AI Artificial Neuron

Gonzalo Reynaga García
September, 2024

This document presents a new neuron model, conceptualizing the perceptron as a collection of these neurons. The explanation is structured step-by-step to improve understanding of AI learning algorithms used in data processing and to clarify key components like the activation function.

The dataset used to test was the MNIST dataset[1], in which consist in 60000 handwritten grayscale images 28x28(784 pixels).

Binary Neuron

In this neuron model, each neuron works as a small lookup table, where binary inputs correspond to a binary output (limited to values of either 0 or 1). Two types of neurons will be used: one with a single input and another with two inputs, with the outputs represented as p_i values.

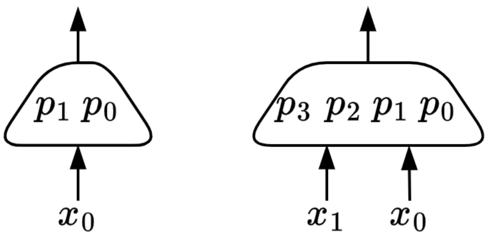


Figure 1. Binary Neurons

$$x_i \in \{0,1\},\ p_i \in \{0,1\}$$

$$\begin{array}{l} [x_0] \\ [1] \mapsto p_1 \\ [0] \mapsto p_0 \end{array} \qquad \begin{array}{l} [x_1\ x_0] \\ [1\ 1] \mapsto p_3 \\ [1\ 0] \mapsto p_2 \\ [0\ 1] \mapsto p_1 \\ [0\ 0] \mapsto p_0 \end{array}$$

The function of this lookup table is a $\mathrm{mod} 2$ algebra, the details are described in *Appendix A*.

Learning

When the cells are connected to other cells, multiple data paths are created depending on the input, resulting in a binary output that can be either correct or incorrect.

The learning rule for these cells is simple: if the output is correct, decrease the counter for that path in the cell; if the output is incorrect, increase the counter. When the counter reaches a certain threshold, the cell changes its output value for that path.

Thus, all the cells receive a global signal indicating whether the tree output is incorrect.

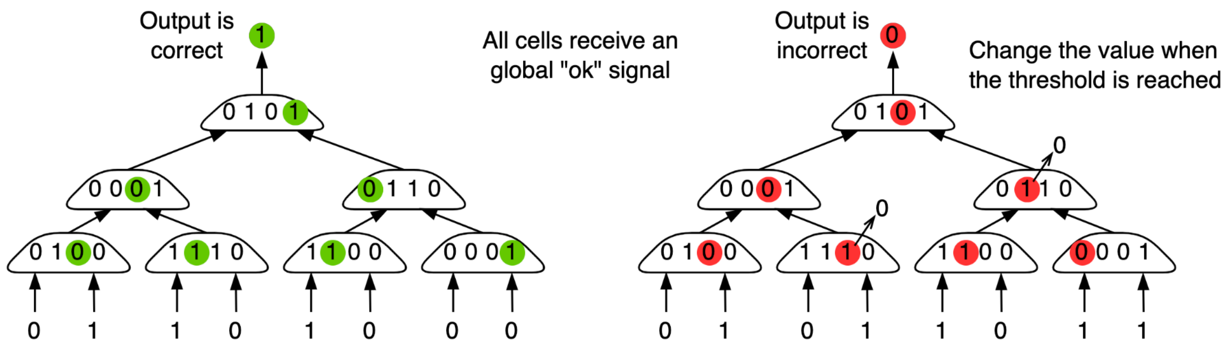


Figure 2. Learning

With this simple rule, the tree of cells can adapt to get closer to the desired output.

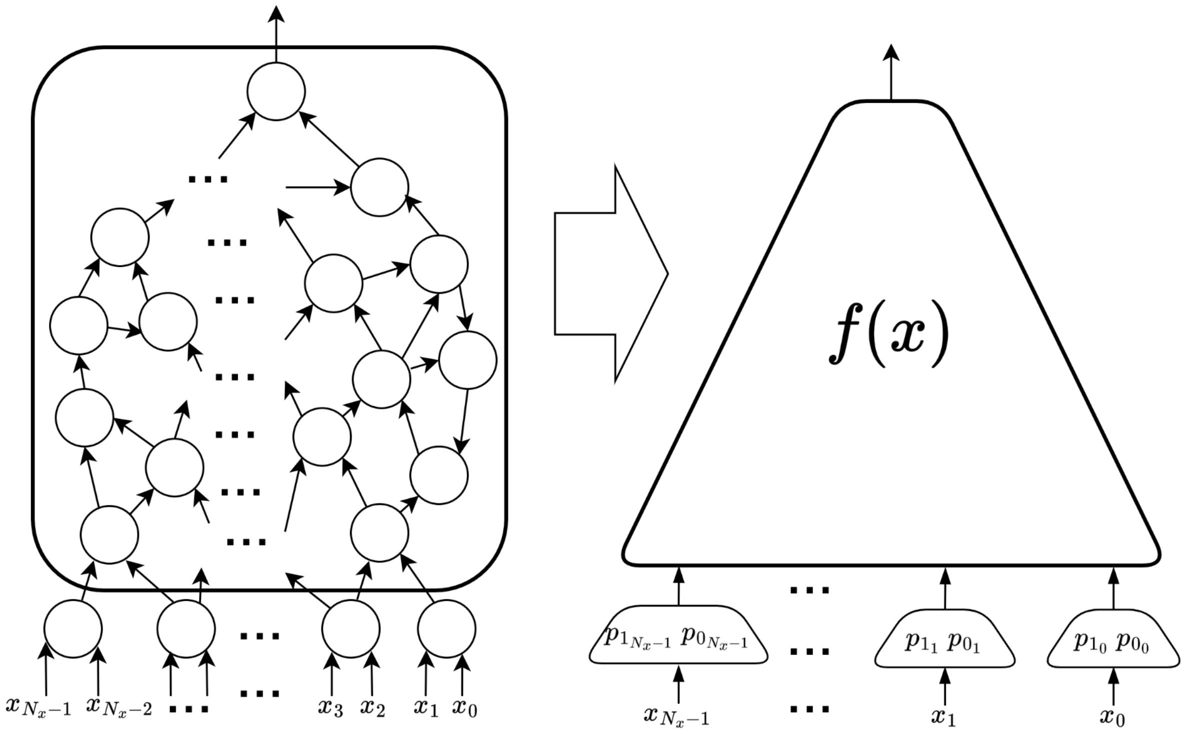
Testing this simple rule on MNIST dataset with 784 binary inputs (the images were converted to a binary format), to detect even numbers from odd numbers, it get an accuracy of 70~79% for 60000 samples with 1000 epochs.

Expanding this tree does not provide a notable increase in accuracy, and due to the slow testing process, it has not been analyzed further.

Simplified tree

In this next section, the tree structure is simplified to explore ways to combine it with other trees and enhance accuracy.

Biological neurons can connect randomly; we will model this behavior (ignoring possible loops) with our binary neurons. As discussed in the previous section, the neurons seem to approximate our desired value, depending on their inputs x_i.



x_{N_x-1} x_{N_x-2} ... x_3 x_2 x_1 x_0
$f(x)$
$p_{1_{N_x-1}}$ $p_{0_{N_x-1}}$
p_{1_1} p_{0_1}
p_{1_0} p_{0_0}
x_{N_x-1} x_1 x_0

Figure 3. Simplified tree

Therefore, all neurons create a function based on the inputs cells to achieve our desired output.
In our simplified model, we simplified the entry cells into a single input cell. The single input cells transforms the inputs before they enter the function as shown in *Figure 3*.
Based on a simple idea that similar inputs generate similar outputs, then we can use the algebra to sum all the transformed inputs and create two categories, where the sum indicates the degree of similarity among the inputs.

$$z = \frac{1}{N_x} \sum_{i=0}^{N_x-1} \begin{cases} p_{1_i} & \text{if } x_i = 1 \\ p_{0_i} & \text{if } x_i = 0 \end{cases} \tag{1}$$

$$f(z) = \begin{cases} 0 & \text{if } z \leq 0.5 \\ 1 & \text{if } z > 0.5 \end{cases} \tag{2}$$

Where N_x is the number of inputs.

