Machine Learning

Artificial Neural Networks (ANN)

The idea of artificial neural networks (ANN) arose from thoughts and insights about the human mind.

Cultures throughout history have speculated about the nature of the brain, mind, heart, and soul. The interest in this article is not about the physiology of the brain but how the brain becomes a mind, through mental processes. It is in the realm of psychology, or the human psyche, a term that appeared in Greek philosophy as early sixth century BCE.

Scientific studies have revealed that our brain consists of billions of cells called neurons. They connect to form networks of abstract mental structures for acquiring and processing knowledge. In the words of Swiss psychologist Jean Piaget (d. 1980), "As experiences happen and new information is presented, new schemas are developed and old schemas are changed or modified". The human mind is conceived to consist of schemas, a cognitive framework for study of the human mind. A schema may be about a specific person in terms of the person's appearance or personality traits such as preferences and behavior. A schema may be a mental framework in terms of how people behave in certain situations; it may be about an event, or a view about oneself. Mental frameworks can facilitate our interactions with the world by accessing previously stored experiences. They can also be inhibiting by filtering new information according to our pre-existing beliefs and ideas.

From the models that depicted the human mind as consisting of neurons and connected networks, and sub-networks in the form of schemas of human mind experiences, scientists began to explore possibilities of building artificial neural networks. Humans receive input signals from the generally known senses of hearing, sight, touch, smell and taste. Output responses are produced through the senses such as speaking, writing reflecting speech and a variety of other forms.

From conceptual models of the human mind, scientists started efforts to build artificial neural networks (ANN) to model human behavior. American neurophysiologist Warren Sturgis McCulloch and mathematician Walter Pitts published a paper in 1943 modeling the working of neurons in the form of a simple network with electrical circuits. Reinforcing this concept of neurons and how they work was a book written by Donald Hebb. The *Organization of*

Behavior was written in 1949. It pointed out that neural pathways are strengthened each time that they are used.

Building neural networks in hardware did not prove to be very successful in showing how the human mind responds to inputs called stimuli. Perceptron built in hardware is among the widely known ANN as an outcome of research by Cornell University neurobiologist Frank Rosenblatt. It created euphoric feelings about the field of artificial intelligence. Perceptron proved to be very limited in terms of its potential. A period of disillusionment about artificial intelligence and ANN followed.

Opportunities for significant outcomes were opened up with the development of general-purpose computers in the early 1950s. In 1956 the Dartmouth Summer Research Project on Artificial Intelligence provided a boost to both artificial intelligence and neural networks. One of the outcomes was to stimulate research in both the intelligent side, AI, as it is known throughout the industry, and in the much lower level neural processing part of the brain.

As more and powerful computers were being built, the focus shifted to building specialized ANN by using algorithms that modeled human experiences, and implemented in the form of computer programs known as software. Today, the neural networks are finding practical applications in a vast variety of areas. These networks can learn dynamically through training with vast amounts of data. In ANNs, learning refers to the process of extracting structure—statistical regularities—from input data, and encoding that structure into the parameters of the network.

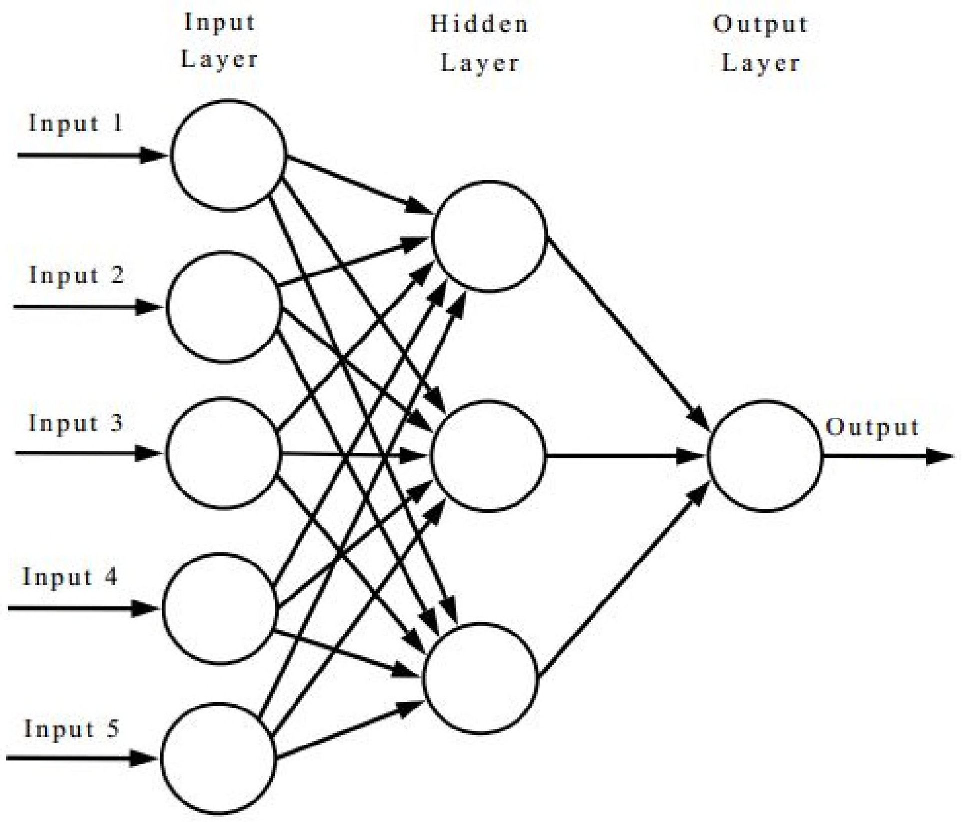
With increasing uses of ANN, there is also a growing interest in what is called algorithmic bias. An ANN models human activities and behavioral outcomes. Models are approximations of reality. How the modeler views the reality can introduce bias in ANN learning. This is called model bias. What data is used and how this data is organized in training an ANN can cause data bias. And other forms of biases may also exist with potentially negative consequences in people-related situations.

