XOR Logic with a Neural Network on a Raspberry Pi Pico

Two input XOR logic is implemented with an artificial neural network (ANN) (see Figure 1) written in MicroPython on a Raspberry Pi Pico microcontroller (see Figure 2) as an educational exercise of using artificial intelligence to control machinery (LEDs in this example). Project files (including this document) are stored in a GitHub repository (see section 7.0).

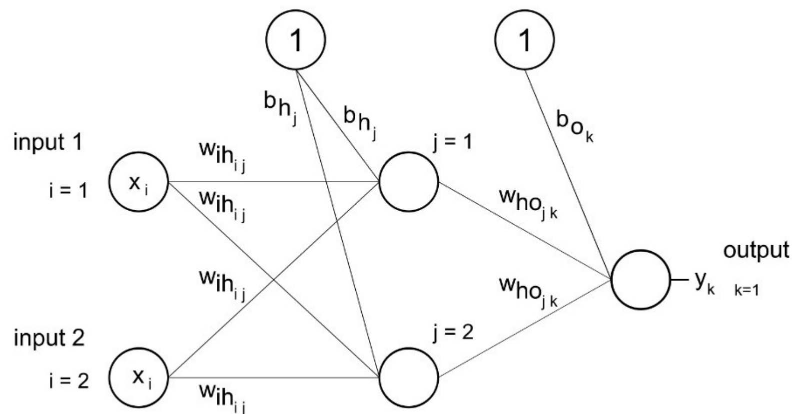

Figure 1 – Artificial neural network for XOR logic

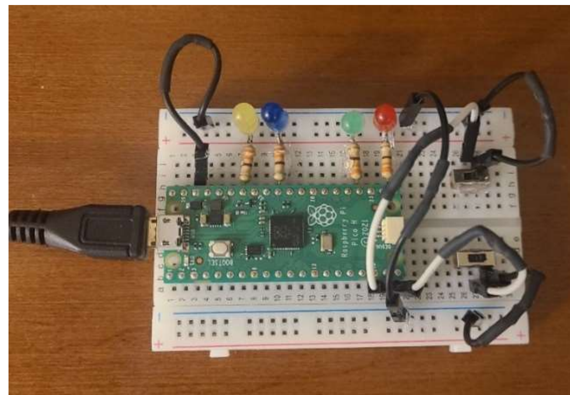

Figure 2 – Raspberry Pi Pico running an artificial neural network

See section 7.0 for details of a GitHub repository containing this document and other resources.

I can be contacted by email for questions and to report errors.

© James S. Canova, jscanova@gmail.com

Victoria, BC

Licence

xor_nn_rasppipico_v0 © 2025 by James S. Canova is licensed under Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International

About the licence: https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode.en

© James S. Canova, jscanova@gmail.com
Victoria, BC

Revision History

Revised on	Version	Description	By
21 February 2025	0	Initial version	JS Canova

© James S. Canova, jscanova@gmail.com

Victoria, BC

Contents

1.0 Introduction 7
2.0 Background 8
3.0 Artificial neural networks 10
3.1 Structure of a biological neural network 10
3.2 Perceptrons 12
3.2.1 Perceptron model 12
3.2.2 Perceptron activation function: 12
3.2.3 Calculating perceptron output 13
3.2.4 Updating weights and biases (with the delta rule) 13
3.2.5 Perceptron example 15
4.0 Artificial Neural Networks 18
4.1 Structure of an ANN 18
5.0 Application of perceptron equations to ANNs 19
5.1 Matrices and NumPy library 19
5.2 Algorithm to calculate weights and biases (training) 22
5.3 Propagation equations 22
5.3.1 Forward propagation 22
5.3.2 Backward propagation 23
5.3.3 Inference 25
6.0 Implementation of an ANN on a microcontroller 26
6.1 XOR logic 26
6.2 ANN model for XOR logic 26
6.3 Hardware 27
6.4 Schematic 28
6.5 Raspberry Pi Pico 28
6.6 IDE 29
6.7 MicroPython code for microcontroller (see Appendix I) 29
6.7.1 Equations in MicroPython 29
6.8 Results 30
7.0 Resources 35
8.0 References 36

© James S. Canova, jscanova@gmail.com
Victoria, BC

Figures

Figure 1 – Artificial neural network for XOR logic 1
Figure 2 – Raspberry Pi Pico running an artificial neural network........ 1
Figure 3 – AI and ANNs [2] 8
Figure 4 - Linear regression [3] 8
Figure 5 – Neuron input and output [5]........10
Figure 6 – Action potential and synapse [6]........10
Figure 7 – Action potential [7]........11
Figure 8 - A perceptron 12
Figure 9 - Sigmoid activation function [9]........13
Figure 10 - A perceptron (copy of Figure 8)........14
Figure 11 – Perceptron for AND logic example 15
Figure 12 - Solution surface for AND logic [ChatGPT]........16
Figure 13 - Decision boundary for AND logic [ChatGPT]........17
Figure 14 – An example of an artificial neural network 18
Figure 15 – ANN model for XOR logic........26
Figure 16 – Assembly 27
Figure 17 - Schematic 28
Figure 18– Error during training neural network 31
Figure 19 - Solution surface for XOR logic [ChatGPT]........32
Figure 20 - Decision boundaries for XOR logic [ChatGPT] 32

© James S. Canova, jscanova@gmail.com
Victoria, BC

Appendices

Appendix I – MicroPython code..37

© James S. Canova, jscanova@gmail.com

Victoria, BC

1.0 Introduction

Artificial neural networks (ANNs) are mathematical models that replicate some capabilities of the human brain.

ANNs can find complex relationships between variables and with current computing technology can do this to make practical applications widespread.

Artificial neural networks can be implemented on microcontrollers to control hardware as demonstrated with the example project presented in this document.

© James S. Canova, jscanova@gmail.com

Victoria, BC

2.0 Background

Artificial intelligence is the science of replicating capabilities of the human brain with computing technology.

Machine learning (ML) is the study of computer algorithms that can improve automatically through experience and by the use of data. [1]

Artificial neural networks[1] are a sub-set of machine learning (see Figure 3).

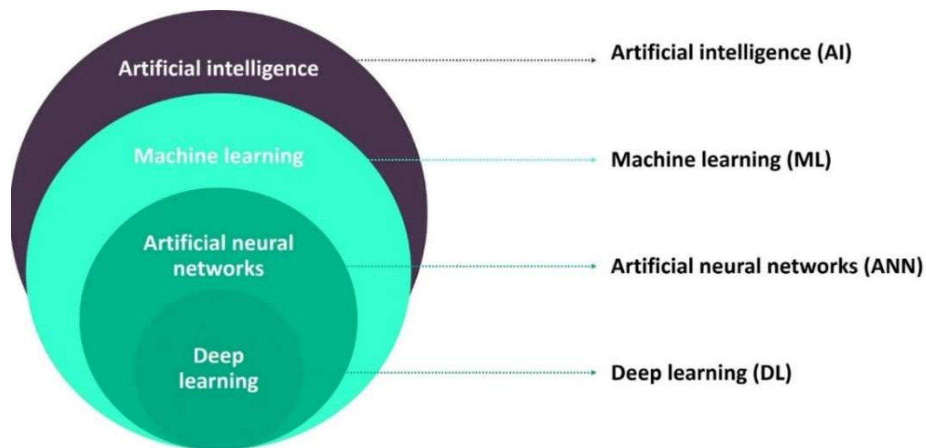

Figure 3 – AI and ANNs [2]

Linear regression is an example of machine learning.