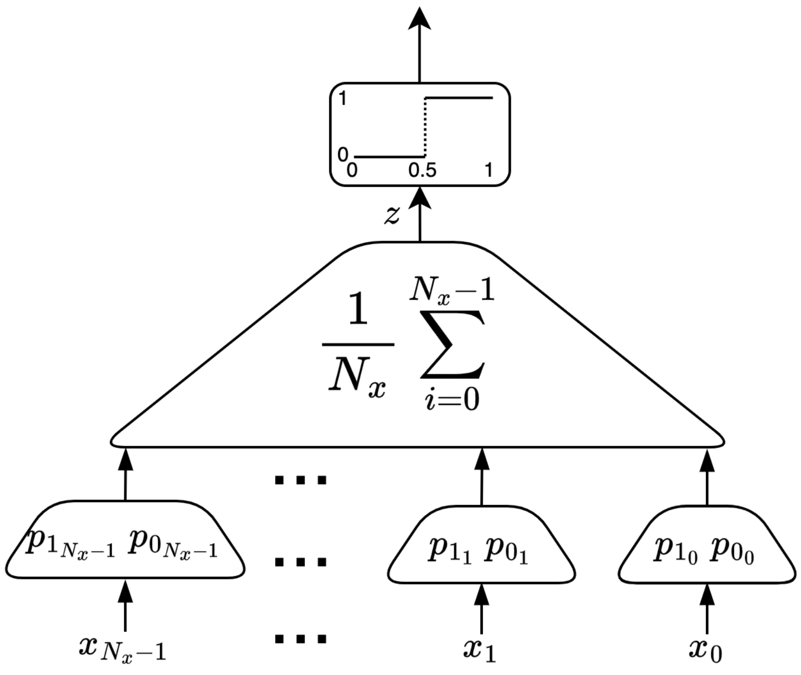


Figure 4. Binary model tree

If we apply the equation to MNIST dataset, a significant issue arises with the background pixels. Since a large portion of the image consists of background pixels that are consistently black across the dataset, the model only learns the value p_0 for these pixels. As a result, if a cell input has a value of 1, it will return its untrained value p_1, leading to distortions in the summation, that if accumulate, can negatively affect the classification results.
Therefore, it is necessary to apply noise to the inputs so the model can learn p_1 during training. To achieve this, we will convert the binary neuron to a binary probability model.

Binary probability neuron

In the previous section, we discussed the inputs and outputs in the time domain. From this section onward, we will focus on the probability domain.
The original binary model was modified to fit a probability model, so it does not represent the same model.

Cell inputs

As we saw in the previous section, it is required to apply noise to completely train the look up table of the cell. The amount of noise was determined by the perception of numbers of the *Figure 5*

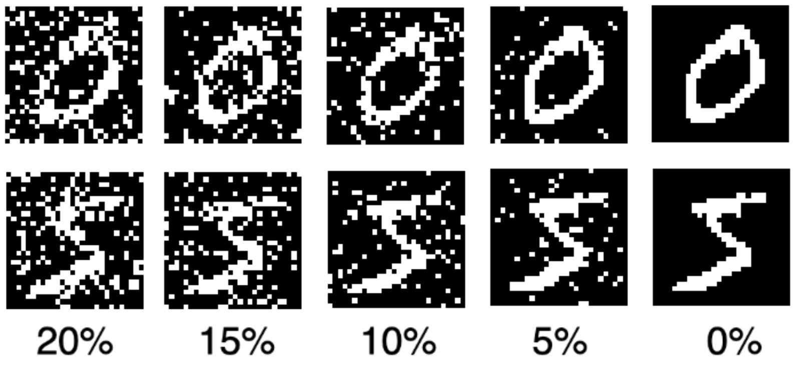
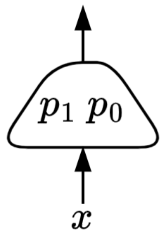


Figure 5. Noise in MNIST samples

As shown in Figure 5, a 10% noise level in the data is enough to clearly recognize the numbers in the images. Therefore, I chose to add noise of 0.1, setting the range for x_i between 0.1 and 0.9 probability. As a result, the grayscale values in the image will correspond to intermediate probabilities within this 0.1 to 0.9 range.

Activation function



$$\begin{aligned} P(1) &= p_1 x + p_0(1-x) \\ &= (p_1 - p_0)x + p_0 \end{aligned} \tag{3}$$

$$x \in [0,1],\ p_i \in [0,1]$$

The Eq. 3 is the probability of the cell to output 1 for the given x.
All the cells output a probability, but obtaining the total probability of the model is difficult. To address this, we will use an approximation. We divide all the input cells into four groups and approximate the probability for each group as the average of the output cells from that group.

Next, we obtain the output probability as a model of 4 inputs P_0, P_1, P_2 and P_3, and to simplify even more, we are going to suppose $z \approx P_0 \approx P_1 \approx P_2 \approx P_3$, where z is the average output of all cells.

1111	**1110**	**1100 - *50%***	**1000**	**0000**
z^4	$4z^3(1-z)$	$3z^2(1-z)^2$	0.0	0.0

Table 1. Probability to output one for inputs P

The *Table 1* shows the probability considering the model is using *Eq. 2* to output one or zero. The column 1111 is the probability all four groups output one, the column 1100 is the probability to get 2 ones and 2 zeros, and we considered is a 50% chance to get one in this case, so it is divided by 2.

Adding the elements of the table, we have $z^2(3-2z)$. If we do the same with 8 groups, we obtain $z^4(-20z^3+70z^2-84z+35)$.

$$\sigma(x) = \frac{1}{1+e^{-\alpha(x+\beta)}} \qquad (4)$$

Logistic function [2]

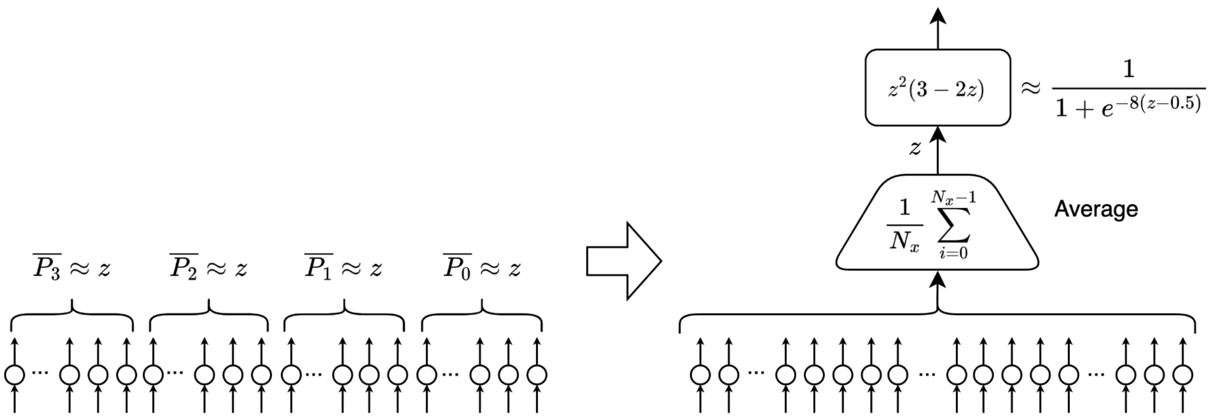

$\overline{P_3} \approx z$
$\overline{P_2} \approx z$
$\overline{P_1} \approx z$
$\overline{P_0} \approx z$
$z^2(3-2z)$
$\approx \frac{1}{1+e^{-8(z-0.5)}}$
z
$\frac{1}{N_x}\sum_{i=0}^{N_x-1}$
Average

Figure 6. Activation function as a logistic function

Figure 7 shows the comparison with the logistic function. By considering the inputs as probability values and using Eq. 2, a logistic function is created in the output, and this function will be used instead.

