Solving the XOR problem using a backpropagation-trained feedforward neural network

A step-by-step implementation

Ali Magzari

01/18/2022

I. Introduction

This project consists of training a feedforward neural network using the error backpropagation algorithm to approximate the XOR function (Table 1). Since this problem is linearly inseparable, a single perceptron is not adequate for the said task. The adopted network architecture is composed of two inputs, one hidden layer containing four neurons, and one output neuron (Figure 1).

Table 1: The XOR function

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Three different data representation-hyperparameter combinations will be tested to compare the total error convergence time:

a. Binary representation with a learning rate of 0.2 and momentum factor of 0
b. Bipolar representation with a learning rate of 0.2 and momentum factor of 0
c. Bipolar representation with a learning rate of 0.2 and a momentum factor of 0.9

Note: This project was implemented in both java and R.

II. Methods

Training the neural network consists of two parts: a feedforward and a backpropagation part. They are executed sequentially and repetitively until the total error at the output has reached a value lower than 0.05.

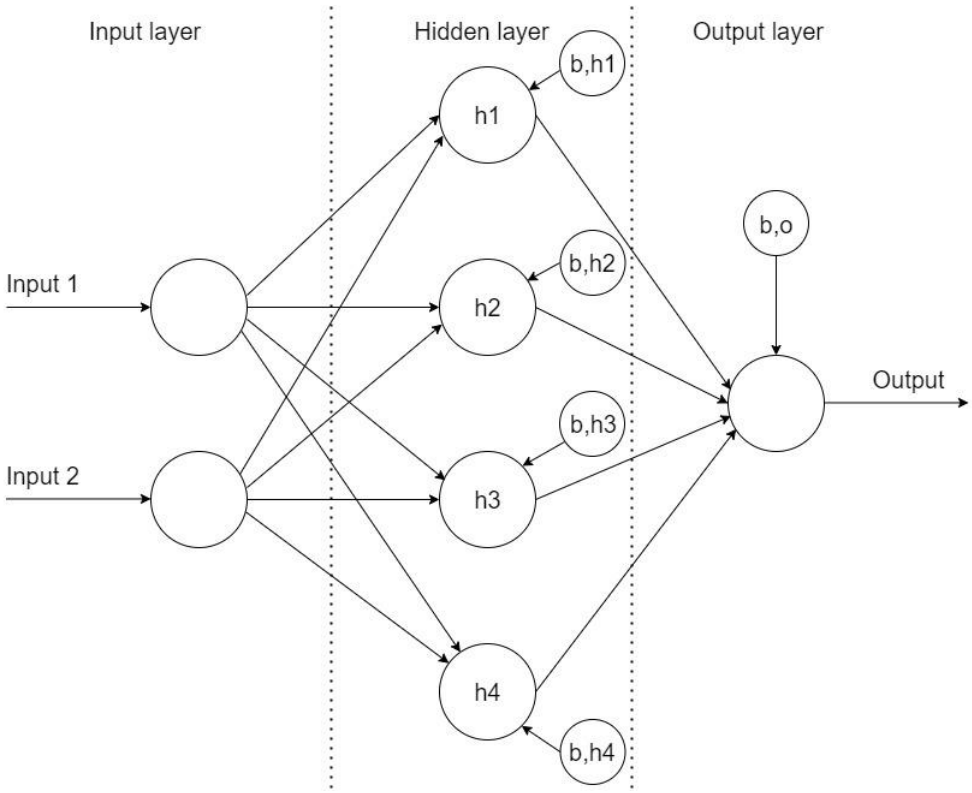

Figure 1: Neural network architecture

1. Activation functions

It is ought to first point out that each of the neuron-to-neuron connections is governed by a certain weight and each of the hidden and output neurons are subject to a bias. Also, both the hidden and output layer neurons contain an activation function, also called nonlinearity.