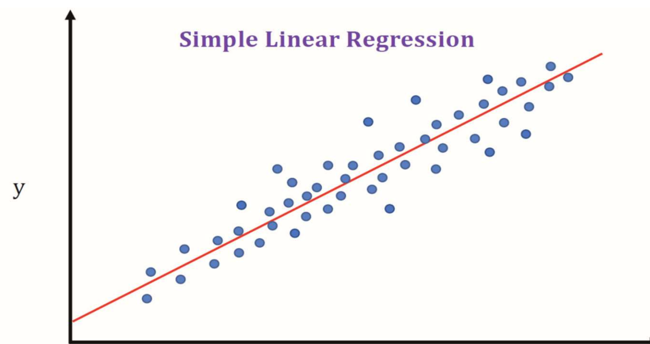

Figure 4 - Linear regression [3]

[1] Deep learning refers to artificial neural networks that have many hidden layers.

© James S. Canova, jscanova@gmail.com
Victoria, BC

The equation $y = ax + b$ relates the output to the input.

The parameters a and b can be chosen (trained) to minimise, for example, the sum of the square of the difference between the true values of y and those predicted by the straight line.

The collection of x, y pairs is called the dataset.

The parameters a and b can be updated programmatically if the dataset changes.

An artificial neural network is an equation that relates the output to the input.

The parameters are called weights and biases. These are determined through a training process using the same error minimization criterion used with linear regression.

Datasets for ANNs may be much more diverse than x, y pairs.

© James S. Canova, jscanova@gmail.com

Victoria, BC

3.0 Artificial neural networks

The structure and function of biological neural network, e.g. a human brain, are mimicked by ANNs.

3.1 Structure of a biological neural network

Neurons are cells that transmit signals from the inputs x to the outputs y (see Figure 5). There are about 86 billion in the human central nervous system with about 16 billion in the cerebral cortex (responsible for higher functions) [4] .

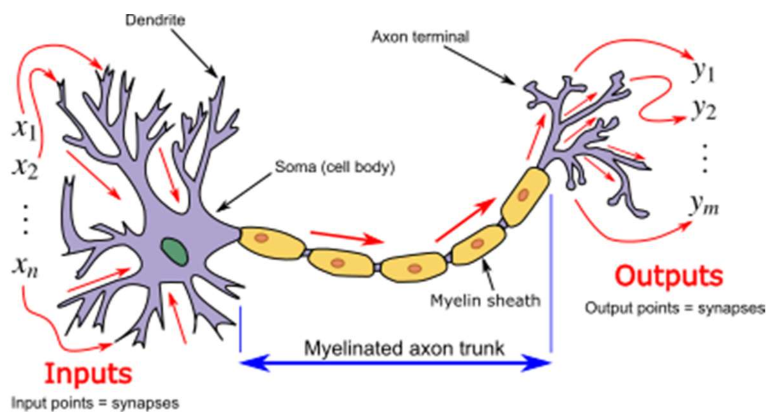

Figure 5 – Neuron input and output [5]

Signals from upstream neurons are received at dendrites. If their sum reaches a threshold value then the neuron generates a transient voltage spike, an action potential (see Figure 6 and Figure 7), which travels towards downstream neurons through axions. The magnitude of the potential that is transmitted to a downstream dendrite (and neuron) is regulated by a synapse (see Figure 6) which includes a gap over which neurotransmitters flow across.

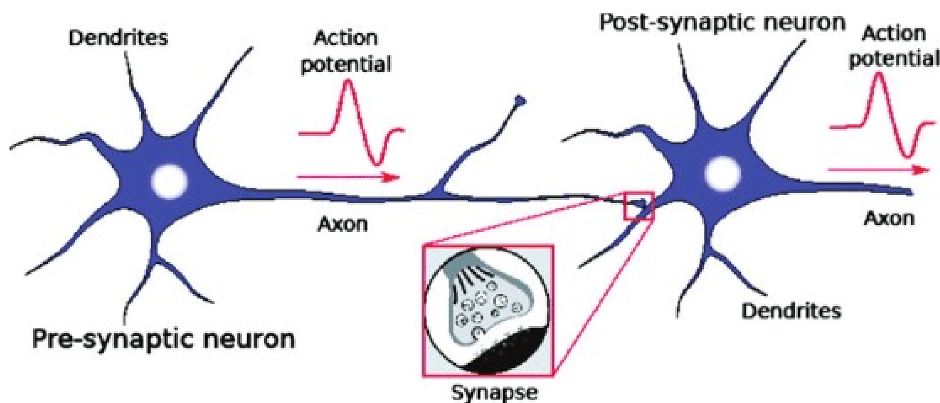

Figure 6 – Action potential and synapse [6]

© James S. Canova, jscanova@gmail.com
Victoria, BC

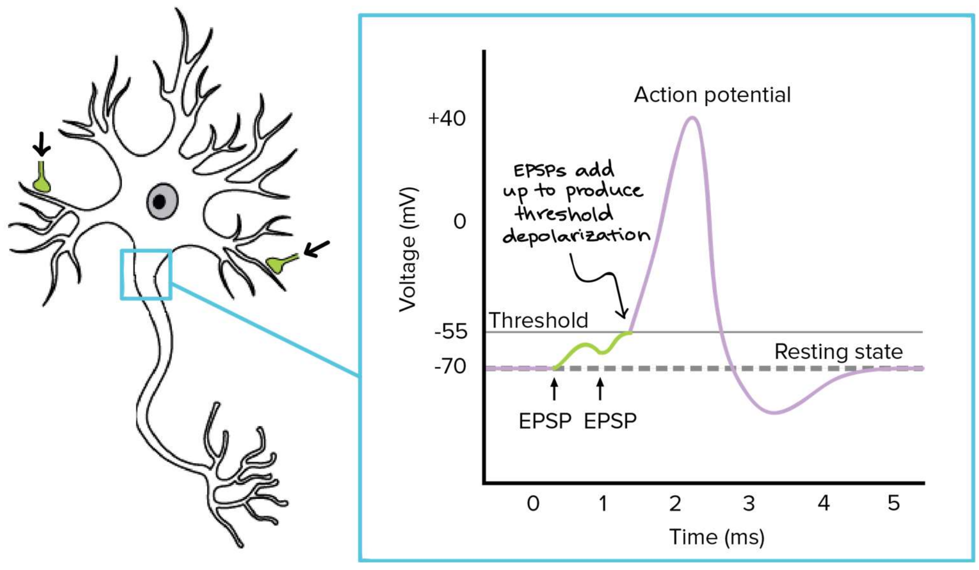

Figure 7 – Action potential [7]

© James S. Canova, jscanova@gmail.com
Victoria, BC

3.2 Perceptrons

Perceptrons are mathematical models of neurons and are the building block of artificial neural networks.

3.2.1 Perceptron model

A perceptron is shown in Figure 8.

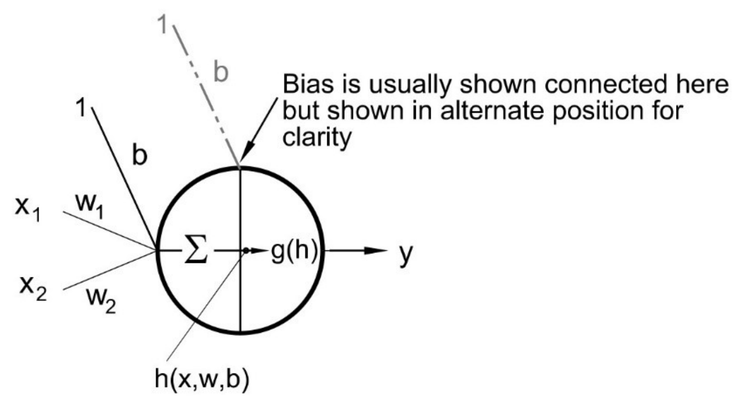

Figure 8 - A perceptron

x_i = inputs
w_i = weights
b = bias
$\mathrm{h}(\mathrm{x_i}, \mathrm{w_i}, \mathrm{b})$ = weighted sum of inputs plus a bias (a scalar)
$g(h) =$ activation function
y = calculated output (a scalar)

3.2.2 Perceptron activation function:

The activation function controls when the perceptron will fire and produce the equivalent of the action potential in a neuron [8].