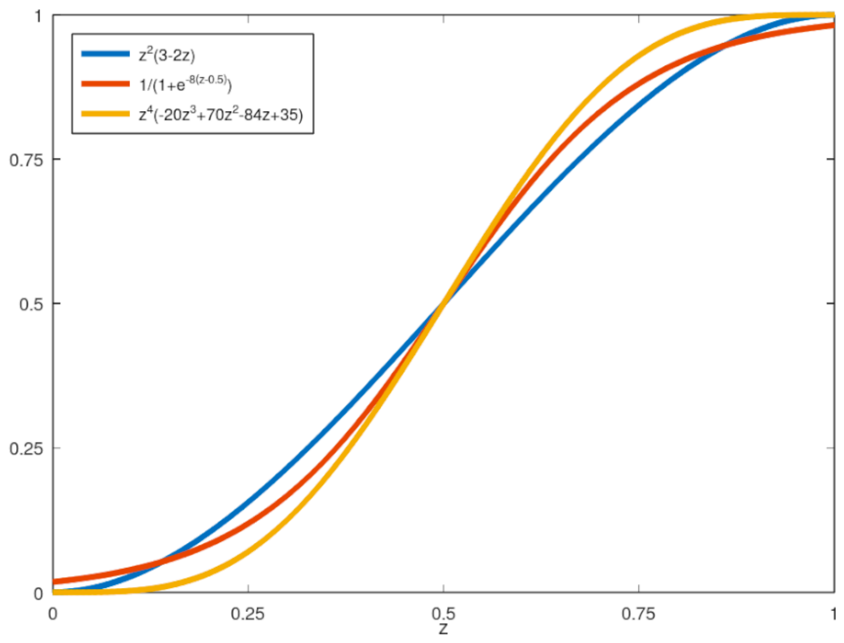


Figure 7. Comparation with logistic function

Normalized weights (distribution vector)

From *Eq. 3*, we define the cells normalized weights as:

$$\begin{aligned} z &= \frac{1}{N_x}\sum_{i=0}^{N_x-1}\left((p_{1_i}-p_{0_i})x_i+p_{0_i}\right) \\ &= \sum_{i=0}^{N_x-1}\frac{(p_{1_i}-p_{0_i})}{N_x}x_i+\frac{1}{N_x}\sum_{i=0}^{N_x-1}p_{0_i} \\ &= \sum_{i=0}^{N_x-1}\hat{w}_i x_i+\beta \qquad (5) \end{aligned}$$

Where:

$$\hat{w}_i=\frac{p_{1_i}-p_{0_i}}{N_x}$$

$$\beta=\frac{1}{N_x}\sum_{i=0}^{N_x-1}p_{0_i}$$

$$x_i\in[0,1],\quad p_{k_i}\in[0,1],\quad \hat{w}_i\in\left[\frac{-1}{N_x},\frac{1}{N_x}\right]$$

$p_{0_{i+1}}=p_{1_i}$ must be considered to be able to transform w_i to p_{k_i}.

AI neuron tree

In the context discussed in this article, we are modeling a group of neurons. However, most references, such as [3] and [4], refer to the following model as a single neuron named *sigmoid neuron*.

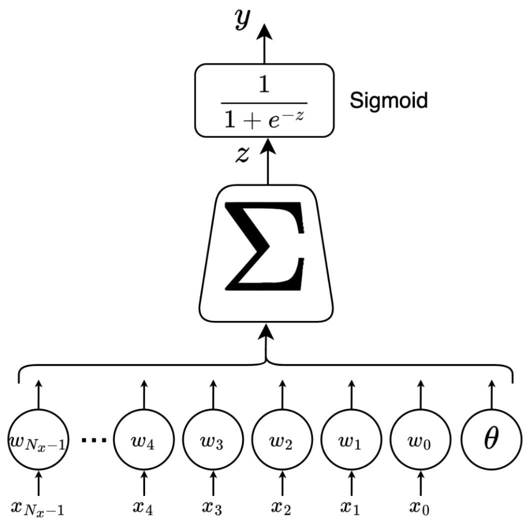


Figure 8. AI neuron tree

$$y = \frac{1}{1+e^{-z}}$$

$$z = \sum_{i=0}^{N_x-1} w_i x_i + \theta$$

$$w_i \in [-\infty, \infty]$$

$$x_i \in [0, 1]$$

This is the neural tree model. However, as we discussed in the previous section, the sigmoid function receives the average output of the cells. We can break down the equation into:

$$y = \frac{1}{1+e^{-z}} \tag{6}$$

$$\begin{aligned} z &= \sum_{i=0}^{N_x-1} w_i x_i + \theta \\ &= \mathbf{w} \cdot \mathbf{x} + \theta \qquad (7) \\ &= \alpha\, \hat{\mathbf{w}} \cdot \mathbf{x} + \alpha\beta \\ &= \alpha\left(\hat{\mathbf{w}} \cdot \mathbf{x} + \beta\right) \qquad (8) \end{aligned}$$

Where $\hat{\mathbf{w}} \cdot \mathbf{x} + \beta$ is the average probability of the entry cells and α adjust the slope of the function.

Bias β
Offset $\theta = \alpha\beta$
Weights $\mathbf{w}$
$\alpha = \sqrt{N_x(w_n^2 + \cdots + w_1^2 + w_0^2)}$
Normalized weights (distribution vector) $\hat{\mathbf{w}} = \frac{\mathbf{w}}{\alpha}$

$w_i \in [-\infty, \infty]\,, \qquad \hat{w}_i \in \left[-\frac{1}{N_x}, \frac{1}{N_x}\right]$
noise = 0.1

The model is practically the same as the sigmoid function defined in [3:1], except we are applying a 0.1 noise in the input and we defined some parameters that doesn't alter the model.

In this case, the probabilities are defined within the range $x \in [0, 1]$, and the sigmoid function (*Eq.3*) has its transition at 0.0. To better visualize the probabilities, we shift the input by adjusting it to $x' = x + 0.5$.
Later in this document, alternatives functions are presented where the probabilities are mapped into $[-1, 1]$ for convenience.

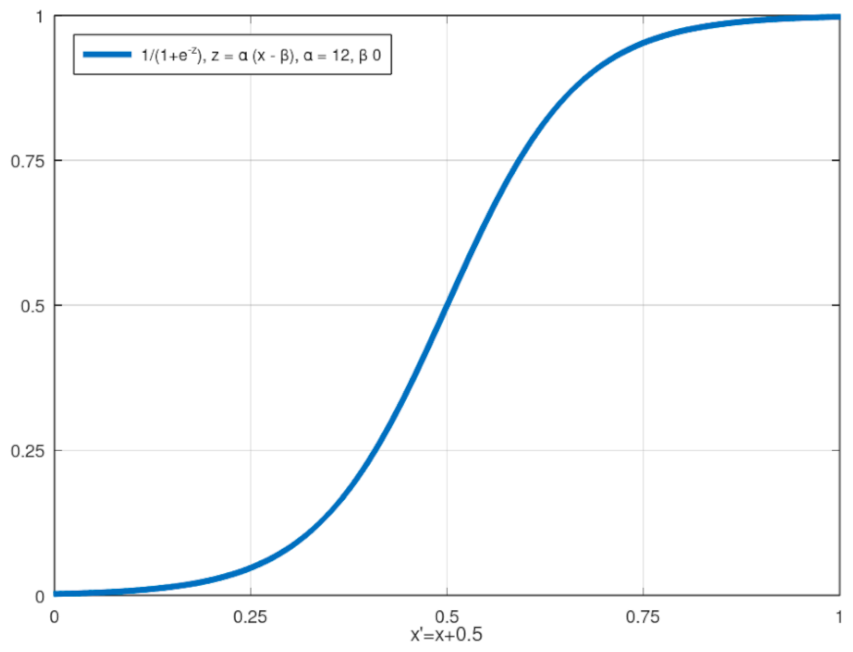
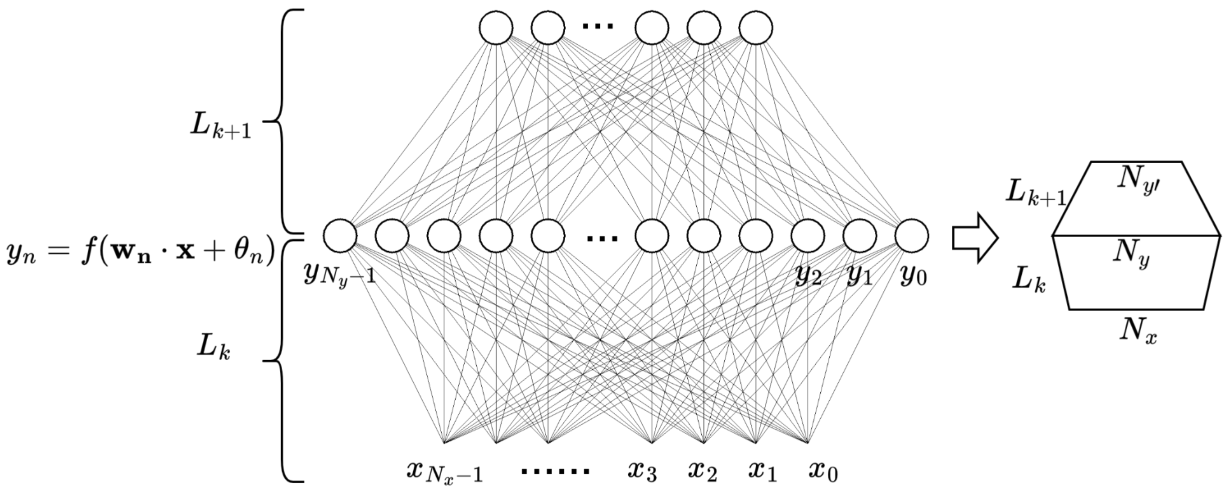


