

# **Neural Networks in Artificial Intelligence**

**Artificial intelligence is the process of using computers to analyze complex information in order to assist with making decisions.**

*"Is that really you looking at me?"*



**There are many approaches to artificial intelligence.**

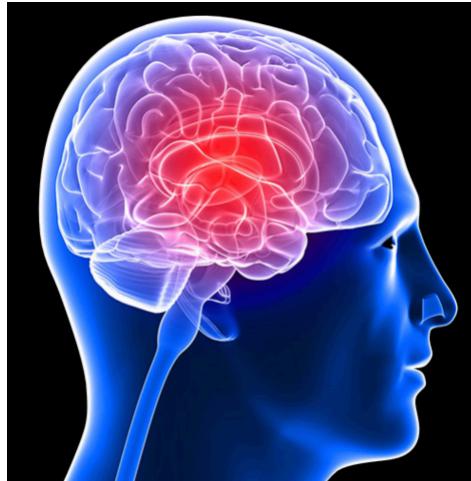
**The approach we introduce here is the use of [artificial neural networks](#).**

**There are software tools available which allow people to apply neural networks to solve problems without having to understand what is going on inside the tools.**

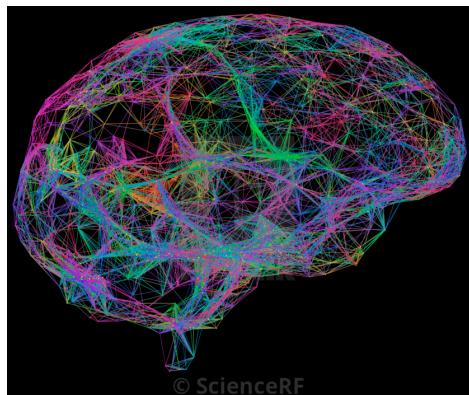
**Our purpose is to explain the computational aspects behind simple artificial neural networks.**

**The goal is an understanding of the basics which underly the software tools.**

*MATLAB code used here is available at <https://github.com/RichardHerz/neural-networks>*

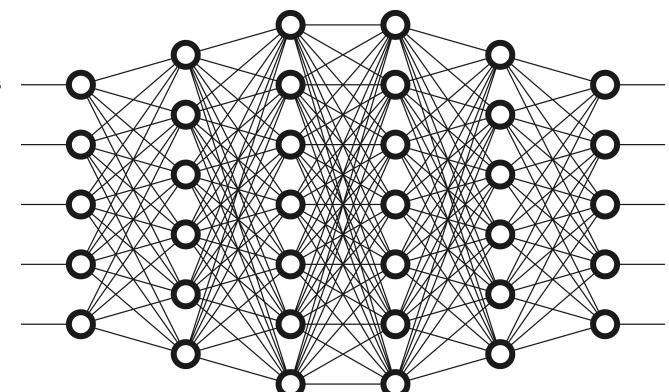
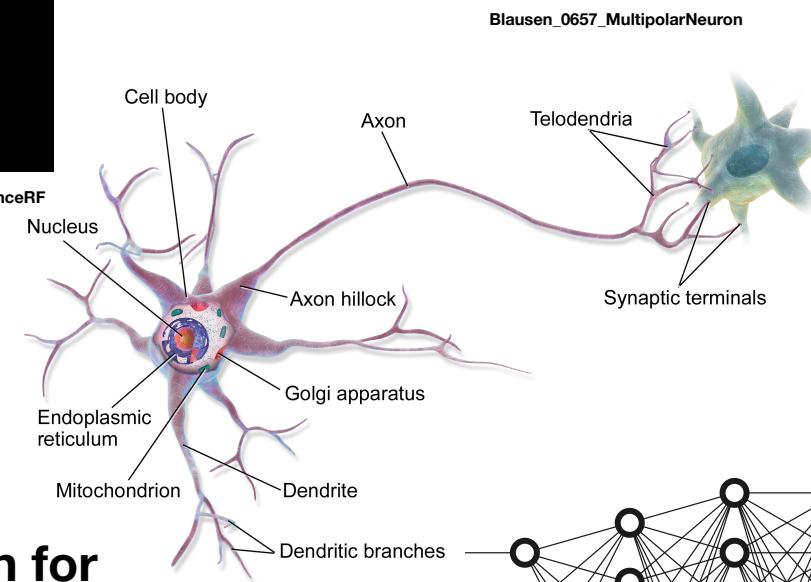