A very common activation function used with ANNs is the sigmoid function, $\sigma(x)$ (see Figure 9).

© James S. Canova, jscanova@gmail.com
Victoria, BC

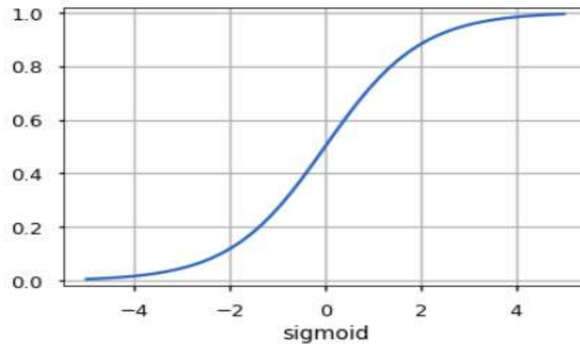

Figure 9 - Sigmoid activation function [9]

The sigmoid function:

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

3.2.3 Calculating perceptron output

$$h(\mathrm{x_i}, \mathrm{w_i}, \mathrm{b}) = \sum_{i=1}^{n} (w_i x_i) + b \qquad \text{eqn. 1}$$

The value of n is the number of inputs.

The output of a perceptron is therefore:

$$\mathrm{g(h)} = \frac{1}{1+e^{-h}} \qquad \text{eqn. 2}$$

3.2.4 Updating weights and biases (with the delta rule)

Weight values provide a functionality similar to synapses controlling the strength of connection between nodes. I could not find and explanation of how a bias is produced in a neuron.

Weights and biases are calculated using an iterative process using the delta rule [10].

This rule calculates the incremental change in upstream weights and biases that will minimize a cost function incorporating the error in the output.

© James S. Canova, jscanova@gmail.com
Victoria, BC

Error in the output:

$e = t - y$
t = target (known value of output for given input, x_i)
y = calculated output of perceptron (see 3.2.3)

Cost function:

$C = \frac{1}{2}e^2$

The delta rule for the case that the activation function is the sigmoid function.

$$\Delta w_j = \alpha e y(1 - y)x_j \quad \text{eqn. 3}$$

α = learning rate (see below)
Δw_j = change that will decrease the error in the output of a perceptron

A bias is updated by using the delta rule by setting the input, , equal to 1.

$$\Delta b = \alpha e y(1 - y)(1) \quad \text{eqn. 4}$$

The learning rate moderates the change in weights and biases during the iterative process to converge on a solution (see section 5.1). If too large, the solution process becomes unstable and if too small, the solution takes an unnecessarily long time.

Each weight and bias can be updated independently of other weights and biases (see Figure 10).

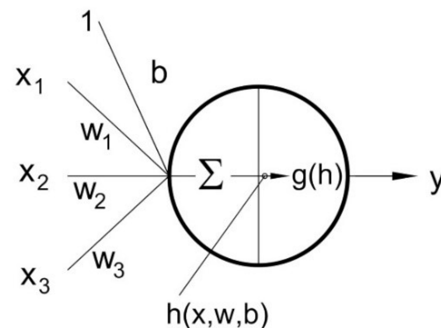

Figure 10 - A perceptron (copy of Figure 8)

© James S. Canova, jscanova@gmail.com
Victoria, BC

3.2.5 Perceptron example

This example illustrates application of the delta rule (eqn. 3 and eqn. 4).

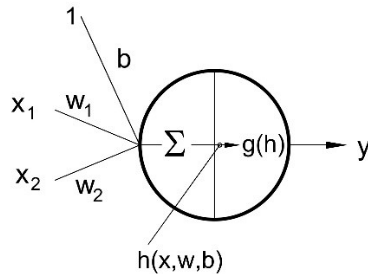

Figure 11 – Perceptron for AND logic example

Table 1- AND logic dataset (4 records)

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

x_1 = Input 1
x_2 = Input 2
t = target value, (Output from Table 1)

Procedure:

1) Initialize weights and bias with random values between -1 and 1
2) Calculate the output, y , and update the error, e,
3) Update weights and bias with the equations below
4) Repeat steps 2 and 3 until results are "correct"

$$\Delta w_1 = \alpha e y(1 - y)x_1$$

$$\Delta w_2 = \alpha e y(1 - y)x_2$$

$$\Delta b = \alpha e y(1 - y)(1)$$

Recall:

y = calculated value of output

$e = t - y$

© James S. Canova, jscanova@gmail.com

Victoria, BC

A Python program[2] with the learning rate, α, equal to 0.5 and using 5,000 iterations (epochs) produced the following values for the weights and the bias:

w_1 = 6.53
w_2 = 6.53
b = -9.88

Using these values, the results were (compare last two columns):

Table 2- Results

x_1	x_2	$w_1 * x_1$	$w_2 * x_2$	b	$h(x_i, w_i, b)$	$\mathbf{y = \sigma(h)}$	$\boldsymbol{t}$
0	0	0	0	-9.8	-9.8	**0**	**0**
0	1	0	6.53	-9.8	-3.27	**0.03**	**0**
1	0	6.53	0	-9.8	-3.27	**0.03**	**0**
1	1	6.53	6.53	-9.8	3.26	**0.96**	**1**

The solution surface of the neural network is shown in Figure 12. It is surface that minimizes the sum of the square of the difference between the true values of the output and those predicted by the surface and therefore is the solution of regression analysis.

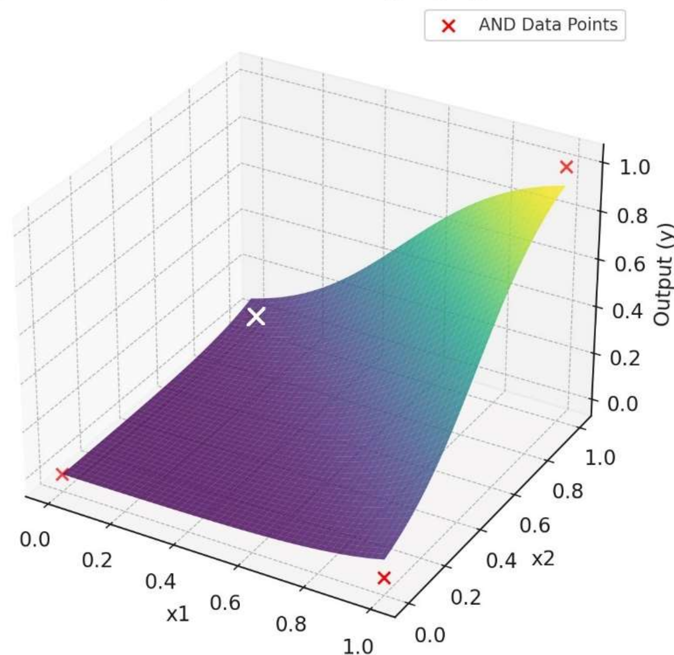

Figure 12 - Solution surface for AND logic [ChatGPT]

[2] Saved in a GitHub repository (see section 7.0)

© James S. Canova, jscanova@gmail.com

Victoria, BC

Further, if a plane is arbitrarily drawn at $z = 0$, then the intersection, called the decision boundary, with the solution curve is viewed in the 2D plane defined by the two inputs (see Figure 13), it is seen that the data point with a value of 1 is separated from the datapoints with a value of 0. This is known as classification.

Classification is a practical application of neural networks. For example, if the inputs were two inspection dimensions of a manufactured part then the decision boundary could separate accepted from rejected parts.