Figure 9. Sigmoid function

In the *Figure 9* is the plot of sigmoid function where the parameter α modify the slope of the transition and β displaces the function to the right or left.

Connecting multiple trees

As the defined tree closely resembles the sigmoid neuron, which is well-studied and widely used, we employed a commonly used architecture: fully connected trees forming layers, also known as *Feed-forward neural networks*[5].


L_{k+1}
$y_n = f(\mathbf{w_n} \cdot \mathbf{x} + \theta_n)$
L_k
y_{N_y-1}
y_2 y_1 y_0
x_{N_x-1} ...... x_3 x_2 x_1 x_0
L_{k+1}
$N_{y'}$
N_y
L_k
N_x
Figure 10. Full connected trees

N_y number of outputs.
N_x number of inputs.
N_L number of layers.

Each layer consist in a input vector $\mathbf{x}$ creating a output vector $\mathbf{y}$. Every element in vector $\mathbf{y}$ has a tree with its own weights and bias.
To evaluate the neural network, we calculate in lowest layer $y_n = f(\mathbf{w_n} \cdot \mathbf{x} + \boldsymbol{\theta_n})$, where $f()$ is the activation function.
This y_n will be the new input for the next layer and the process is repeated.
If we the define $g(\mathbf{x}) = f(\mathbf{W}\mathbf{x} + \boldsymbol{\theta})$ for one layer ($\mathbf{W}$ as a Matrix) , multiple layers are evaluated as:

$$\begin{aligned} y(\mathbf{x}) &= g_{N_{L-1}}(\cdots(g_2(g_1(g_0(\mathbf{x}))))) \\ &= g_{N_{L-1}} \circ \cdots \circ g_2 \circ g_1 \circ g_0 \end{aligned} \tag{9}$$

Learning (backpropagation)

Our neural network function generates an output by *Eq.9*, and the output is the probability to be right or wrong, as we are applying noise and our model it is in probability domain.

The idea of backpropagation[6] is to linearize the function *Eq.9* in a specific data point to calculate the relation of ${\partial E}/{\partial w_i}$ and ${\partial E}/{\partial \theta}$ to update the weights and bias to reduce the error.
In this scenario, the input consists of labeled data images, and we want our neural network output be the label that corresponds for that input, this is called supervised learning[7].

Error Function

First, we have to define the first transfer gradient, which will be calculated using the error function. The neural network produces an output vector $\mathbf{y}$, while our target vector is $\mathbf{y}_t$ which are of the same dimension.

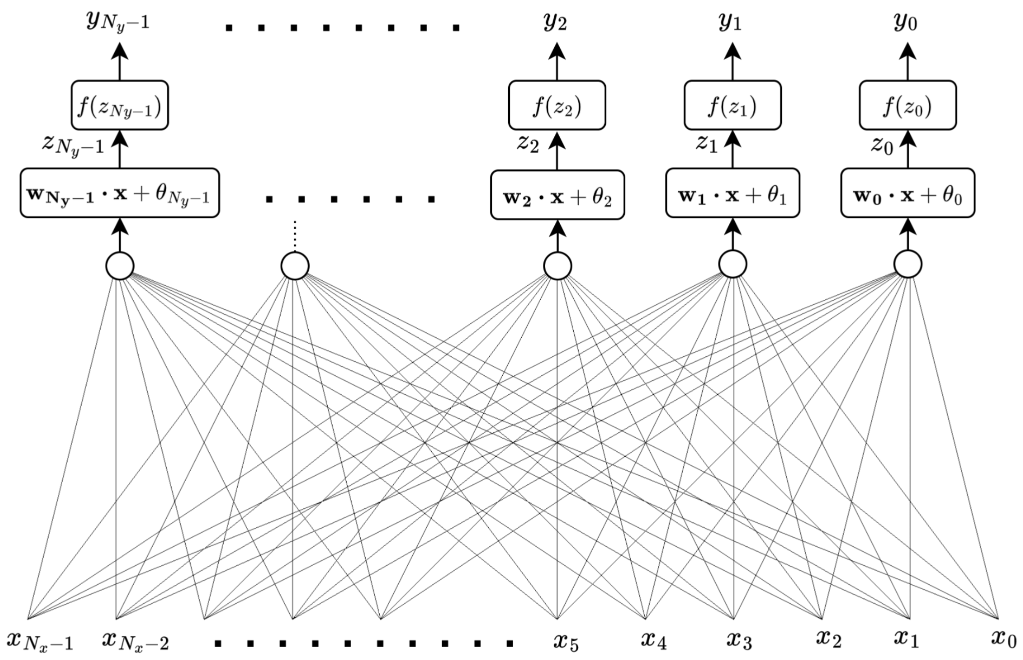
$$E_n = (y_n - y_{tn})^2 \tag{10}$$

Derivative:

$$\frac{\partial E_n}{\partial y_n} = 2(y_n - y_{tn}) \tag{11}$$

Where E_n is the error of the neural network and the derivative $\frac{\partial E_n}{\partial y_n}$ is passed to the previous layer.

Updating layers weights



y_{N_y-1}
y_2
y_1
y_0
$f(z_{Ny-1})$
$f(z_2)$
$f(z_1)$
$f(z_0)$
z_{N_y-1}
z_2
z_1
z_0
$\mathbf{w_{N_y-1}} \cdot \mathbf{x} + \theta_{N_y-1}$
$\mathbf{w_2} \cdot \mathbf{x} + \theta_2$
$\mathbf{w_1} \cdot \mathbf{x} + \theta_1$
$\mathbf{w_0} \cdot \mathbf{x} + \theta_0$
x_{N_x-1}
x_{N_x-2}
x_5
x_4
x_3
x_2
x_1
x_0

Figure 11. Layer structure

The index i refers to the iterator ranging from 0 to $N_x - 1$, while the index n represents the iterator spanning from 0 to $N_y - 1$.

Updating the layer weights and biases requires the derivative $\frac{\partial E_n}{\partial y_n}$, which is used in the following formula:

$$w_{n_i} = w_{n_i} - \eta_w \frac{\partial E_n}{\partial w_{n_i}} \tag{12}$$

$$\theta_n = \theta_n - \eta_\theta \frac{\partial E_n}{\partial \theta_n} \tag{13}$$

We are using $\eta_w = \eta_\theta$ and it's the learning rate.