In the case of a binary data representation (0 and 1), a logistic activation function (Figure 2) is chosen:

$$f(x) = \frac{1}{1 + e^{-x}} \qquad (1)$$

In the case of a bipolar representation (-1 and +1), a hyperbolic tangent activation function (Figure 3) is used:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \qquad (2)$$

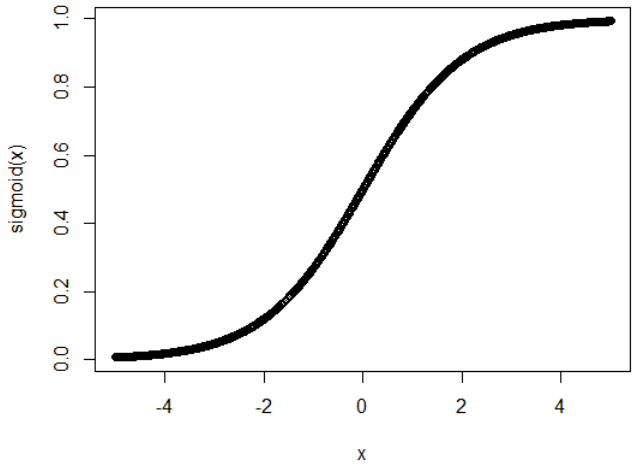

Figure 2: Logistic activation function

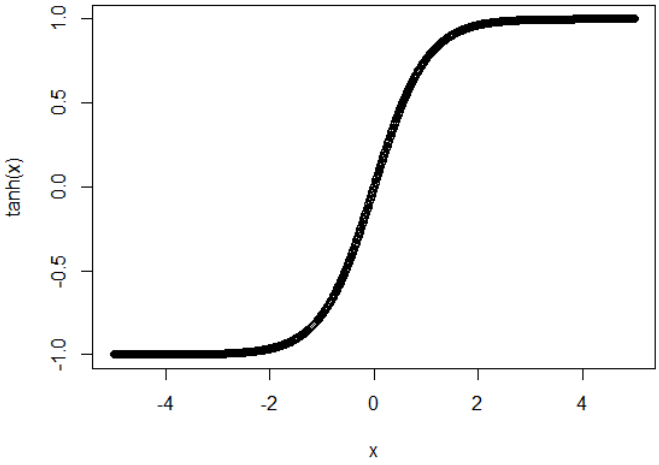

Figure 3: Hyperbolic tangent activation function

Note how the sigmoid graph is bounded by 0 and 1, while the hyperbolic tangent function is bounded by -1 and +1, thus explaining their utilization for the different data representations.

2. Feedforward propagation

This stage follows the natural direction of the network, and is very intuitive. The following equation summarizes the feedforward pass from the input to hidden layer.

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = f_{act}\left(\begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \\ w_{4,1} & w_{4,2} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_{h_1} \\ b_{h_2} \\ b_{h_3} \\ b_{h_4} \end{bmatrix}\right) \qquad (3)$$

where:

- h_a is the post-activation value of hidden neuron *a*
- f_{act} is the adequate activate function
- $w_{a,b}$ is the weight from input *b* to hidden neuron *a*
- x_a is the value of input *a*
- b_{h_a} is the bias fed to the hidden neuron *a*

The final result is then calculated as follow:

$$output = f_{act}\left(\begin{bmatrix} w_{o,1} & w_{o,2} & w_{o,3} & w_{o,4} \end{bmatrix} \times \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} + b_o\right) \qquad (4)$$

where:

- $w_{o,a}$ is the weight from hidden neuron *a* to output
- b_o is the bias fed to the output neuron

3. Error backpropagation

After the feedforward pass, an error is calculated based on the difference between the desired and actual outputs, after which the weights are updated via gradient descent. This stage steps are depicted as follow:

Calculate the error at the output node:

$$e_o = o_{desired} - o_{actual} \qquad (5)$$

Calculate the local gradient at the output:

$$\delta_o = e_o \frac{d}{dx} f_{act}(o_{actual}) \qquad (6)$$

Update the hidden-to-output weights:

$$\Delta_m w_{o,h_a} = \Delta w_{o,h_a} + \Delta_m w_{o,h_a} \qquad (7)$$

where:

- $\Delta w_{o,h_a}$ is the adjustment of the weight from the hidden neuron *a* to the output
- $\Delta_m w_{o,h_a}$ is the adjustment of the weight from the hidden neuron *a* to the output, relative to momentum, which is a cumulative adjustment process (shown in the equation above)

$$\Delta w_{o,h_a} = \alpha \delta_o h_a \qquad (8)$$

where:

- α is the learning rate

$$w_{o,h_a} = w_{o,h_a} + \Delta w_{o,h_a} + \mu \Delta_m w_{o,h_a} \qquad (9)$$

where:

- μ is the momentum factor

Calculate the local gradient at the hidden neurons:

$$\delta_{h_a} = w_{o,h_a} \delta_o \frac{d}{dx} f_{act}(h_a) \qquad (10)$$

where:

- δ_{h_a} is the local gradient at hidden neuron *a*
- w_{o,h_a} is the weight from hidden neuron *a* to output
- h_a is the post-activation value of the hidden neuron *a*

Update the input-to-hidden weights:

$$\Delta_m w_{h_a,i_b} = \Delta w_{h_a,i_b} + \Delta_m w_{h_a,i_b} \qquad (11)$$

where:

- $\Delta w_{h_a,i_b}$ is the adjustment of the weight from input *b* to hidden neuron *a*
- $\Delta_m w_{h_a,i_b}$ is the adjustment of the weight from input *b* to hidden neuron *a*, relative to momentum, which is a cumulative adjustment process.

$$\Delta w_{h_a,i_b} = \alpha \delta_{h_a} i_b \qquad (12)$$

$$w_{h_a,i_b} = w_{h_a,i_b} + \Delta w_{h_a,i_b} + \mu \Delta_m w_{h_a,i_b} \quad (13)$$

Update the output bias:

$$\Delta_m b_o = \Delta b_o + \Delta_m b_o \quad (14)$$

where:

- Δb_o is the adjustment of the output bias
- $\Delta_m b_o$ is the adjustment of the output bias, relative to momentum, which again is a cumulative adjustment process.

$$\Delta b_o = \alpha \delta_o \quad (15)$$

$$b_o = b_o + \Delta b_o + \mu \Delta_m b_o \quad (16)$$

Update the hidden neurons biases:

$$\Delta_m b_{h_a} = \Delta b_{h_a} + \Delta_m b_{h_a} \quad (17)$$

where:

- Δb_{h_a} is the bias adjustment of the hidden neuron *a*
- $\Delta_m b_o$ is the bias adjustment of the hidden neuron *a*, relative to momentum, which again is a cumulative adjustment process.

$$\Delta b_{h_a} = \alpha \delta_{h_a} \quad (18)$$

$$b_{h_a} = b_{h_a} + \Delta b_{h_a} + \mu \Delta_m b_{h_a} \quad (19)$$

4. Pseudocode

Set the input and output vectors, and network parameters

Randomly initialize all weights and biases between -0.5 and 0.5

While loop

    For loop (going through the input vector instances)

        Feedforward propagation

        Calculate instance error

        Backpropagation

    Until all input instances are fed

    Calculate epoch error (Eq. 20)

until total error is less than 0.05

$$\frac{1}{2}\sum_{i=0}^{n} {e_{inst,i}}^2 \qquad (20)$$

III. Results

This sections displays the error curves for three different scenarios:

1. Binary data representation, learning rate $\alpha = 0.2$, momentum factor $\mu = 0$

The graph below displays the error converging to a value lower than 0.05 at 2535th index.