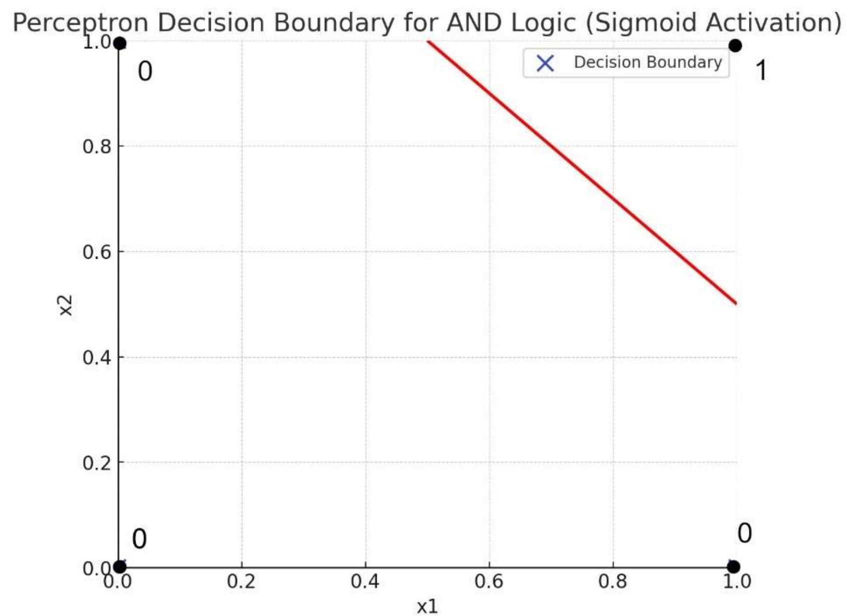

Figure 13 - Decision boundary for AND logic [ChatGPT]

© James S. Canova, jscanova@gmail.com

Victoria, BC

4.0 Artificial Neural Networks

4.1 Structure of an ANN

Perceptrons can be assembled into networks to solve more complex problems (see Figure 14).

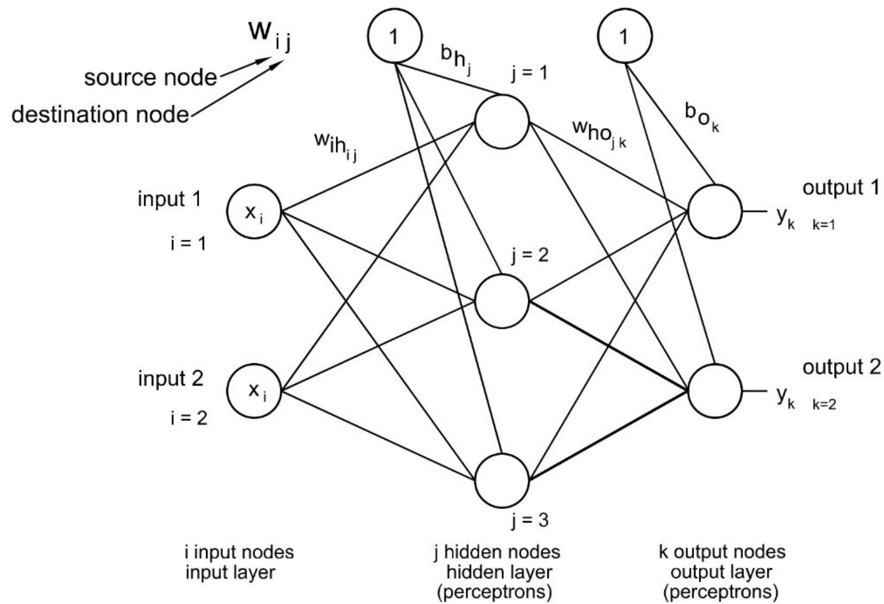

Figure 14 – An example of an artificial neural network

Nodes are arranged in layers and adjacent layers are connected by connections which are also called links or edges.

Inputs are on the left and outputs on the right. The number of nodes in each layer depends on the application. There may one or, more commonly, more than one hidden layer.

The w values are called weights. The b values are called biases and are not always required.

The circles are referred to as nodes. Nodes in the hidden and output layers are perceptrons (see section 3.2).

© James S. Canova, jscanova@gmail.com

Victoria, BC

5.0 Application of perceptron equations to ANNs

This section is substantially based on ref. [11].

For variables, refer to Figure 8 and Figure 14.

5.1 Matrices and NumPy library

The equations used for artificial neural network calculations are described with (NumPy) matrices. Matrix operations are implemented in Python and in MicroPython (the language used for this project) are implemented in the NumPy library.

This is a matrix, A:

$$A = \begin{bmatrix} 3 & 10 & 8 \\ 15 & 2 & 6 \end{bmatrix}$$

It has 2 rows and 3 columns. It dimensions are written as 2 x 3.

Individual elements are identified by their row and column numbers. The row number is the first index i and the second index is the column j. The top left element is in row 1 and column 1, e.g., $a_{11} = 3$.

So, A can be written as:

$$\mathrm{A} = \begin{bmatrix} \mathrm{a}_{11} & \mathrm{a}_{12} & \mathrm{a}_{13} \\ \mathrm{a}_{21} & \mathrm{a}_{22} & \mathrm{a}_{23} \end{bmatrix}$$

To access an element with NumPy: `a[i][j]`

<u>Dot product</u>

The dot product between two matrices is written:

$$C = A \cdot B$$

If matrix B is:

$$\mathrm{B} = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix}$$

Then the dot product is:

© James S. Canova, jscanova@gmail.com
Victoria, BC

$$\mathrm{C} = \begin{bmatrix} a_{11}\mathrm{b}_{11} + a_{12}\mathrm{b}_{21} + a_{13}\mathrm{b}_{31} \\ a_{21}\mathrm{b}_{11} + a_{22}\mathrm{b}_{21} + a_{23}\mathrm{b}_{31} \end{bmatrix}$$

This has dimension of: 2 x 1

The dot product can be made if the number of columns of the first matrix A, has the same number of row as the second matrix B. The dimensions of the resulting matrix C has the same number of rows as the first matrix and the same number of columns the second matrix.

For the example above:

2 x **1** = **2** x 3 • 3 x **1**

To take a dot product with NumPy: `C = np.dot(A,B)`

Row by row multiplication

Matrices with the same number of rows and each with one column can be multiplied row by row:

$$\mathrm{D} = \begin{bmatrix} d_{11} \\ d_{21} \\ d_{31} \end{bmatrix}$$

$$E = B * D$$

$$\mathrm{E} = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} * \begin{bmatrix} d_{11} \\ d_{21} \\ d_{31} \end{bmatrix}$$

$$\mathrm{E} = \begin{bmatrix} b_{11}\mathrm{d}_{11} \\ b_{21}\mathrm{d}_{21} \\ b_{31}\mathrm{d}_{31} \end{bmatrix}$$

To complete a row by row multiplication with NumPy: `E = B * D`

© James S. Canova, jscanova@gmail.com
Victoria, BC

Transpose

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

Transpose is obtained by switching row and column indexes for each element.

$$F = A^T$$

$$\mathrm{F} = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \\ a_{13} & a_{23} \end{bmatrix}$$

To complete take a transpose with NumPy:

```
F = A.T
```

or:

```
F = np.transpose(A)
```


© James S. Canova, jscanova@gmail.com

Victoria, BC

5.2 Algorithm to calculate weights and biases (training)

The perceptron equations are applied to a complete network to first calculate the outputs from a network (forward propagation) and then update all weights and biases (backward propagation).