Figure 11 illustrates the operations within the layer. Here, $f(z)$ represents the activation function applied in the layer, leading to the following expression:

$$y_n = f(z_n)$$

$$z_n = \mathbf{w_n} \cdot \mathbf{x} + \theta_n$$

And their derivatives:

$$\begin{aligned}\frac{\partial y_n}{\partial z_n} &= \frac{\partial f(z_n)}{\partial z_n} \\ \frac{\partial z_n}{\partial w_{n_i}} &= x_i \\ \frac{\partial z_n}{\partial \theta_n} &= 1\end{aligned}$$

Using chain rule we get:

$$\begin{aligned}\frac{\partial E_n}{\partial w_{n_i}} = \frac{\partial z_n}{\partial w_{n_i}} \frac{\partial y_n}{\partial z_n} \frac{\partial E_n}{\partial y_n} \\ = x_i \frac{\partial f(z_n)}{\partial z_n} \frac{\partial E_n}{\partial y_n} \quad (14)\end{aligned}$$

$$\begin{aligned}\frac{\partial E_n}{\partial \theta_n} = \frac{\partial z_n}{\partial \theta_n} \frac{\partial y_n}{\partial z_n} \frac{\partial E_n}{\partial y_n} \\ = \frac{\partial f(z_n)}{\partial z_n} \frac{\partial E_n}{\partial y_n} \quad (15)\end{aligned}$$

With *Eq.14* and *Eq.15* we can update weights and biases in *Eq.12* and *Eq.13*.

Additional notes

One important observation is that in *Eq. 8*, we define $\theta = \alpha\beta$, where β represents the summation of p_{0_i} as described in *Eq. 5*. Additionally, w_i is defined as a pair of p_i. This implies that the learning rate should be $\eta_w = \frac{2}{N_x}\eta_\theta$, but in practice, the learning of the network would be quite slow.

Calculating transfer gradients

Before updating the weights, we need to calculate the transfer gradient $\frac{\partial E_i}{\partial x_i}$ to pass it to the previous layer. In this context, the input x_i of the current layer corresponds to the output y_n from the previous layer. This process is then repeated to update the weights and biases across the entire network.

$$\frac{\partial z_n}{\partial x_i} = w_{n_i}$$

Then using chain rule we obtain:

$$\frac{\partial E_i}{\partial x_i} = \sum_{n=0}^{Ny-1} \frac{\partial z_n}{\partial x_i} \frac{\partial y_n}{\partial z_n} \frac{\partial E_n}{\partial y_n}$$

$$= \sum_{n=0}^{Ny-1} w_{n_i} \frac{\partial f(z_n)}{\partial z_n} \frac{\partial E_n}{\partial y_n} \qquad (16)$$

The transfer gradient calculated in *Eq. 16* is passed to previous layer.

Activation function

Sigmoid (1x1)

At this stage, we modeled the activation function as a sigmoid or logistic function in the probability domain, based on the premise that similar inputs produce similar outputs, as described in *Eq. 2*. However, requiring a 50% change in pixel values (in the case of images) for the activation function to alter its output can be excessive. While adjusting the bias can help to reduce the activation area, this also means the deactivation area is larger.

Gauss (2x1)

Consider a neural tree that detects a specific feature, such as the image of the digit '2' from the MNIST dataset. When this image is input, the tree outputs a value of one, representing a 100% probability of recognizing the digit. However, as noise is added (with all inputs being binary in the time domain), the number of pixels matching the feature decreases, leading to a reduced output probability. As the noise approaches 100%, the result is the inverted image.

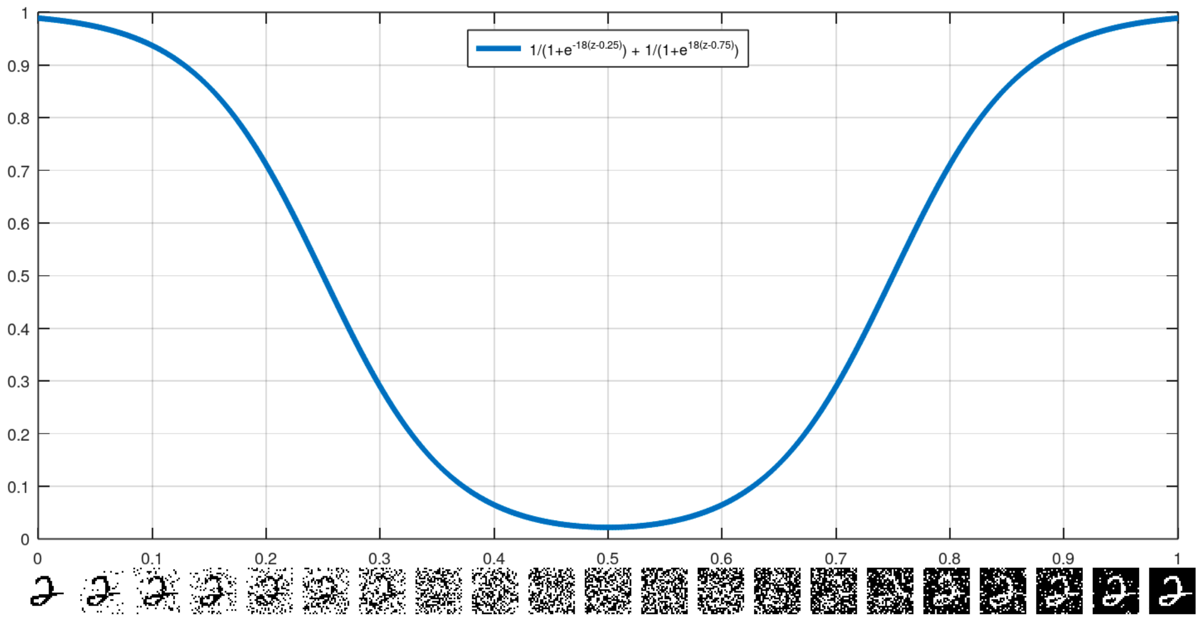


Figure 12. Double logistic function

We model this perception using two logistic functions, as illustrated in *Fig. 12*. The resulting shape closely resembles an inverted Gaussian bell curve. For simplicity, we can directly model it as a Gaussian bell curve, as the labels (0 and 1) are merely symbolic. This approach allows our neural network to learn the system's parameters $\mathbf{w}$ and θ without the need to invert the Gaussian bell curve.

$$f(z) = e^{-z^2} \tag{17}$$

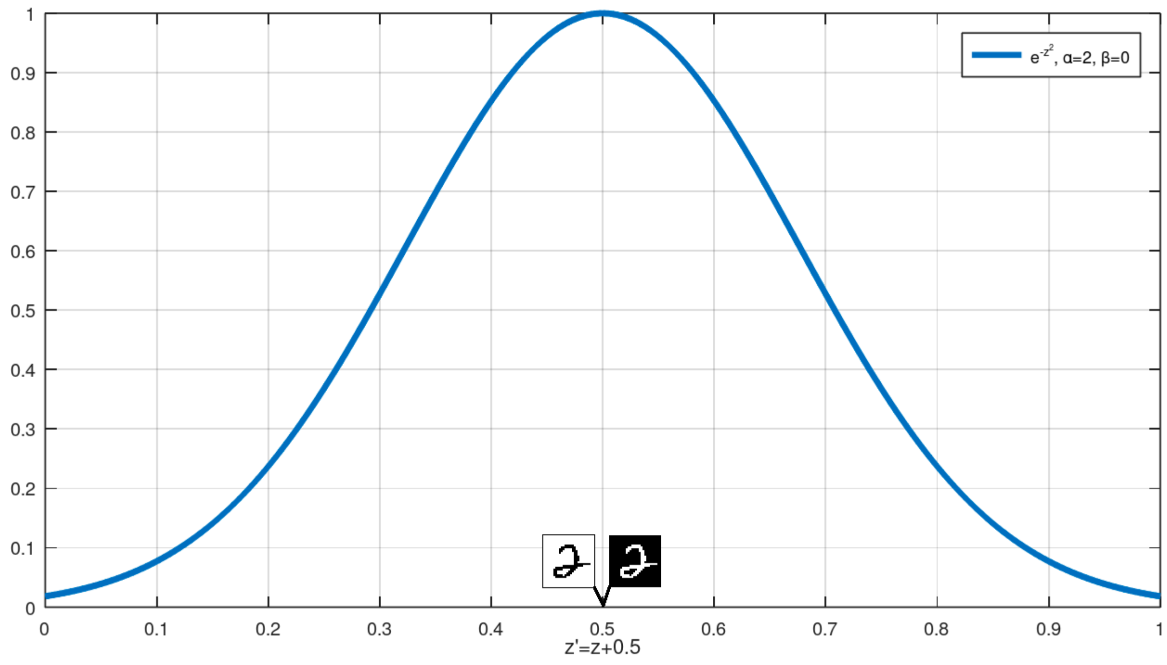


Figure 13. Gauss function

Using this Gaussian bell curve enables the learning of two features within the same neural tree, particularly in the context of image data. However, if this isn't necessary, the network can adjust the data distribution to approximate a sigmoid-like function by learning the parameters α, $\hat{w}$, and β.

