow networks makes an approximation to some degree possible, but they won't be enough for voice-,image-recognition or any assistance that comes with numerous inputs and configurations, while still demanding precision, and durability. The advantage of deep learning is that the required nodes only grow linear instead of exponential in the shallow-network.
Cybenko's theorem is the founding stone for what, we use today and reveals itself as the predecessor to nowadays technology, that gave rise to more and better opportunities.

7 References

[1] Barron A. R., *Approximation and estimation bounds for artificial neural networks*, Machine Learning 14, San Mateo, 1994, pg. 115–133

[2] Bhiksha Raj, *11-785 Introduction to Deep Learning - Lecture 2*, Carnegie Mellon University, Pittsburgh, 2019
Retrieved from http://deeplearning.cs.cmu.edu/ (23.09.2019)

[3] Cheney Ward, Light Will, *A Course in Approximation Theory*, Brooks/Cole, 2000

[4] Cybenko G., *Approximation by Superposition of a Sigmoidal Function*, Mathematics of Control, Signals, and Systems, Springer-Verlag New York Inc., 1989, pg. 303-314

[5] Hassoun Mohamad H., *Fundamentals of Artificial Neural Networks*, MIT Press, 1995

[6] Hornik Kurt, *Approximation Capabilities of Multilayer Feedforward Networks*, Neural Networks, vol.4, Wien, 1991

[7] Kim Phil, *MATLAB Deep Learning: With Machine Learning, Neural Networks and Intelligence*, Apress, Seoul, 2017

[8] Nielsen Michael A., *Neural Networks and Deep Learning*, Determination Press, 2015
Retrieved from http://neuralnetworksanddeeplearning.com/chap4.html (23.09.2019)