Training is an iterative process:

1) Initialization of all weights and biases with random numbers between -1 and 1

2) Forward propagation (calculate outputs)

3) Backwards propagation (update weights and biases)

4) Repeated application of steps 2 and 3 until the cost function is minimized.

Each iteration is called an epoch. It is important to note that the complete dataset is processed during each epoch. The number of epochs can be determined by monitoring the cost function (see Figure 18).

Once trained and the weights and biases are known, the neural network is ready to use with a forward propagation step. This is known as inference.

5.3 Propagation equations

For variables in this section, refer to Figure 14.

5.3.1 Forward propagation

Forward propagation is the process of calculating the output for a given input. Working from left to right, output from the nodes from upstream layers are combined in the next layer of nodes using the equations for a perceptron (eqn. 1 and eqn. 2 section 3.2.3).

Calculate the outputs from the hidden layer:

The x values are the inputs to the neural network.

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \begin{bmatrix} w_{ih_{11}} & w_{ih_{21}} \\ w_{ih_{12}} & w_{ih_{22}} \\ w_{ih_{13}} & w_{ih_{23}} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_{h1} \\ b_{h2} \\ b_{h3} \end{bmatrix}$$

© James S. Canova, jscanova@gmail.com
Victoria, BC

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = g \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}$$

The y values are then used as the input (x values) into the output layer.

Calculate the output from output layer:

$$\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} w_{ho_{11}} & w_{ho_{21}} & w_{ho_{31}} \\ w_{ho_{12}} & w_{ho_{22}} & w_{ho_{32}} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_{o1} \\ b_{o2} \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = g \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

The y values are the output from the output layer.

5.3.2 Backward propagation

Backward propagation is the process of updating all weights and biases using the delta rule (eqn. 3. and eqn. 4 section 3.2.4). For layers with multiple perceptrons, the equation is modified to (j is input index, k is output Index):

$$\Delta w_{jk} = \alpha e_k y_k (1 - y_k) x_{jk}$$

Weights and biases are updated working from right to left.

Output layer to hidden layer:

Values of t are target values (known).

$$\begin{bmatrix} e_{o_1} \\ e_{o_2} \end{bmatrix} = \begin{bmatrix} t_1 - y_1 \\ t_2 - y_2 \end{bmatrix}$$

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \alpha \begin{bmatrix} e_{o_1} \\ e_{o_2} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \begin{bmatrix} 1 - y_1 \\ 1 - y_2 \end{bmatrix}$$

Updating weights:

The x values are outputs y from hidden layer.

© James S. Canova, jscanova@gmail.com

Victoria, BC

$$\begin{bmatrix} \Delta w_{ho_{11}} & \Delta w_{ho_{21}} & \Delta w_{ho_{31}} \\ \Delta w_{ho_{12}} & \Delta w_{ho_{22}} & \Delta w_{ho_{32}} \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$$

Updating biases:

$$\begin{bmatrix} \Delta b_{o1} \\ \Delta b_{o2} \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$ because x values are 1

Hidden layer to input layer:

Error from hidden layer [11] , pp.81, 82, 140:

$$e_h = w_{ho}^T \cdot e_o$$

$$\begin{bmatrix} e_{h_1} \\ e_{h_2} \\ e_{h_3} \end{bmatrix} = \begin{bmatrix} \Delta w_{ho_{11}} & \Delta w_{ho_{12}} \\ \Delta w_{ho_{21}} & \Delta w_{ho_{22}} \\ \Delta w_{ho_{31}} & \Delta w_{ho_{32}} \end{bmatrix} \cdot \begin{bmatrix} e_{o_1} \\ e_{o_2} \\ e_{o_3} \end{bmatrix}$$

The y values are output from hidden layer.

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \alpha \begin{bmatrix} e_{h_1} \\ e_{h_2} \\ e_{h_3} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \begin{bmatrix} 1 - y_1 \\ 1 - y_2 \\ 1 - y_3 \end{bmatrix}$$

Updating weights:

$$\begin{bmatrix} \Delta w_{ih_{11}} & \Delta w_{ih_{21}} \\ \Delta w_{ih_{12}} & \Delta w_{ih_{22}} \\ \Delta w_{ih_{13}} & \Delta w_{ih_{23}} \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} \begin{bmatrix} x_1 & x_2 \end{bmatrix}$$

Updating biases:

$$\begin{bmatrix} \Delta b_{h_1} \\ \Delta b_{h_2} \\ \Delta b_{h_3} \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$ because x values are 1

© James S. Canova, jscanova@gmail.com
Victoria, BC

5.3.3 Inference

Inference is the process of a single feedforward propagation using the final values of weights and biases (see section 5.3.1).

© James S. Canova, jscanova@gmail.com
Victoria, BC

6.0 Implementation of an ANN on a microcontroller

6.1 XOR logic

The logic for a two-input logic gate is in Table 3.

Table 3- XOR logic dataset (4 records)

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

x_1 = Input 1
x_2 = Input 2
t = target value, (output from Table 3)

6.2 ANN model for XOR logic

The model for this problem can be found on the internet. It is commonly used for explaining how neural networks work.

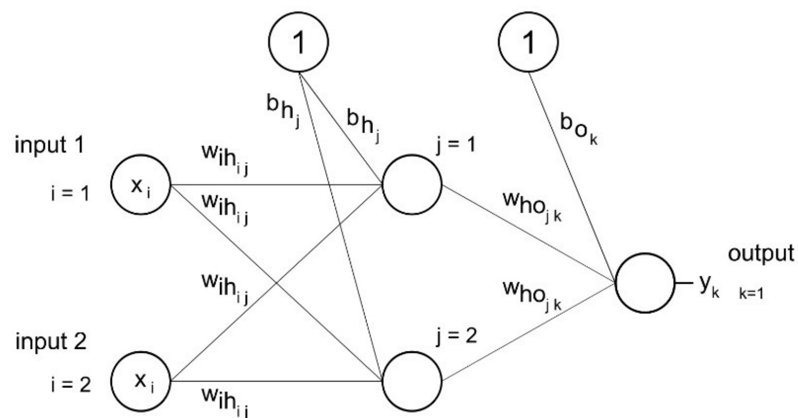

Figure 15 – ANN model for XOR logic

© James S. Canova, jscanova@gmail.com

Victoria, BC

6.3 Hardware

The project assembly is shown in Figure 16.

When the Raspberry Pi Pico is reset, the neural network begins to train. The red LED is on until the training process completes which takes about 2.5 [minutes]. When training is complete, the red LED turns off and the green LED turns on.

Following training, the program enters an infinite loop during which the program polls the positions of the two input switches (0 or 1) and calculates the output (0 or 1). If the output from the neural network is 1 then the blue LED is turned on otherwise the output is 0 and the yellow LED is turned on.

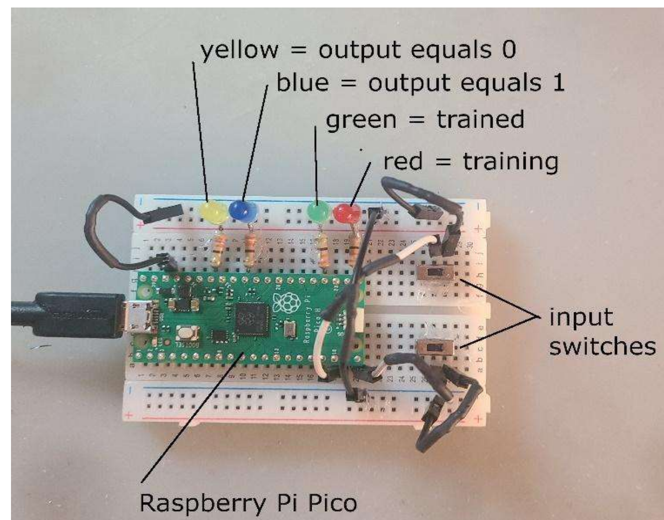

Figure 16 – Assembly

© James S. Canova, jscanova@gmail.com
Victoria, BC

6.4 Schematic

Schematic is shown below (see Figure 17). All components are commonly found.