In tests comparing the Gaussian bell curve to the sigmoid function on the MNIST dataset after 10 epochs, both achieved an accuracy of 98.6%. However, when training on both the original and inverted data, the sigmoid function reached an accuracy of 96.4%, while the Gaussian bell curve achieved 98.1%.

These results are not definitive, as the accuracy may vary depending on factors like learning rate or initial parameter settings.

Cosine (Nx1)

We infer the gauss activation function from joining 2 logistic functions, but what if the add 3, 4 or N logistic functions, in that case the obtain a cosine wave, also from *Fig 12* or *Fig.13* can be observe the similarity to a cosine wave function.

$$f(z) = \frac{1 - \cos(z)}{2} \tag{18}$$

$$\begin{aligned} z &= \mathbf{w} \cdot \mathbf{x} + \theta \\ &= \alpha \hat{\mathbf{w}} \cdot \mathbf{x} + \theta \end{aligned}$$

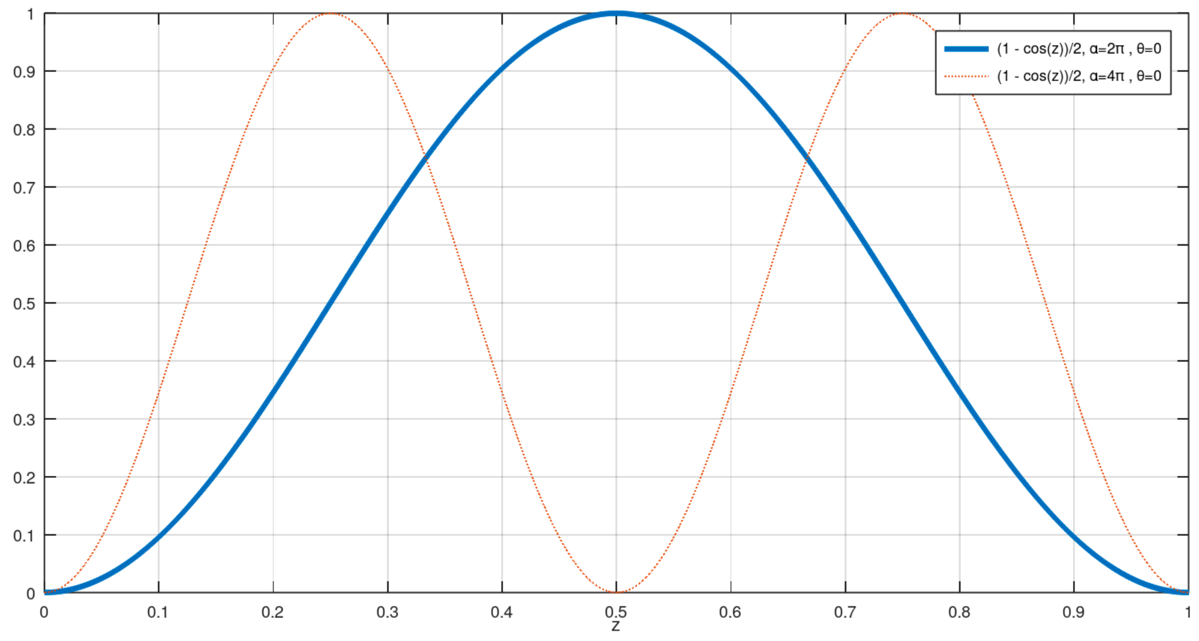


Figure 14. Cosine wave function

In this activation function the parameter α would be the frequency of the function where the parameter θ is the offset.

I think in practice, the cosine wave only would learn α at very low frequencies due the condition that similar inputs should create similar outputs.

From the graphs, the sigmoid function approximates half of a wave, the Gaussian function approximates a full wave, and the cosine wave appears to be the most general function, that includes both by learning the frequency α.

Relu, triangle and triangle wave

The ReLU (Rectified Linear Unit) activation function[8] is currently the most commonly used in training neural networks. This is due to its simplicity in calculation and its reduced susceptibility to the vanishing gradient problem.
If ReLU can be seen as derived from the sigmoid function, then for a Gaussian bell curve, a corresponding activation function might resemble a triangular function. Similarly, for a cosine wave, a triangle wave might be an appropriate counterpart.

Activation functions table

The figure below illustrates the activation function characterized by the parameters α and β, demonstrating the shape of the function. The weights $\hat{w}$ is constrained within the interval $\left[\frac{-1}{N_x}, \frac{1}{N_x}\right]$, including the functions which probability mapping is in range $[-1, 1]$ for convenience. The initial values of w are based in Xavier initialization method[9][10] by changing the parameter α.
N_x number of inputs
$\{x\}$ denotes the fractional part of x.

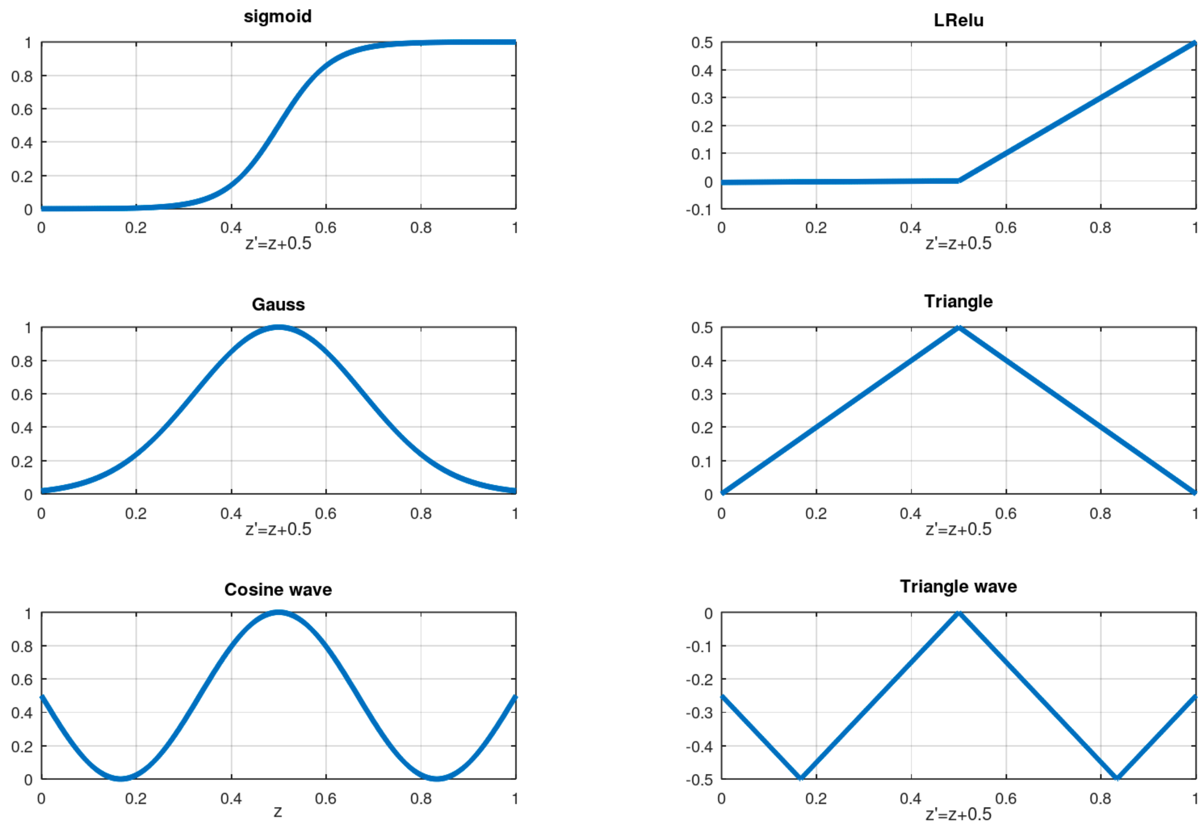


Figure 15. Activation functions P[0,1]