In the classification scheme for machine learning, ANN fall in the category of unsupervised learning.

We receive signals through our senses called inputs, these inputs are processed mentally through networks and subnetworks in the form of schemas from previous experiences, leading to output responses.

Consider an abstract model of an ANN shown in Figure 1.

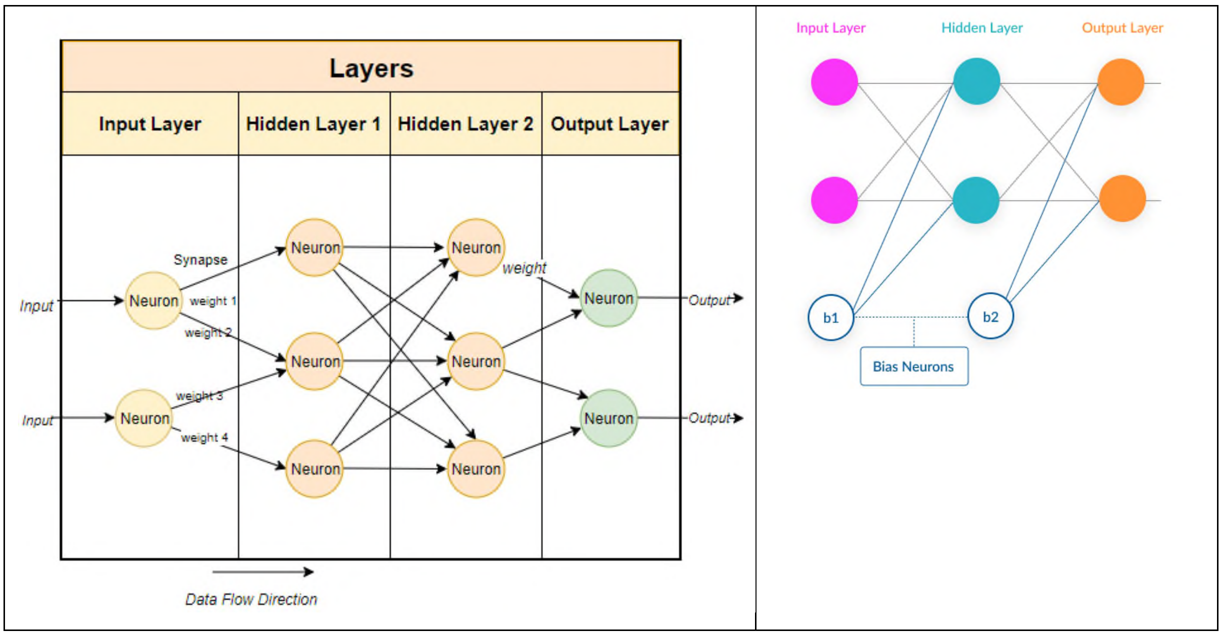
Figure 1: Graphical Model of a simple Artificial Neural Network



In this model five inputs are entering the network through an input layer. There is a single hidden layer that applies some algorithmic process to produce outcomes. The input layer transforms inputs into data for processing, and the output layer transforms the data entering into it as inputs into meaningful output responses. In practical situations, an ANN would have several hidden experience building layers.

Another more meaningful example of a neural networks, taken from https://medium.com/fintechexplained/neural-networks-a-solid-practical-guide-9f343594b02a and https://missinglink.ai/guides/neural-network-concepts/neural-network-bias-bias-neuron-overfitting-underfitting is shown in Figure 2. Additional relevant terms are synapse and weights. In brain physiology, a synapse, also called neuronal junction, is the site of transmission of electric nerve impulses between two nerve cells (neurons) or between a neuron and a gland or muscle cell (effector). A synaptic connection between a neuron and a muscle cell is called a neuromuscular junction. A weight is a parameter within a neural network that transforms input data within the network's hidden layers. A neural network is a series of nodes, or neurons. Within each node is a set of inputs, weight, and a bias value. As an input enters the node, it gets multiplied by a weight value and the resulting output is either observed, or passed to the next layer in the neural network. Often the weights of a neural network are contained within the hidden layers of the network. Bias serves two functions within the neural network – as a specific neuron type, called Bias may be a specific neuron type or a statistical measure used in training of an ANN model.

Figure 2: ANN Models Example with Some Additional Information



The bias neuron is a special neuron added to each layer in the neural network, which simply stores a value of 1. This makes it possible to move or "translate" the activation function left or right on the graph. Without a bias neuron, each neuron takes the input and multiplies it by a weight, because with nothing else added to the equation which could be problem, if the input is zero.

Although neural networks can work without bias neurons, in reality, they are almost always added, and their weights are estimated as part of the overall model.

Uses of ANN are growing in almost every area of human activity in business, government, manufacturing and service sectors. Examples of some general areas of ANN uses are:

- Quality assurance, performance monitoring and control of systems.
- Surveillance and response
- Aircraft component fault detection and simulations
- Electronic chips layout, failure vision and analysis, modeling
- Chemical process modeling and control
- Robotics
- Talent search, hiring efficiency, employee retention, work satisfaction.
- Matching children with foster care givers
- Diagnostics and predictive analysis of medical data for gaining insights in medical care.
- Skin tracking in dermatology
- Bank credit card attrition, loans application evaluation, delinquencies, risks and frauds.
- Counter terrorism, facial recognition and extraction of features, target tracking.
- Adaptive learning in education, performance modeling, personality profiling.
- Stock trading advisory systems
- Transportation logistics.