[www.braininjuryaustralia.org.au](http://www.braininjuryaustralia.org.au)



© ScienceRF

**Our brains sense and think using connected networks of cells called neurons**



neural\_network\_shutterstock\_all\_is\_magic.jpg

**These networks are the inspiration for “artificial neural networks” which can be trained to solve complex problems such as object recognition in images and speech recognition**

[github.com/RichardHerz](https://github.com/RichardHerz)

# Neural Network

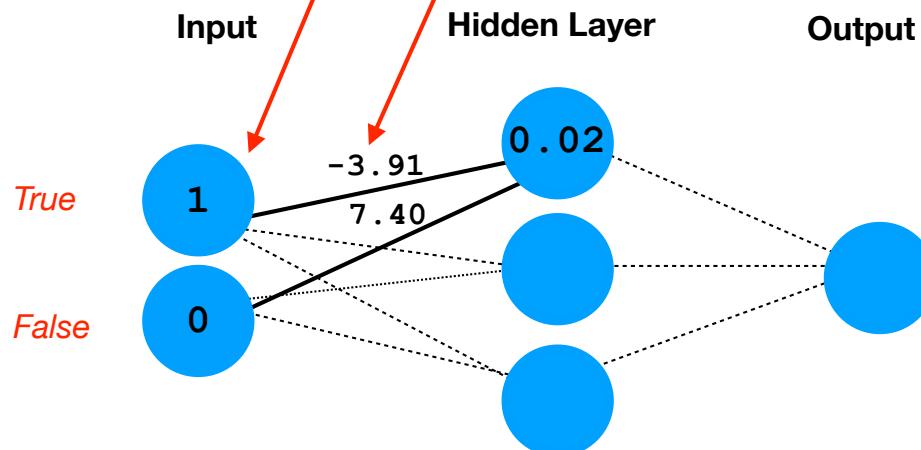
1

Simulates XOR logic - "exclusive or"

Output is TRUE when one input is TRUE but not both

**2 inputs, 1 output,  
1 hidden layer  
with 3 neurons &  
9 synapses**

We ignore biases here.  
An XOR network with  
biases only needs 2  
hidden nodes.



2

Each circle in the diagram represents a node or "neuron."  
Each line represents a connection or "synapse."  
The value in a neuron is its "activation."  
Each synapse has a connection "weight."

3 EXAMPLE for input of  
1  
0

node value = sigmaFunc( sum of ( connection weight \* node activation ) )

where  $\text{sigmaFunc}(x) = \exp(x) / (1 + \exp(x))$  >> converts all input x values into range 0 to 1

INPUT > HIDDEN LAYER

$\text{sigmaFunc}((-3.91 * 1) + (7.40 * 0)) = 0.02 = \text{hidden node 1 activation}$

$\exp(x)$  is exponential  
function of x

4

Every node - neuron - has a connection - synapse - to every neuron in nearest-neighbor layers of neurons  
in this basic type of neural network.