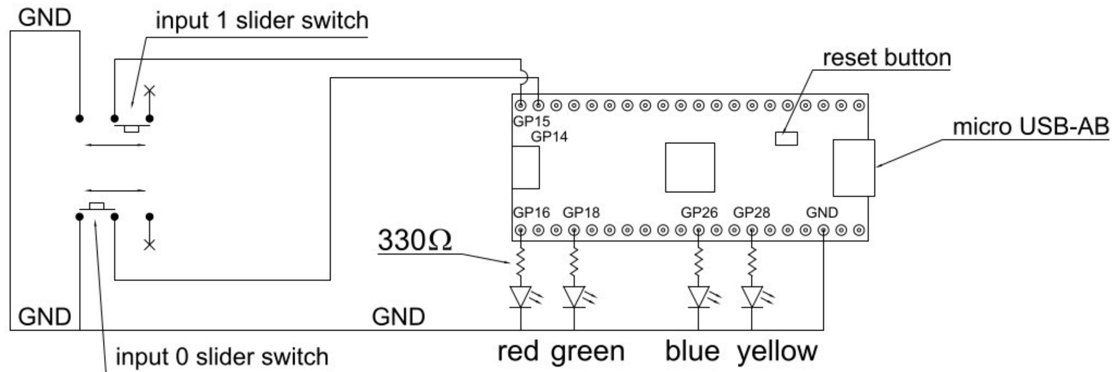

Figure 17 - Schematic

6.5 Raspberry Pi Pico

Raspberry Pi Pico specifications are tabulated below (see Table 4).

Table 4 – Raspberry Pi Pico

CPU	RP2040: 32bit, dual-core Arm Cortex-M0+ processor, 133MHz
RAM	254 KB
Flash	2 MB
Price (Amazon.ca)	$15.95CAD (with headers)

MicroPython must be loaded on to the microcontroller. This is accomplished using a uf2 format file and various builds are available on the internet (see b. below). A version of uf2 built to include the ulab library because it contains a compact version of the Python NumPy library. The NumPy library includes the variable types and methods required for the many matrix calculations required for neural network calculations.

a) Set-up instructions:
https://projects.raspberrypi.org/en/projects/getting-started-with-the-pico

b) Download uf2[3] with ulab from here (RPI_PICO.uf2):
https://github.com/v923z/micropython-builder/releases

[3] UF2 (USB Flashing Format) is a format designed by Microsoft to flash firmware on microcontrollers

© James S. Canova, jscanova@gmail.com
Victoria, BC

c) To load uf2 (that includes ulab library) on to Raspberry Pi Pico:
https://www.raspberrypi.com/documentation/microcontrollers/micropython.html

The host operating system can be either a Linux distribution or Windows.

6.6 IDE

Thonny (installation and use):
https://projects.raspberrypi.org/en/projects/getting-started-with-the-pico/2

6.7 MicroPython code for microcontroller (see Appendix I)

There is one class, it is for the neural network and includes a constructor plus methods to train and infer results.

The dataset for training is contained within the program.

6.7.1 Equations in MicroPython

Forward propagation equations (see section 5.3.1)

Forward propagation is simply applying the input and calculating all downstream values to calculate a predicted output.

```
128:            #forward propagation-----
129:            #calculate hidden outputs from inputs
130:            hidden_sums = np.dot(self.wih, input) + self.bih # 2 x 2 . 2 x 1 + 2 x 1
= 2 x 1
131:            hidden_outputs = sigmoid(hidden_sums)  # 2 x 1
132:
133:            #calculate predicted output from hidden outputs
134:            output_sum = np.dot(self.who, hidden_outputs) + self.bho # 1 x 2 . 2 x 1
+ 1 x 1 = 1 x 1
135:            final_output = sigmoid(output_sum)  # 1 x 1
```


© James S. Canova, jscanova@gmail.com
Victoria, BC

Backward propagation equations (see section 5.3.2)

```
138:            #backward propagation-----
139:            #update weights for hidden to output layer
140:            output_error = target - final_output   #1 x 1 - 1 x 1
141:            dWho = self.lr * np.dot((output_error * final_output * (1.0 -
final_output)), hidden_outputs.T) # 1 x 1 . 1 x 2 = 1 x 2
142:            self.who += dWho # 1 x 2
143:
144:            #update bias for output layer
145:            dbho = self.lr * output_error * (final_output * (1.0 - final_output)) # 1
x 1
146:            self.bho += dbho # 1 x 1
147:
148:            #update weights for hidden layer
149:            hidden_error = np.dot(self.who.T, output_error) # 2 x 1 . 1 x 1 = 2 x 1
[Rashid, pp.81,82,140]

150:            dWih = self.lr * np.dot((hidden_error * hidden_outputs * (1.0 -
hidden_outputs)), input.T) # 2 x 2
151:            self.wih += dWih  # 2 x 2
152:
153:            #update biases for input to hidden layer
154:            dbih = self.lr * hidden_error * (hidden_outputs * (1.0 - hidden_outputs))
# 2 x 1
155:            self.bih += dbih  # 2 x 1
```

Inference equations (see section 5.3.1)

```
160:    # to infer output from inputs
161:    def infer(self, input):
162:
163:      #calculate hidden outputs from inputs
164:      hidden_sums = np.dot(self.wih, input) + self.bih  # 2 x 2 . 2 x 1 + 2 x 1 = 2 x
1
165:      hidden_outputs = sigmoid(hidden_sums)  # 2 x 1
166:
167:      #calculate predicted output from hidden outputs
168:      output_sum = np.dot(self.who, hidden_outputs) + self.bho   # 1 x 2 . 2 x 1 + 1
x 1 = 1 x 1
169:      inferred_output = sigmoid(output_sum)  # 1 x 1
170:
171:      return inferred_output # 1 x 1
```

6.8 Results

The project works as intended.

During training, the cost (error) is calculated during each epoch, i.e. during each update of weights and biases.

Training used a total of 20,000 epochs and took about 2.5 [min.].

From the graph, the cost in the results is about 5%.

© James S. Canova, jscanova@gmail.com
Victoria, BC

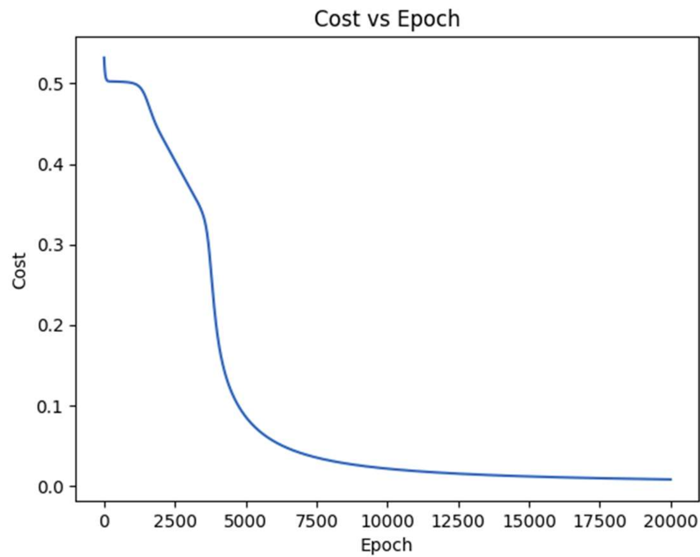

Figure 18– Error during training neural network[4]

The solution surface is shown in Figure 19 and the decision boundary is shown in Figure 20 and has 2 decision boundaries as opposed to 1 as in the case of AND logic (see section 3.2.5). A perceptron can only define 1 decision boundary as opposed the 2 required for OR logic which explains the more complicated neural network model used for the XOR logic.