The following table provides examples of some specific applications.

Table1: Some Specific Examples of ANN Application and ANN Processes

Application	**ANN Process**
Classification of data	Predict presence of one or more objects based on a set of data,
Anomaly detection	Based on transactions by an entity, find if a given transaction is fraudulent.
Speech recognition	Based on records of speaking from several sources, recognize who is the speaker in a given case.
Audio generation	Given the inputs as audio files, it can generate new music based on various factors like genre, singer, and others.
Time series analysis	An ANN can be trained using stock market data and then used to predict the stock prices.
Character recognition	An ANN can be trained using handwriting data to detect handwritten characters
Machine translation	Using dictionary and vocabulary data, an ANN can be trained to translate one language into another language.

Image processing	After training with data on images, an ANN recognize and extract features and match images.

In any application of an ANN, there are two important considerations. We must have large volumes of data as prior experiences and this data must be representative of the population for a given application.

In the example that follows, we will start with a simpler form of ANN. It falls in the category of a supervised learning systems. It consists of a number of simple elements, called neurons or perceptron. Each neuron can make simple decisions, and feeds those decisions to other neurons, organized in interconnected layers. Together, the neural network can emulate almost any function, and answer practically any question, given enough training samples and computing power.

Before working on the details of the example for implantation in Python, there are some other ANN guiding concepts that require basic understanding, such as the following.

A simpler form of ANN is known as a “shallow” neural network (SNN). It has only three layers of neurons:

- An **input layer** that accepts the independent variables or inputs of the model with the goal of making a decision or prediction about the data. Inputs to a neural network are typically a set of real values; each value is fed into one of the neurons in the input layer.
- One **hidden layer**
- An **output layer** that generates responses called predictions. Neural networks generate their predictions in the form of a set of real values or Boolean decisions. Each output value is generated by one of the neurons in the output layer.

Each neuron accepts part of the input and passes it through the **activation function**. One of the commonly known activation functions is a sigmoid. An activation functions helps generate output values within an acceptable range, and their non-linear form is crucial in achieving that goal. Each neuron is assigned a numeric **weight**. The weights, together with the activation function, define each neuron’s output. Neural networks are trained by fine-tuning weights, to discover the optimal set of weights that generates the most accurate prediction.

A **Forward pass** through the network takes the inputs, passes them through the network and allows each neuron to react to a fraction of the input. Neurons generate their outputs and pass them on to the next layer, until eventually the network generates an output.

An **Error Function** defines how far the actual output of the current model is from the correct output. When training the model, the objective is to minimize the error function and bring output as close as possible to the correct value.

A **Backward Pass** is performed in order to discover the optimal weights for the neurons, moving back from the network's prediction to the neurons that generated that prediction. This is called backpropagation. Backpropagation tracks the derivatives of the activation functions in each successive neuron, to find weights that brings the loss function to a minimum, which will generate the best prediction. This is a mathematical process called gradient descent.

When training neural networks, like in other machine learning techniques, we try to balance between bias and variance.

Bias measures how well the model fits the training set—able to correctly predict the known outputs of the training examples.

Bias neuron has a value of 1. However, the weight assigned to it is random at first, and model optimizes it for our target output. Generally, we keep the learning rate as low as possible so that we can achieve a minimum error rate. Bias neuron allows moving the activation function to left, right, up or down. It prevents input of zero when multiplied by a weight to produce 0 as outcome. Activation function movements are required to generating the required output response from the network.

Variance measures how well the model works with unknown inputs that were not available during training. Another meaning of bias is a "bias neuron" which is used in every layer of the neural network. The bias neuron holds the number 1, and makes it possible to move the activation function up, down, left and right on the number graph.

A **perceptron** is a binary classification algorithm modeled after the functioning of the human brain—it was intended to emulate the neuron. The perceptron, while it has a simple structure, has the ability to learn and solve very complex problems.

Gradient Descent is a machine learning algorithm that operates iteratively to find the optimal values for its parameters. It takes into account, user-defined learning rate, and initial parameter values.

One cycle through the training data set is called an **Epoch.** Usually, training a neural network takes more than a few epochs. In other words, if we feed a neural network the training data for more than one epoch in different patterns, we hope for a better generalization when given a new "unseen" input (test data). The term epoch is often mixed up with an iteration. Iterations is the number of batches or steps through partitioned packets of the training data, needed to complete one epoch. Heuristically, one motivation is that (especially for large but finite training sets) it gives the network a chance to see the previous data to readjust the model parameters so that the model is not biased towards the last few data points during training.

A **training set** is generally a large volume of example data, experience, used to train an ANN.

A **Deep Neural Network (DNN)** has a similar structure, but it has two or more "hidden layers" of neurons that process inputs. While shallow neural networks are able to tackle complex problems, deep learning networks are more accurate, and improve in accuracy as more neuron layers are added. Additional layers are useful up to a limit of 9-10, after which their predictive power starts to decline. Today most neural network models and implementations use a deep network of between 3-10 neuron layers.

Example 1:

Let us now consider a very simple example but with practical flavor, from an article dated June 29, 2020 authored by Pratik Shukla, Roberto Iriondo**:**
https://medium.com/towards-artificial-intelligence/building-neural-networks-from-scratch-with-python-code-and-math-in-detail-I-536fae5d7bbf

The situation is that of making a diagnosis from a sample of symptoms shown in Table 2.