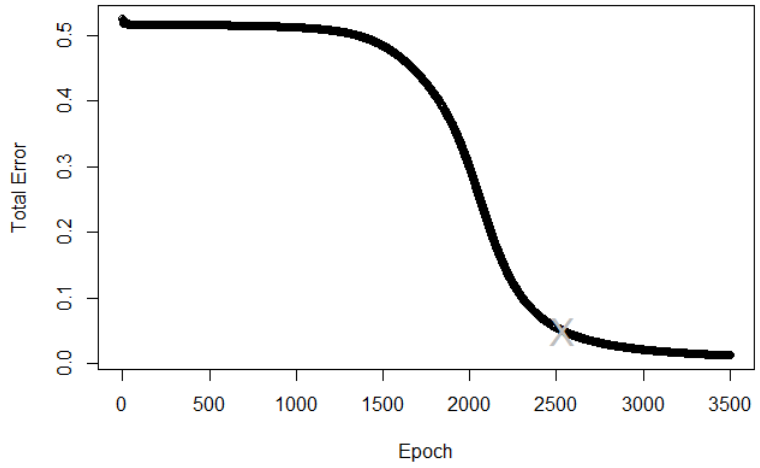

Figure 4: Error curve for binary representation without momentum

2. Bipolar data representation, learning rate $\alpha = 0.2$, momentum factor $\mu = 0$

The graph below shows the error reaching a value of 0.049 at the 215th epoch, which is higher than a 10-fold convergence improvement.

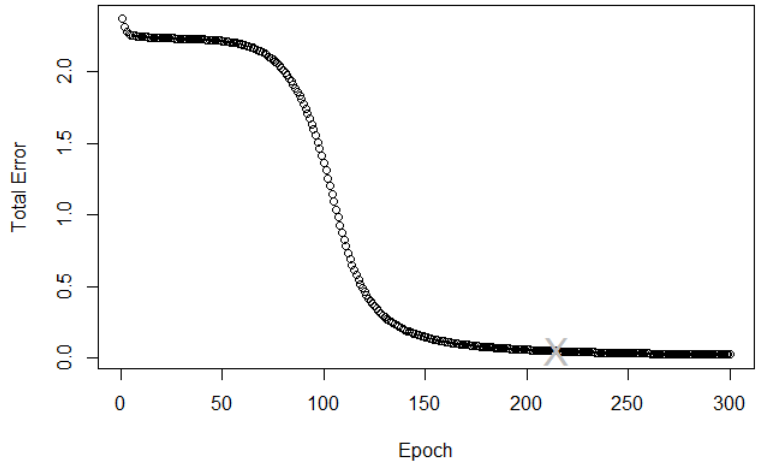

Figure 5: Error curve for bipolar representation without momentum

3. Bipolar data representation, learning rate $\alpha = 0.2$, momentum factor $\mu = 0.9$

The graph below shows the error reaching a value of 0.038 at the 26th epoch, which is around a 10-fold convergence improvement from the previous setting.

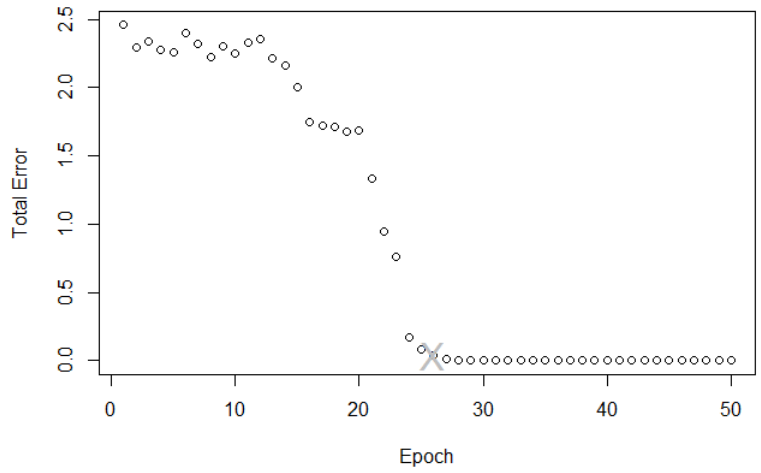

Figure 6: Error curve for bipolar representation with momentum

IV. Conclusion

The conclusion to be made from the results is quite straightforward: While binary data representation allows the backpropagation algorithm to converge, bipolar representation speeds up the process. Coupling the latter with momentum, decreases convergence time considerably.