[4] Plotted with a Python program in Google Colab

© James S. Canova, jscanova@gmail.com File name: xor_nn_rasppipico_v0
Victoria, BC

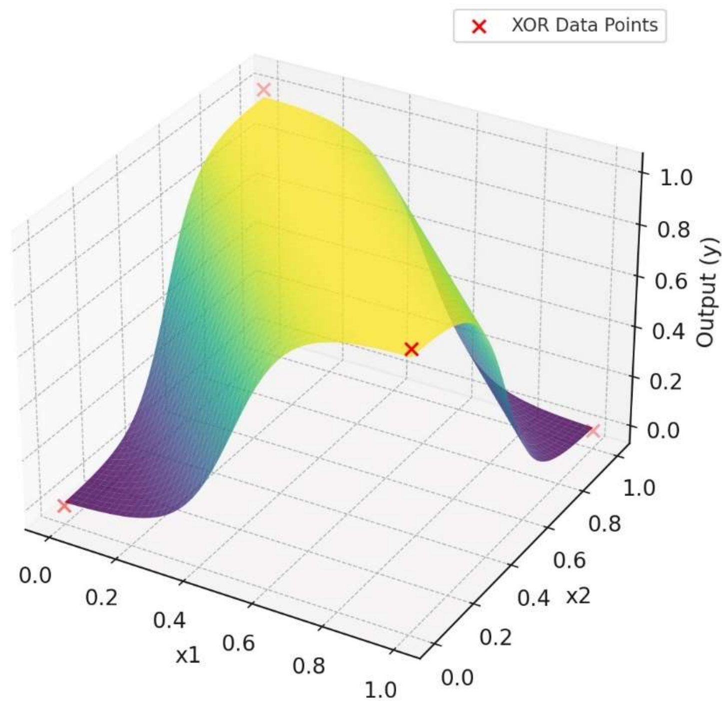

Figure 19 - Solution surface for XOR logic [ChatGPT]

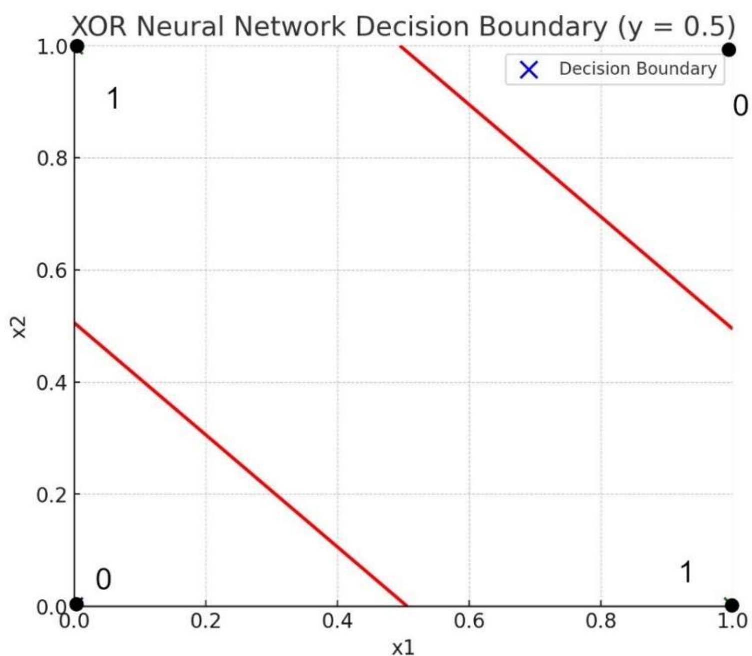

Figure 20 - Decision boundaries for XOR logic [ChatGPT]

© James S. Canova, jscanova@gmail.com
Victoria, BC

As more layers are added, the capabilities of artificial neural networks extend from regression and classification to identification of features and recognition of patterns both spatially (such as images) and in time or in sequences (such as languages).

© James S. Canova, jscanova@gmail.com

Victoria, BC

Note:

There is a line in the MicroPython program:

```
rnd.seed(30)
```

The random number generator for initializing weights and biases had to be seeded with a particular value (30) for the solution to converge. This should not be the case. Same was not true for a version that I wrote in Python. I have no explanation for this.

© James S. Canova, jscanova@gmail.com

Victoria, BC

7.0 Resources

This document in pdf format, the MicroPython code, UF2 file and other resources are stored in a GitHub repository:

https://github.com/James-Canova/XOR_NN.git

© James S. Canova, jscanova@gmail.com

Victoria, BC

8.0 References

[1] "Wikipedia (Machine learning)," [Online]. Available: https://en.wikipedia.org/wiki/Machine_learning.

[2] "bbntimes," [Online]. Available: https://www.bbntimes.com/science/artificial-intelligence-vs-machine-learning-vs-artificial-neural-networks-vs-deep-learning.

[3] "Medium," [Online]. Available: https://medium.datadriveninvestor.com/machine-learning-101-part-1-24835333d38a.

[4] "Wikipedia (Neurons)," [Online]. Available: https://en.m.wikipedia.org/wiki/List_of_animals_by_number_of_neurons#.

[5] "Wikipedia (Neuron model)," [Online]. Available: https://en.wikipedia.org/wiki/Biological_neuron_model.

[6] "ResearchGate (Neuron)," [Online]. Available: https://www.researchgate.net/figure/Biological-neuron-and-synapse_fig1_324229756.

[7] "Kahn (Action potential)," [Online]. Available: https://www.khanacademy.org/science/biology/human-biology/neuron-nervous-system/a/the-synapse.

[8] "Wikipedia (Action potential)," [Online]. Available: https://en.wikipedia.org/wiki/Action_potential.

[9] "OpenCV," [Online]. Available: https://learnopencv.com/understanding-feedforward-neural-networks/.

[10] "Wikipedia (Delta rule)," [Online]. Available: https://en.wikipedia.org/wiki/Delta_rule.

[11] T. Rashid, Make Your Own Neural Network, Illustrated edition ed., CreateSpace Independent Publishing Platform, 2016.

© James S. Canova, jscanova@gmail.com
Victoria, BC

Appendix I – MicroPython code

```
 1: #main.py (Version 0)
 2:
 3: #Micropython and Raspberry Pi Pico
 4: #Date created: 19 January 2024
 5: #last updated: 19 February 2025
 6:
 7: #James Canova
 8: #jscanova@gmail.com
 9:
10: #Based on:
11: #Rashid, Make your own Neural Network
12: # https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/
13:
14: #This program solves two input (one output) XOR logic using a neural network
15:
16: import micropython
17: from machine import Pin
18: from ulab import numpy as np  #for matrix operations
19: import random as rnd          #to initialize weights and biases
20:
21: #--------------------------------------------------------
22: #Hyperparameters (i.e. they control the solution)
23: #note: these were selected by trial and error
24: LEARNING_RATE = 0.08
25: EPOCHS = 20000
26:
27: #initialize random number generator
28: rnd.seed(30)
29:
30:
31: #setup LEDs
32: #red LED on: not trained
33: #green LED on: trained
34: #blue LED on: output == 1
35: #yellow LED on: output == 0
36: ledRed = Pin(16, Pin.OUT)
37: ledGreen = Pin(18, Pin.OUT)
38: ledBlue = Pin(26, Pin.OUT)
39: ledYellow = Pin(28, Pin.OUT)
40:
41:
42: #setup slider switches for inputs
43: slider1 = Pin(14, Pin.IN, Pin.PULL_UP)
44: slider2 = Pin(15, Pin.IN, Pin.PULL_UP)
45:
46:
47: #activation function
48: def sigmoid (x):
49:
50:     return 1/(1 + np.exp(-x))
51:
52: pass
53:
54:
55: #for initializing weights and biases between and including:-1, 1
56: #shape contains dimensions of required matrix
57: def create_random_array(shape):
58:
59:     new_array = np.zeros(shape)
60:
61:     nRows = shape[0]
62:     nColumns = shape[1]
63:
64:     for i in range(nRows):
65:         for j in range(nColumns):
66:             new_array[i][j] += rnd.uniform(-1, 1)
67:     return new_array
68:
```