Table 2: Commonly Mentioned Symptoms and Diagnosis for Covid-19 Cases

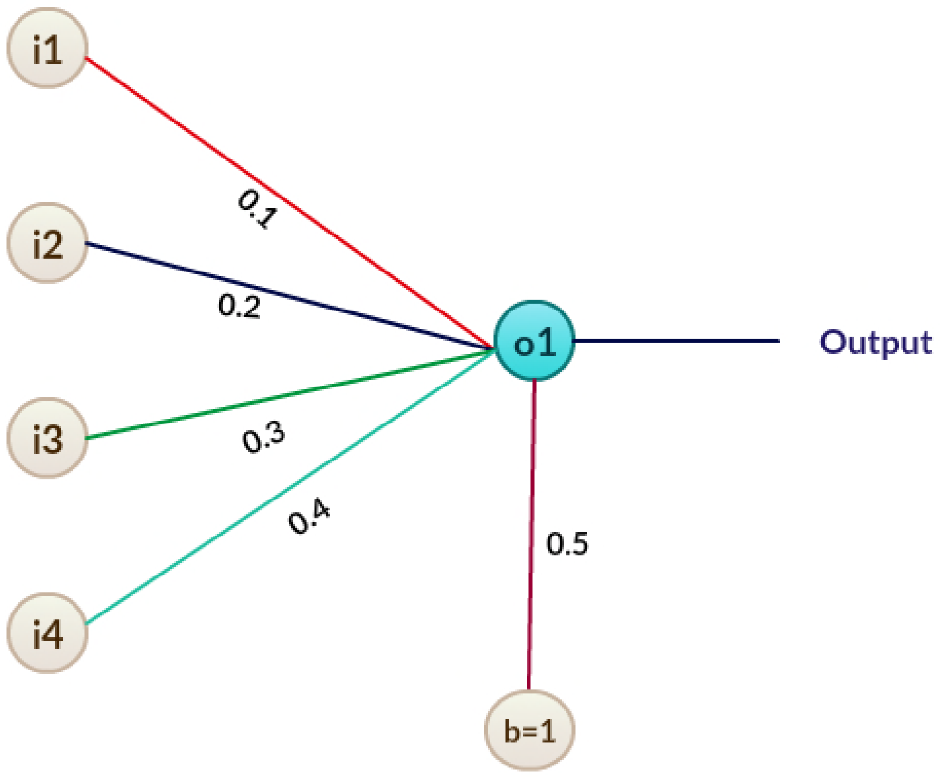
Person	Smell Loss (i1)	Weight Loss (i2)	Runny Nose (i3)	Body Pain (i4)	Diagnosis, y (1: positive)
1	1	0	0	1	1
2	1	0	0	0	1
3	0	0	1	1	0
4	0	1	0	0	0
5	1	1	0	0	1
6	0	0	1	1	1
7	0	0	0	1	0
8	0	0	1	0	0

This simple example is intended to build a toy ANN and play with it in order to gain practical insights for building meaningful applications of ANN. Data features are expressed simply as 0 for an absence of a symptom and 1 for presence of a symptom.

The form in which the data is shown here is called a truth table. There are four cases where the diagnosis is positive.

This situation can be modeled as a shallow ANN without a hidden layer as shown in Figure 3.

Figure 3: Model of a Shallow Neural Network (Perceptron)



The initial weights assigned to inputs are arbitrary and get adjusted during the training process. In the simple situation at hand, we can find some clues for assignment of weights from simply a visual inspection. For example, Loss of Smell (i1) is present in 75% of the positive diagnosis. The next influencer is Body Pain (i4) causing positive diagnosis in 50% of the cases, while Weight Loss (i2) and Runny nose (i3) appear to cause positive diagnosis in only 25% of the cases. You will see this evidence in the final optimized values of weights printed after the training process.

Similar considerations may relevant to assigning the values to bias neuron and its weight. The output values are what is expected from the network.

The implementation of this model in Python language is described in steps, with the coding appearing in boxes, and explanations before and after a box.

Step 1: Importing Packages as Need

```
import numpy as np
```

Numpy is a package containing pre-built methods for applications using numerical computations.

Step 2: Load and Display Input Data

```
input_features = np.array([[1,0,0,1],[1,0,0,0],[0,0,1,1],
 [0,1,0,0],[1,1,0,0],[0,0,1,1],
 [0,0,0,1],[0,0,1,0]])
print (input_features.shape)
print (input_features)
```

```
(8, 4)
[[1 0 0 1]
 [1 0 0 0]
 [0 0 1 1]
 [0 1 0 0]
 [1 1 0 0]
 [0 0 1 1]
 [0 0 0 1]
 [0 0 1 0]]
```

Values of input feature are created as binary numbers shown in the truth table, resulting output. The shape of input data is 8 rows data points and 4 columns of values for the four features.

Step 3. Load and Display Expected Output Responses

```
target_output = np.array([[1,1,0,0,1,1,0,0]])
target_output = target_output.reshape(8,1)
print(target_output.shape)
```

```
print (target_output)
```

```
(8, 1)
[[1]
 [1]
 [0]
 [0]
 [1]
 [1]
 [0]
 [0]
```

Data is loaded and shaped into a column vector values to columns of input data.

Step 4: Assign and Display weights for Input Data

```
weights = np.array([[0.1],[0.2],[0.3],[0.4]])
print(weights.shape)
print (weights)# Bias weight :
```

```
(4, 1)
[[0.1]
 [0.2]
 [0.3]
 [0.4]]
```

Step 5: Applying Activation Function for constraining the values between 0 and 1.