Type	Equation	derivative	Initial values
Logistic	$f(z) = \dfrac{1}{1+e^{-z}}$	$f'(z) = f(z)(1-f(z))$	$\eta = 0.1$ $\alpha = 2\sqrt{N_x}$ $\beta = 0$
Gauss	$f(z) = e^{-z^2}$	$f'(z) = -2zf(z)$	$\eta = 0.01$ $\alpha = \sqrt{N_x}$ $\beta = 0$
Cosine	$f(z) = \dfrac{1-\cos(z)}{2}$	$f'(z) = \dfrac{\sin(z)}{2}$	$\eta = 0.01$ $\alpha =$ $\pi\sqrt{N_x}$ $\beta = 0.5$
LReLU	$f(z) = \begin{cases} 0.01z & \text{if } z \le 0 \\ z & \text{if } z > 0 \end{cases}$	$f'(z) =$ $\begin{cases} 0.01 & \text{if } z \le 0 \\ 1 & \text{if } z > 0 \end{cases}$	$\eta = 0.01$ $\alpha = \sqrt{N_x}$ $\beta = 0$
Triangle	$f(z) = \begin{cases} z+1 & \text{if } z \le 0 \\ -z+1 & \text{if } z > 0 \end{cases}$	$f'(z) =$ $\begin{cases} 1 & \text{if } z \le 0 \\ -1 & \text{if } z > 0 \end{cases}$	$\eta = 0.001$ $\alpha = \sqrt{N_x}$ $\beta = 0$
Triangle wave	$f(z_t) = \{z/4\} - 0.5$ $f(z_t) =$ $\begin{cases} z_t+1 & \text{if } z_t \le 0 \\ -z_t+1 & \text{if } z_t > 0 \end{cases}$	$f(z_t) = \{z/4\} - 0.5$ $f'(z_t) =$ $\begin{cases} 1 & \text{if } z_t \le 0 \\ -1 & \text{if } z_t > 0 \end{cases}$	$\eta = 0.001$ $\alpha = \sqrt{N_x}$ $\beta = 0$

Table 2. Activation functions P[0,1]

Code sample: https://github.com/Gonz3d/AI_Artificial_Neuron/Network_P01

In Fig.16 presents activation functions that map probability values within the range of [-1, 1].

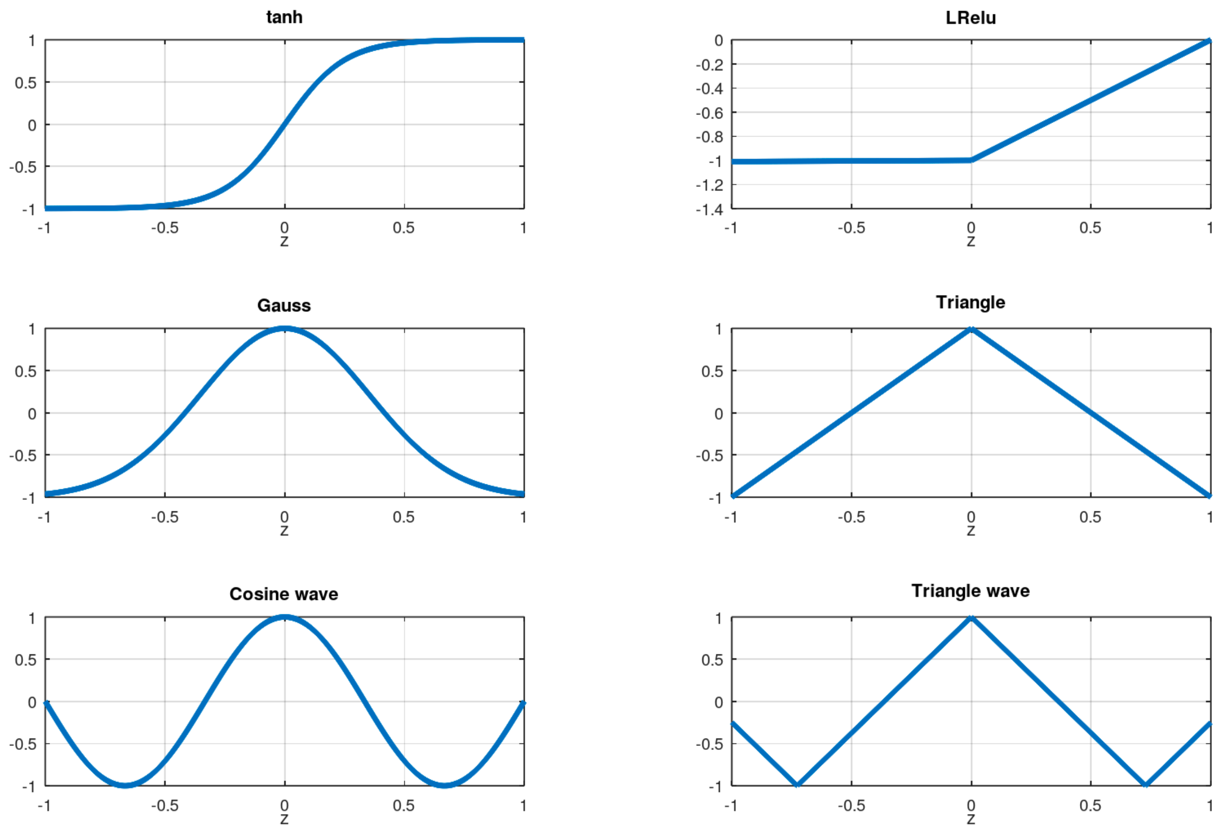


Figure 16. Activation functions P[-1,1]

Type	Equation	derivative	Initial values
Logistic	$f(z) = \dfrac{2}{1+e^{-2z}} - 1$ $= \tanh(z)$	$f'(z) = 1 - f^2(z)$	$\eta = 0.002$ $\alpha = \sqrt{N_x}$ $\beta = 0$
Gauss	$f(z) = 2e^{-z^2} - 1$	$f'(z) = -2z(f(z)+1)$	$\eta = 0.001$ $\alpha =$ $\frac{1}{2}\sqrt{N_x}$ $\beta = 0$
Cosine	$f(z) = \cos(z)$	$f'(z) = -\sin(z)$	$\eta = 0.002$ $\alpha =$ $\frac{\pi}{2}\sqrt{N_x}$ $\beta = 0$
LReLU	$f(z) =$ $\begin{cases} 0.01z - 1 & \text{if } z \le 0 \\ z - 1 & \text{if } z > 0 \end{cases}$	$f'(z) =$ $\begin{cases} 0.01 & \text{if } z \le 0 \\ 1 & \text{if } z > 0 \end{cases}$	$\eta = 0.001$ $\alpha = \sqrt{N_x}$ $\beta = 0$

Type	Equation	derivative	Initial values
Triangle	$f(z) = \begin{cases} z+1 & \text{if } z \le 0 \\ -z+1 & \text{if } z > 0 \end{cases}$	$f'(z) = \begin{cases} 1 & \text{if } z \le 0 \\ -1 & \text{if } z > 0 \end{cases}$	$\eta = 0.001$ $\alpha = \sqrt{N_x}$ $\beta = 0$
Triangle wave	$z_t = 4\{z/8\} - 2$ $f(z_t) = \begin{cases} -z_t - 1 & \text{if } z_t \le 0 \\ z_t - 1 & \text{if } z_t > 0 \end{cases}$	$z_t = 4\{z/8\} - 2$ $f'(z_t) = \begin{cases} -1 & \text{if } z_t \le 0 \\ 1 & \text{if } z_t > 0 \end{cases}$	$\eta = 0.001$ $\alpha = \sqrt{N_x}$ $\beta = 0$

Table 3. Activation functions P[-1,1]

Code sample: https://github.com/Gonz3d/AI_Artificial_Neuron/Network_P11

Results

The results below are the accuracy obtained with MNIST dataset with 128 nodes in hidden layer and 10 output nodes. The training set consist in 60000 samples and the training set in 10000 samples.