© James S. Canova, jscanova@gmail.com
Victoria, BC

```
 69: 
 70: #Neural network class
 71: class neuralNetwork:
 72:     # initialise the neural network
 73:     # note: one hidden layer only
 74:     # inputnodes = number of input nodes
 75:     # hiddennodes = number of hidden nodes
 76:     # ouptutnodes = numper of output nodes
 77: 
 78:     #member functions:
 79:     #-init
 80:     #-train
 81:     #-infer
 82: 
 83:     #--------------------------------------------------------------------------------
 84:     def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate, epochs):
 85: 
 86:         # set number of nodes in each input, hidden, output layer
 87:         self.inodes = inputnodes
 88:         self.hnodes = hiddennodes
 89:         self.onodes = outputnodes
 90:         self.epochs = epochs
 91: 
 92:         #Initialize weights and biases with random numbers, -1 <= num <= 1
 93:         self.wih = create_random_array((hiddennodes, inputnodes))   # 2 x 2
 94:         self.who = create_random_array((outputnodes, hiddennodes))  # 1 x 2
 95: 
 96:         self.bih = create_random_array((hiddennodes,1))  # 2 x 1
 97:         self.bho = create_random_array((outputnodes,1))  # 1 x 1
 98: 
 99:        # learning rate
100:         self.lr = learningrate
101: 
102:         # number of epochs
103:         self.epochs = epochs
104: 
105:         #flag for training
106:         self.bTrained = False
107: 
108: 
109:     #------------------------------------------------------------------------------
110:     # train
111:     # inputs is a 4 x 2 numpy array, each row is a pair of input values
112:     # targets is a 4 x 1 numpy array
113:     def train(self, inputs, targets):
114: 
115:         # interate through epochs
116:         for c1 in range(self.epochs):
117: 
118:           epoch_cost = 0.0  # initialize cost per epoch (for plotting)
119: 
120:           # interate through 4 inputs
121:           for c2 in range(inputs.shape[0]):  #inputs.shape[0] equals the number of input 
pairs which is 4
122: 
123:             input = inputs[c2,:]   # 1 x 2
124:             input = input.reshape((2, 1)) # 2 x 1
125: 
126:             target = targets[c2,:]   #target is a 1D numpy array
127: 
128:             #forward propagation-----
129:             #calculate hidden outputs from inputs
130:             hidden_sums = np.dot(self.wih, input) + self.bih # 2 x 2 . 2 x 1 + 2 x 1 = 2 x 
1
131:             hidden_outputs = sigmoid(hidden_sums)  # 2 x 1
132: 
133:             #calculate predicted output from hidden outputs
134:             output_sum = np.dot(self.who, hidden_outputs) + self.bho # 1 x 2 . 2 x 1 + 1 x 
1 = 1 x 1
135:             final_output = sigmoid(output_sum)  # 1 x 1
136: 
```


© James S. Canova, jscanova@gmail.com File name: xor_nn_rasppipico_v0
Victoria, BC

```
 137:
 138:             #backward propagation-----
 139:             #update weights for hidden to output layer
 140:             output_error = target - final_output   #1 x 1 - 1 x 1
 141:             dWho = self.lr * np.dot((output_error * final_output * (1.0 - final_output)),
hidden_outputs.T) # 1 x 1 . 1 x 2 = 1 x 2
 142:             self.who += dWho # 1 x 2
 143:
 144:             #update bias for output layer
 145:             dbho = self.lr * output_error * (final_output * (1.0 - final_output)) # 1 x 1
 146:             self.bho += dbho # 1 x 1
 147:
 148:             #update weights for hidden layer
 149:             hidden_error = np.dot(self.who.T, output_error) # 2 x 1 . 1 x 1 = 2 x 1
[Rashid, pp.81,82,140]
 150:             dWih = self.lr * np.dot((hidden_error * hidden_outputs * (1.0 -
hidden_outputs)), input.T) # 2 x 2
 151:             self.wih += dWih  # 2 x 2
 152:
 153:             #update biases for input to hidden layer
 154:             dbih = self.lr * hidden_error * (hidden_outputs * (1.0 - hidden_outputs)) # 2 x
1
 155:             self.bih += dbih  # 2 x 1
 156:
 157:         self.bTrained = True
 158:
 159:     #---------------------------------------------------------------------------
 160:     # to infer output from inputs
 161:     def infer(self, input):
 162:
 163:       #calculate hidden outputs from inputs
 164:       hidden_sums = np.dot(self.wih, input) + self.bih  # 2 x 2 . 2 x 1 + 2 x 1 = 2 x 1
 165:       hidden_outputs = sigmoid(hidden_sums)  # 2 x 1
 166:
 167:       #calculate predicted output from hidden outputs
 168:       output_sum = np.dot(self.who, hidden_outputs) + self.bho   # 1 x 2 . 2 x 1 + 1 x 1 =
1 x 1
 169:       inferred_output = sigmoid(output_sum)  # 1 x 1
 170:
 171:       return inferred_output # 1 x 1
 172:
 173:
 174: #=====================================================================
 175: #main program
 176: #set LEDs to indicate neural network has not been trained
 177: ledRed.on()
 178: ledGreen.off()
 179: ledBlue.off()
 180: ledYellow.off()
 181:
 182:
 183: # initialize neural network------
 184: # number of input, hidden and output nodes
 185: input_nodes = 2
 186: hidden_nodes = 2
 187: output_nodes = 1
 188:
 189:
 190: # create instance of neural network
 191: nn = neuralNetwork(input_nodes, hidden_nodes, output_nodes, LEARNING_RATE, EPOCHS)
 192:
 193:
 194: #train the neural network-------
 195: #define the training dataset, which are inputs and targets (outputs)
 196: #define inputs and targets for training and infer
 197: inputs_array= np.array([[0,0],[0,1],[1,0],[1,1]])  # 4 x 2
 198: targets_array = np.array([[0],[1],[1],[0]])    # 4 x 1
 199:
 200: #train
 201: nn.train(inputs_array, targets_array);
 202:
```


© James S. Canova, jscanova@gmail.com
Victoria, BC

```
203: #indicate that the neural network has been trained
204: ledRed.off()
205: ledGreen.on()
206:
207:
208: #main loop=================================================================
209: while True:
210:
211:     #read state of slider switches (inputs)
212:     nValueInput0 = slider1.value()
213:     nValueInput1 = slider2.value()
214:
215:     #format inputs for passing to neural network function for inferring results
216:     inputs_list = np.zeros((2,1))
217:     inputs_list[0,0]= nValueInput0;
218:     inputs_list[1,0]= nValueInput1;
219:
220:     #infer result (0 or 1)
221:     final_output= nn.infer(inputs_list)
222:     nInferredResult = int(round(final_output[0,0])) #round to nearest of 0 or 1
223:
224:     #display result using LEDS
225:     if nInferredResult == 1:
226:         ledBlue.on()
227:         ledYellow.off()
228:
229:     elif nInferredResult == 0:
230:         ledYellow.on()
231:         ledBlue.off()
232:
233:     else:
234:         pass
235:
236:     pass
237:
238: pass    #end of infinite loop
239:
```


© James S. Canova, jscanova@gmail.com

Victoria, BC