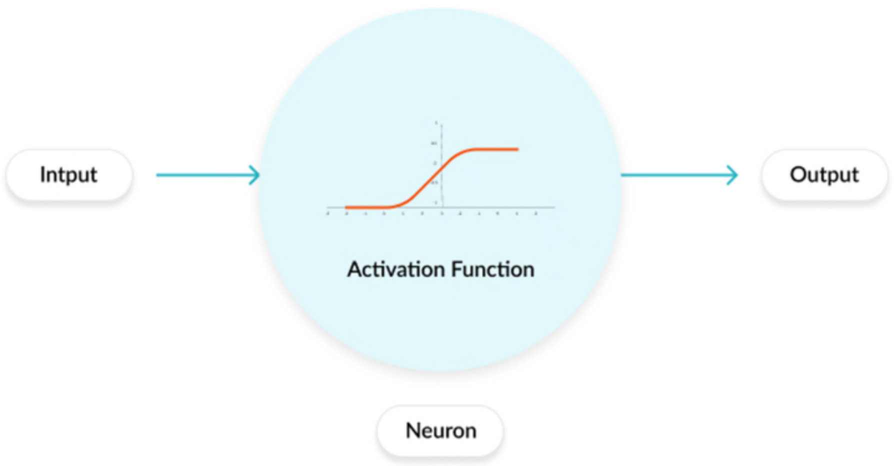
Each input has an associated **weight** (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function ***f*** (defined below) to the weighted sum of its inputs as shown in Figure 3:

The resulting output can be described as:

Output, y= f(b + i1*w1 + i2*w2+ i3*w3 + i4*w4) = f(b+ $\Sigma i_i * w_i$) where b is a bias with an assigned weight.

The function is non-linear and is called an activation function. Commonly used example, particularly for this situation is a Sigmoid, with a visual of its operation in Figure 4.

Figure 4: Operational View of Sigmoid Activation function



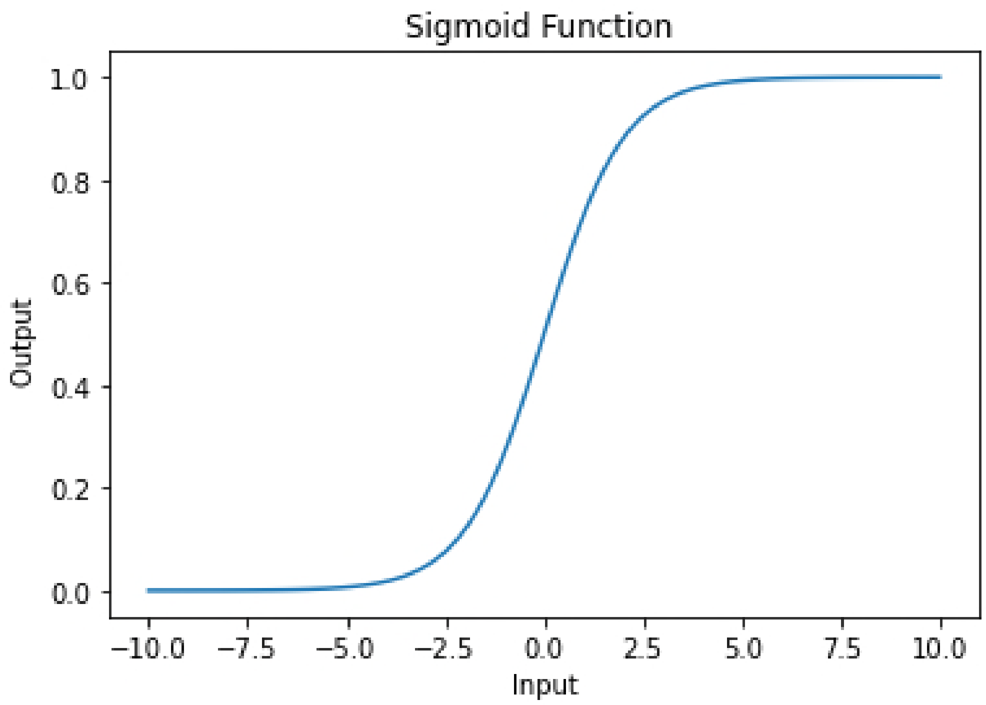
The mathematical representation of the Sigmoid function is:

f(x)= 1/(1+exp(-x)) where e is a mathematical constant, known as Euler's number, 2.718281828459045…, approximated as 2.71828.

Note that for x=0, f(0)= 1/(1+np.exp(-o)) leading to 1/(1+1)=.5. At x=-10, f(x)=.00000454 and at x=10, f(x)=.99999955.

The coding for Sigmoid function and its graphical display is shown below.

```
import numpy as np
import matplotlib.pyplot as plt
input=np.linspace(-10,10,100)
def sigmoid(x):
  return 1/(1+np.exp(-x))
output=sigmoid(input)
plt.plot(input,output)
plt.xlabel("Input")
plt.ylabel("Output ")
plt.title("Sigmoid Function ")
```



Step 6: Bias and Learning Assignments

```
bias = 0.3
lr = 0.05
```

Bias is a parameter in the neural network which is used to adjust the output along with the weighted sum of the inputs to the neuron. Therefore, bias is a constant which helps the model in a way that it can fit best for the given data.

The learning rate is a configurable hyperparameter used in the training of neural networks. It has a small positive value, often in the range between 0.1 and 1.0. The learning rate controls how quickly the model is adapted to the problem at hand.