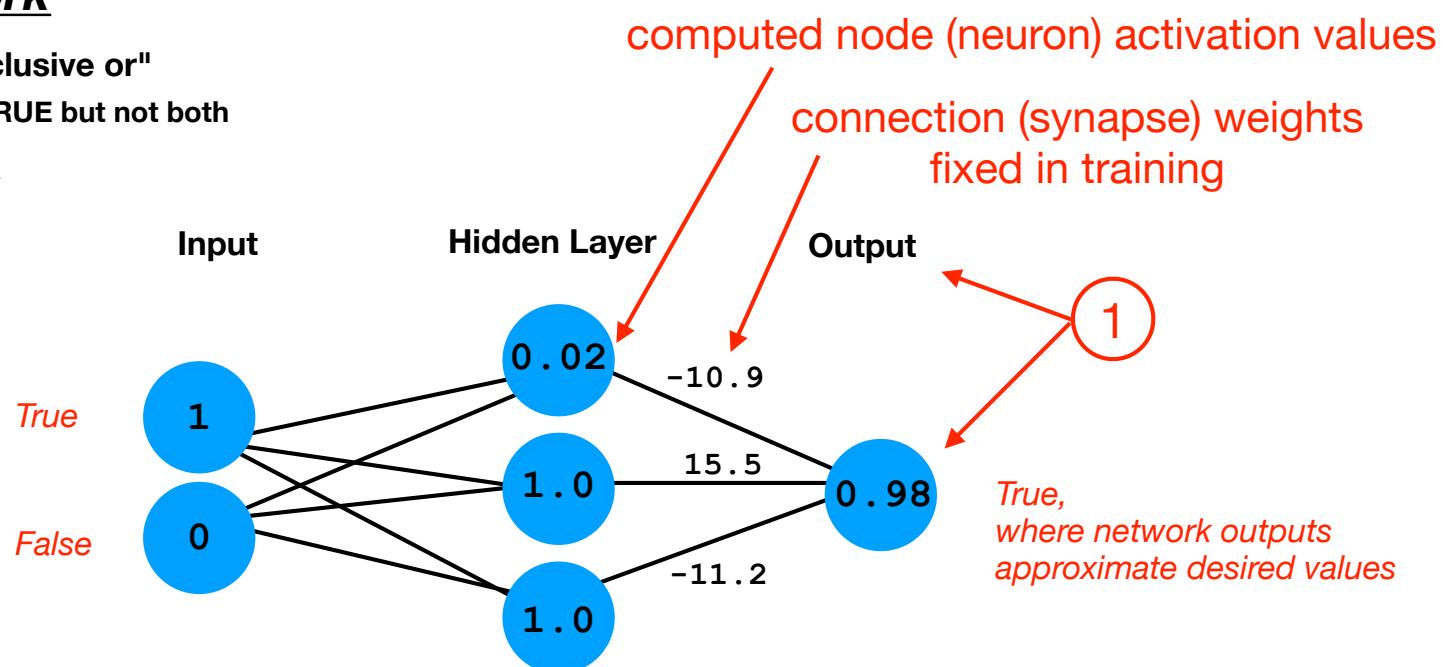
The values are held in memory locations and the CPU executes the math - there are no physical, hardware  
neurons and synapses.

# Neural Network

Simulates XOR logic - "exclusive or"

Output is TRUE when one input is TRUE but not both

**2 inputs, 1 output,  
1 hidden layer  
with 3 neurons &  
9 synapses**



## ② HIDDEN LAYER > OUTPUT

$$\text{sigmaFunc}((-10.9 * 0.020) + (15.5 * 1.0) + (-11.2 * 1.0)) = 0.98 = \text{output node}$$

The MATLAB code to solve for the output remains the same as that below, regardless of the size of the network:

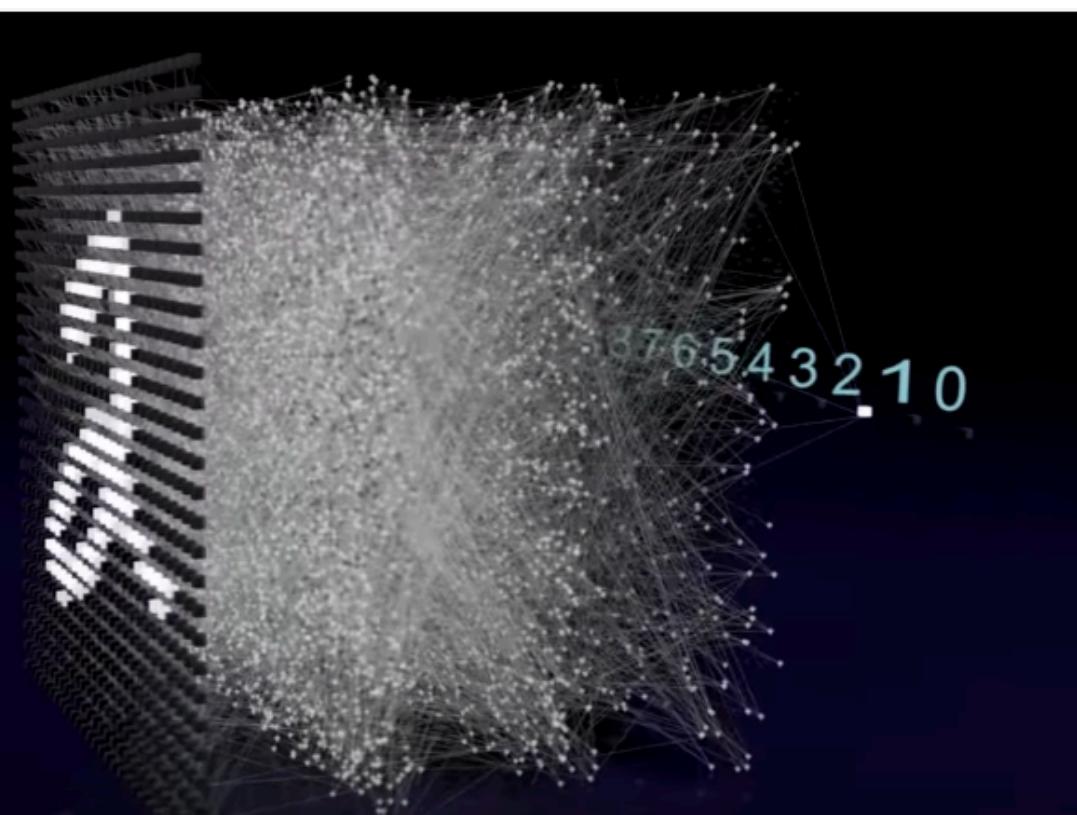
```
for i = 2 : numHiddenLayers + 2
    a{i} = sigmaFunc( W{i-1} * a{i-1} );
end
```

③ ***W*** is a MATLAB cell array whose elements are the matrices of synapse weights for each layer; ***a*** is a cell array whose elements are the vectors of neuron activation values. Each set of ***W*** and ***a*** are matrix-multiplied to obtain the neuron activation values for the next layer in the series of neuron layers.

④ Matrix multiplication is well suited to being accelerated in hardware Graphical Processing Units, since graphic transformations also involve matrix multiplication.

## *A neural network for a 28 x 28 “touch screen” that identifies handwritten numerals 0-9*

Type: ML Perceptron  
Data Set: MNIST  
Hidden Layers: 3  
Hidden Neurons: 10000  
Synapses: 24864180  
Synapses shown: 2%  
Learning: BP



Denis Dmitriev <https://youtu.be/3JQ3hYko51Y>

This network was trained with a set of known inputs and outputs that is much smaller than all the possible ways in which numbers can be drawn. The power of a trained network is to be able to give correct outputs for inputs for which the network wasn't trained. That's the main purpose of neural networks!

A neural network represents a large number of coupled equations which, when given a set of input values, can produce a set of desired output values.

The more neurons and synapses - the more equations - and the greater complexity of inputs and outputs which can be "fit" by the system of equations. Note the significant increase in complexity going from the XOR example (9 synapse weights) to the 2 x 2 "touch screen" (11,300) to the 28 x 28 touch screen in the figure above (24.9 million) to Open AI's GPT-3 (175 billion)! GPT-3 (175 billion)!



## A neural network might be thought of as a general function which can fit anything given enough terms...

In a sense, neural networks are math functions which can “fit” any desired input and output data given enough adjustable parameters, which are the “synapse” connection weights and, thus, enough neurons.