Type	10 epoch	20 epoch	test set (10epoch)
Logistic	98.6%	99.2%	97.6%
Gauss	98.7%	99.2%	97.4%
Cosine	98.5%	98.9%	97.6%
LRelu	98.0%	98.6%	96.8%
Triangle	97.0%	97.6%	96.5%
Triangle wave	92.6%	93.2%	92.5%

Table 4. Results with functions P[0,1]

Type	10 epoch	20 epoch	test set (10epoch)
Logistic	98.5%	99.1%	97.3%
Gauss	98.8%	99.3%	97.6%
Cosine	98.8%	99.3%	97.5%
LRelu	97.3%	97.9%	96.3%
Triangle	97.5%	97.9%	96.5%
Triangle wave	98.0%	98.5%	97.1%

Table 5. Results with functions P[-1,1]

Other experiments

Global error

As neural networks become deeper by stacking multiple layers, they can encounter the vanishing gradient problem. This problem occurs when the gradients of the loss function become very small during backpropagation. As a result, the lower layers learn slowly or may fail to learn altogether, hindering the overall performance of the network.

However, I believe biological neurons likely employ a more logical learning mechanism than passing gradients through a network, though some form of feedback is still necessary for learning.

The key idea is that the entire network must be aware of the output error, which reflects the probability of the network's prediction being correct or incorrect. This error is captured by the gradient.

To explain this mathematically, consider a neural network with five layers. The gradient of the function with respect to the input of the first layer, I_0, can be expressed as:

$$\frac{\partial f(I_0)}{\partial I_0} = \frac{\partial I_1}{\partial I_0}\frac{\partial I_2}{\partial I_1}\frac{\partial I_3}{\partial I_2}\frac{\partial I_4}{\partial I_3}\frac{\partial I_5}{\partial I_4}\frac{\partial E}{\partial I_5}$$

where $\frac{\partial E}{\partial I_5}$ represents the gradient of the output error.
If we add layers such that the following relationship holds:

$$\frac{\partial I_3}{\partial I_2} = \frac{\partial I_5}{\partial I_2}\frac{\partial E}{\partial I_5}\frac{\partial I_3}{\partial E}$$

we can substitute this into the gradient equation to get:

$$\frac{\partial f(I_0)}{\partial I_0} = \frac{\partial I_1}{\partial I_0}\frac{\partial I_2}{\partial I_1}\frac{\partial I_5}{\partial I_2}\frac{\partial E}{\partial I_5}\cdot \frac{\partial I_3}{\partial E}\frac{\partial I_4}{\partial I_3}\frac{\partial I_5}{\partial I_4}\frac{\partial E}{\partial I_5}$$

This formulation shows that learning through backpropagation can be achieved by splitting the process into two completely independent parts.
Testing this approach with multiples layers with MNIST [128, 128, 10, 128, 128, 10, 128, 128, 10, 128, 128, 10, 128, 128, 10, 128, 128, 10, 128, 128, 10] using sigmoid(with reduced learning rate), and the layers learn the output gradient where the layer has 10 outputs, it gets a 95.9% in 10 epochs(99.5% in 100 epochs).
I think it would be better if we learn a percent of the total weights instead of learning all the weights at the same time.
Code sample: https://github.com/Gonz3d/AI_Artificial_Neuron/exp/GlobalError

Neural Network like Fourier series.

Consider a 2 layer neural network using cosine wave activation function, in the first layer the input x_0 is multiplied by w_{0_0} and the output is a cosine function $\cos(\mathbf{w_0 x} + \theta)$.
By the second layer, the process is repeated and the output of first layer is multiplied by a magnitude w_0 of the second layer and sum with the others nodes, and the series of cosine functions are obtained in the second layer $\cos(...w_1 \cos(\mathbf{w_1 x} + \theta_1) + w_0 \cos(\mathbf{w_0 x} + \theta_0))$ like in Fourier series.

If we transform the series into normalized form $\cos(...w_1 \cos(\alpha_1 \mathbf{\hat{w}_1 x} + \theta_1) + w_0 \cos(\alpha_0 \mathbf{\hat{w}_0 x} + \theta_0))$ and learn only the α and θ, and maintain constant $\mathbf{\hat{w}_n}$ also work to converge into the desired result, but we have to add more nodes and layers to get good accuracy.
By learning only α and θ in MNIST dataset with a [2048, 2048, 2048, 2048,10] layer configuration it gets an accuracy of 95.1% with 10 epochs.
This example only shows a different way to reduce the error by keeping $\hat{w}$ constant, at least for this simple example with MNIST dataset.
Code sample: https://github.com/Gonz3d/AI_Artificial_Neuron/exp/w_constant

Appendix A

For a lookup table of one input:

$$p(x) = c_1 x_0 + c_0 \pmod 2$$

$$\begin{bmatrix} p_1 \\ p_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_0 \end{bmatrix}$$

where:

$M_1 = M_1^{-1} (\text{mod} 2) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$

$p_i \in \{0, 1\}$ are the outputs p_0 for $x_0 = 0$ and p_1 for $x_0 = 1$

$c_i \in \{0, 1\}$ are the coefficients of $\text{mod} 2$ equation

---

For two inputs:

$$p(x) = c_3 x_1 x_0 + c_2 x_1 + c_1 x_0 + c_0 \pmod 2$$

$$\begin{array}{c} x[11] \\ x[10] \\ x[01] \\ x[00] \end{array} \begin{bmatrix} p_3 \\ p_2 \\ p_1 \\ p_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix}$$

where:

$M_2 = M_2^{-1} (\text{mod} 2) = \begin{bmatrix} M_1 & M_1 \\ 0 & M_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

$p_i \in \{0, 1\}$ are the outputs p_i for $x_{[1,0]}$

$c_i \in \{0, 1\}$ are the coefficients of $\text{mod} 2$ equation

---

It can be observed the matrix M_2 is constructed from M_1 by concatenating the matrix and can be generalized as $M_n = \begin{bmatrix} M_{n-1} & M_{n-1} \\ 0 & M_{n-1} \end{bmatrix}$ and it's a involutory matrix in $\text{mod} 2$ for any number of inputs, but the equation must be constructed by carefully assigning the proper order of the elements so the matrix will be a valid relation.
This matrix presents a direct relation between the output function values $\vec{q}$ and the coefficients $\vec{c}$.

References

1. https://yann.lecun.com/exdb/mnist/ "MNIST dataset" ↩
2. https://en.wikipedia.org/wiki/Logistic_function "Wikipedia Logistic Function" ↩
3. Michel A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015. ↩ ↩
4. https://en.wikipedia.org/wiki/Artificial_neuron "Artificial neuron" ↩
5. https://en.wikipedia.org/wiki/Feedforward_neural_network "Feed-Forward neural networks" ↩
6. https://en.wikipedia.org/wiki/Backpropagation ↩
7. https://en.wikipedia.org/wiki/Supervised_learning "Supervised learning" ↩
8. https://en.wikipedia.org/wiki/Rectifier_(neural_networks) ↩
9. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian ↩
10. https://www.deeplearning.ai/ai-notes/initialization/index.html ↩