<u>Step 6: Defining Sigmoid Function and its Derivative (Rate of Change)</u>

```
def sigmoid(x):
 return 1/(1+np.exp(-x))
# Derivative of sigmoid function :
def sigmoid_der(x):
 return sigmoid(x)*(1-sigmoid(x))
```

<u>Step 7: Executing Learning Epochs</u>

```
for epoch in range(10000):
    inputs = input_features
    #Feedforward input :
    pred_in = np.dot(inputs, weights) + bias
    #Feedforward output :
    pred_out = sigmoid(pred_in)
    #Backpropogation
    #Calculating error
    error = pred_out - target_output
    #Going with the formula :
    x = error.sum()
    print(x)
    #Calculating derivative :
    dcost_dpred = error
    dpred_dz = sigmoid_der(pred_out)
    #Multiplying individual derivatives :
    z_delta = dcost_dpred * dpred_dz
    #Multiplying with the 3rd individual derivative :
    inputs = input_features.T
    weights -= lr * np.dot(inputs, z_delta)
    #Updating the bias weight value :
    for i in z_delta:
        bias -= lr * i

print("Error Sum Squares", x)
print("Number of Epochs Completed = " epoch)
```

Allowed to take as many as 10,000 learning runs.

Error Sum Squares 0.0032273154199536913

Number of Epochs Completed = 9999

Appears to be a good fit.

Step 8: Display Final Values of Weights and Bias

```
print(weights)
print("\n\n")
print(bias)
```

```
[[12.23844365]
 [ 0.98735984]
 [ 3.78293481]
 [ 3.79338689]]
```

```
[-7.62902211]
```

Make a not of these weights and bias after ANN has gone through training. They have all been adjusted during the training.

Step 8: Find Predicted Response (Diagnosis) Based on Test Inputs (Symptoms)

First Test:

```
# Select Input Values
single_point = np.array([1,0,0,1])
result1 = np.dot(single_point, weights) + bias
result2 = sigmoid(result1)
print(result2)
print (weights)
print("\n\n")
print(bias)
```

```
[0.99915866]

[[12.23844365]
 [ 0.98735984]
 [ 3.78293481]
 [ 3.79338689]]


[-7.62902211]
```

The diagnosis for symptoms 1,0,0,1 is .999 which is 1 for all practical purposes. This value is the same as in the given data shown in Table 2, first row of values. It says that the diagnosis is positive if Loss of Smell and Body Pain are both present.

```
print (weights)
print("\n\n")
print(bias)
```

```
[[10.1634992 ]
 [ 0.45744949]
 [ 2.97441782]
 [ 2.9938138 ]]
```

Second Test:

```
#Taking inputs :
```

```
single_point = np.array([0,0,1,0])
#1st step :
result1 = np.dot(single_point, weights) + bias
#2nd step :
result2 = sigmoid(result1)
#Print final result
print(result2)
print (weights)
print("\n\n")
print(bias)
```

```
[0.02091632]


[[12.23844365]
 [ 0.98735984]
 [ 3.78293481]
 [ 3.79338689]]
```

The diagnosis for symptoms 0,0,1,0 is 0.0209 which is 0 for all practical purposes. This value is the same as in the given data shown in Table 2, last row of values. It says that the diagnosis is negative if there is only Runny Nose.

Third Test:

```
#Taking inputs :
single_point = np.array([1,0,1,0])#1st step :
result1 = np.dot(single_point, weights) + bias#2nd step :
result2 = sigmoid(result1)#Print final result
print(result2)
```

```
[0.99977346]


[[12.23844365]
 [ 0.98735984]
 [ 3.78293481]
 [ 3.79338689]]
```

```
[-7.62902211]
```

The diagnosis for symptoms 1,0,1,0 is .99977 which is 1 for all practical purposes. It says that the diagnosis is positive if Loss of Smell and Runny Nose are both present. This row of symptoms is not in the data shown in Table 2. With four binary variables i1, i2, i3, and i4, there are sixteen possible observations or rows. The provided data covers the sample space only partially.

Complete Listing of Python Coding:

```
# Import required libraries :
import numpy as np
#
input_features = np.array([[1,0,0,1],[1,0,0,0],[0,0,1,1],
 [0,1,0,0],[1,1,0,0],[0,0,1,1],
 [0,0,0,1],[0,0,1,0]])
print (input_features.shape)
print (input_features)
#
target_output = np.array([[1,1,0,0,1,1,0,0]])
target_output = target_output.reshape(8,1)
print(target_output.shape)
print (target_output)
weights = np.array([[0.1],[0.2],[0.3],[0.4]])
print(weights.shape)
print (weights)
#
bias = 0.3
lr = 0.05
#
def sigmoid(x):
 return 1/(1+np.exp(-x))
#
def sigmoid_der(x):
 return sigmoid(x)*(1-sigmoid(x))
print("Before Starting Epocs")
#
# Running our code 10000 times
for epoch in range(10000):
    inputs = input_features
    #Feedforward input :
    pred_in = np.dot(inputs, weights) + bias
```

```
     #Feedforward output :
     pred_out = sigmoid(pred_in)
     #Backpropogation
     #Calculating error
     error = pred_out - target_output
     #Going with the formula :
     x = error.sum()
#    print(x)
     #Calculating derivative :
     dcost_dpred = error
     dpred_dz = sigmoid_der(pred_out)
     #Multiplying individual derivatives :
     z_delta = dcost_dpred * dpred_dz
     #Multiplying with the 3rd individual derivative :
     inputs = input_features.T
     weights -= lr * np.dot(inputs, z_delta)
     #Updating the bias weight value :
     for i in z_delta:
         bias -= lr * i
#
print("Error Sum Squares = ", x)
#Printing final weights:
print (weights)
print("\n\n")
print(bias)
# Find predicted response
#Select inputs values for first test
single_point = np.array([1,0,0,1])
result1 = np.dot(single_point, weights) + bias#2nd step :
result2 = sigmoid(result1)#Print final result
print(result2)
print (weights)
print("\n\n")
print(bias)
#Select input values for second test
single_point = np.array([0,0,1,0])
result1 = np.dot(single_point, weights) + bias
result2 = sigmoid(result1)
print(result2)
print (weights)
print("\n\n")
print(bias)
#Select input values for third test
single_point = np.array([1,0,1,0])#1st step :
result1 = np.dot(single_point, weights) + bias#2nd step :
result2 = sigmoid(result1)#Print final result
```

```
print(result2)
print (weights)
print("\n\n")
print(bias)
```