Using a neural network is somewhat similar to using a polynomial function to fit a series of data points (empirical fit) vs. using a functional form that represents the underlying physics (theoretical fit).

In a neural network, the functional form is fixed by the network structure. The values of the constants in the function are the connection weights, whose values are determined during training.

For the XOR network above, this is the Matlab code which computes the output  $a\{3\}$  given the input  $a\{1\}$

```
for i = 2:3
    a{i} = sigmaFunc( W{i-1}*a{i-1} );
end
```

Matrix  $W\{i-1\}$  and vector  $a\{i-1\}$  are elements of the cell arrays  $W$  and  $a$ . They are matrix multiplied. The Matlab code is very compact. We can see the form of this network’s function by looking at the expanded equation, which shows the individual terms. The output  $a\{3\}$  is a function of the inputs  $a\{1\}$ :

$$\begin{aligned} a^{\{3\}} &= f(a^{\{1\}}) = \sigma \left( W_1^{\{2\}} a_1^{\{2\}} + W_2^{\{2\}} a_2^{\{2\}} + W_3^{\{2\}} a_3^{\{2\}} \right) \\ &= \sigma \left( W_1^{\{2\}} \sigma \left( W_{1,1}^{\{1\}} a_1^{\{1\}} + W_{1,2}^{\{1\}} a_2^{\{1\}} \right) + W_2^{\{2\}} \sigma \left( W_{2,1}^{\{1\}} a_1^{\{1\}} + W_{2,2}^{\{1\}} a_2^{\{1\}} \right) + W_3^{\{2\}} \sigma \left( W_{3,1}^{\{1\}} a_1^{\{1\}} + W_{3,2}^{\{1\}} a_2^{\{1\}} \right) \right) \end{aligned}$$

where, for more compact notation, the superscript  $\{n\}$  of cell arrays  $a$  and  $W$  denotes a matrix in cell array element  $n$ , and the subscripts are the indices within that matrix. The hidden layer activations are  $a\{2\}$ . The nonlinear activation function for this network, which constrains activation values between 0 and 1, is

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

For a larger neural network of this type, there are more terms but the functional form remains unchanged. With the continued development of computers, larger networks can be computed more rapidly.

## ***Training Artificial Neural Networks***

The networks shown earlier in these notes had already been "trained" to produce correct outputs given various inputs. The connection weights are fixed in the initial training stage. This is the hard part of neural networks!

First, a collection of paired inputs and their correct outputs is obtained to use in training the network.

Initial weights are set to random values. Then an input is fed to the network and an output is obtained. This output is compared to the correct output and an error value is computed. The initial error value will be large.

Next, for each connection weight in the network, the rate of change of the error with respect to a change in the value of that weight is computed. After the rates of change for all the weights are obtained, the weights are changed by multiplying a small constant times each of the rates of change, and then subtracting the result from corresponding weights. This small adjustment to the weights, when the input is again fed to the network, will result in a smaller error in the output.

This procedure is called the "gradient descent method." The rates of change are also called gradients. You wish to "descend" to smaller errors.

This gradient descent procedure is repeated until the error reaches a minimum value. The concept is illustrated in the figure on the next slide.

## **More Complex Neural Networks**

These notes have discussed the simplest structure used in neural networks.

There are many more network structures to learn about, including

- The addition of bias values to each hidden and output layer, and use of other activation functions.
- Deep networks with many hidden layers used in “deep learning” - essentially very complex functions to relate complex inputs and outputs.
- Convolutional networks, in which various matrices are mathematically convoluted with input submatrices and scanned across an input matrix in order to help identify specific features, such as faces in an image or words in a sound spectrum.
- Recurrent networks with feedforward and feedback of information, useful in analyzing sequential inputs such as speech.
- Different types of networks arranged in parallel and series.

Search the website of the MathWorks, makers of MATLAB, for their neural network tools  
<https://mathworks.com>