More Sources:

https://www.smartsheet.com/neural-network-applications

http://www2.psych.utoronto.ca/users/reingold/courses/ai/cache/neural4.html

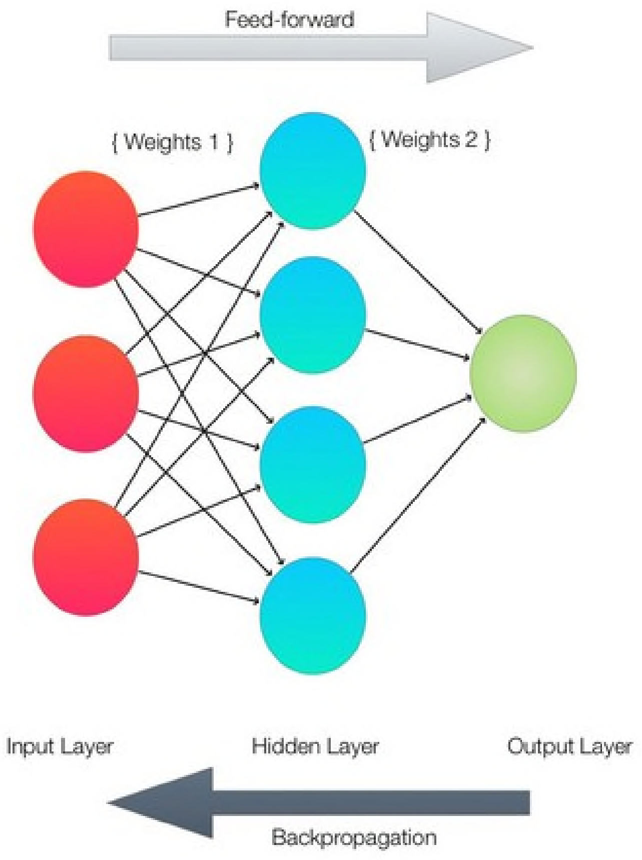
Biological Neuron Model, Wikipedia, https://en.wikipedia.org/wiki/Biological_neuron_model

Logic Gate, Wikipedia, https://en.wikipedia.org/wiki/Logic_gate

Example 2

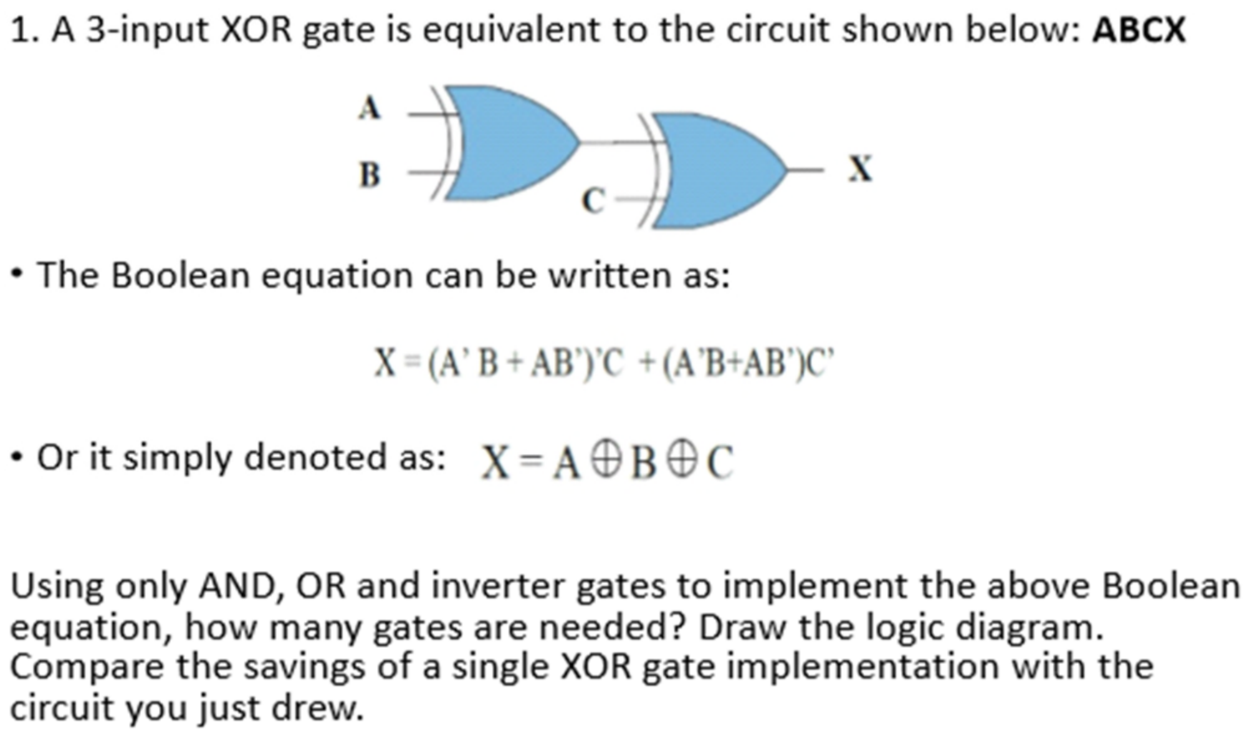
This example goes further by including a hidden layer in the network model as shown in Figure 5

Figure 5: Graphical Model of an Artificial Neural Network with a Hidden Layer of Neurons



In this model, there are three input layer neurons x1, x2, x3, four hidden, four hidden layer neurons h1, h2, h3, and h2, and finally an output layer neuron, y. The data chosen for implementation of this model is an abstract situation, called a 3-input Exclusive OR (XOR) gate as shown in Figure 6.

Figure 6: a 3-input XOR with showing two 2-input XORs to form a 3-input XOR



1. A 3-input XOR gate is equivalent to the circuit shown below: **ABCX**

- The Boolean equation can be written as:

$$X = (A'B + AB')'C + (A'B+AB')C'$$

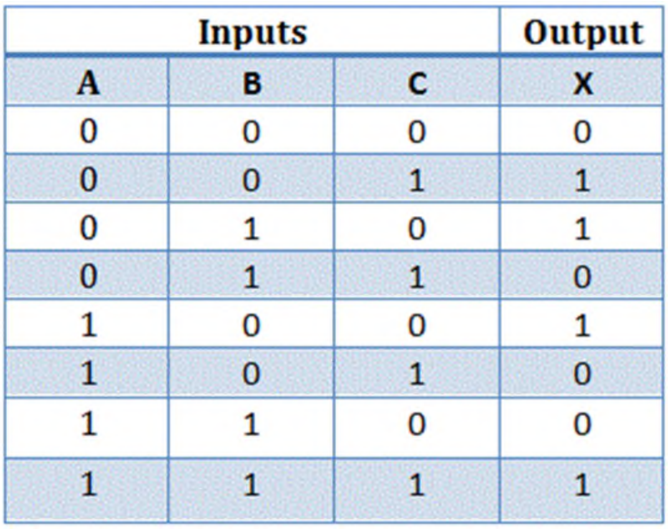
- Or it simply denoted as: $X = A \oplus B \oplus C$

Using only AND, OR and inverter gates to implement the above Boolean equation, how many gates are needed? Draw the logic diagram. Compare the savings of a single XOR gate implementation with the circuit you just drew.

Some practical applications of XOR gates are pseudo-random number generation, correlation of bit patterns, and bit sequence detection.

The truth table for a 3-input XOR is shown in Figure 7.

Figure 7: XOR Truth Table with inputs A, B, C, and output shown as X, corresponding to Figure 7



Inputs			Output
A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

In the following Python coding details, the inputs are named x1, x2, and x3. The 8-rows, three columns matrix is denoted as X= [x1, x2, x3]. The output is denoted as y, an 8-rows, 1 column array. This example is described by John Shovic and Alan Simpson in their Book, Python All-In-One, published by John Wiley and Sons, NJ, 2019. Some data values and coding has been modified for consistency.

Step1: Load Numerical Methods numpy package nick named as np

```
import numpy as np
```

Step 2: Set up Input Data and Display for Verification.

```
# X = input of our 3 input XOR gate
# set up the inputs of the neural network
X = np.array(([0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]), dtype=float)
#y = Expected output of neural network
y= np.array(([1],[1],[0],[0],[0],[0],[0],[1]), dtype=float)
```

```
[[0. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 1.]
 [1. 0. 0.]
 [1. 0. 1.]
 [1. 1. 0.]
 [1. 1. 1.]]
[[0.]
 [1.]
```

 [1.]
 [0.]
 [1.]
 [0.]
The input matrix X of eight rows and 4 columns. The expected array has 8 rows and one column.

Step 3: Set Targets and Weights

```
#what value we want to predict
xPredicted = np.array(([0,0,1]), dtype=float)
print("xPredicted", xPredicted)
X = X/np.amax(X, axis=0) # maximum of X input array
#maximum of xPredicted (our input data for the prediction).
xPredicted = xPredicted/np.amax(xPredicted, axis=0)
print("xPredicted Maximum", xPredicted)
xPredicted [0. 0. 1.]
xPredicted Maximum [0. 0. 1.]
```

```
xPredicted [0. 0. 1.]
```

Step 4: Set Up a csv, called lossFile, for storing Sum Square Errors

```
lossFile = open("SumSguaredLossList.csv", "w")
```

Step 5: Create a Class Called Neural_Network and Methods to be associated with this class.

```
class Neural_Network(object):
  def __init__(self):
    self.inputLayerSize= 3 # X1,X2,X3
    self.outputLayerSize= 1 # Y1
    self.hiddenLayerSize= 4 # Size of the hidden layer
    self.W1 = np.random.randn(self.inputLayerSize,self.hiddenLayerSize)
    self.W2 = np.random.randn(self.hiddenLayerSize,self.outputLayerSize)
#
  def activationSigmoid(self, s):
    return 1/(1+np.exp(-s))
#
  def activationSigmoidDer(self, s):
    return s * (1-s)
#
  def feedForward(self, X):
    self.z = np.dot(X, self.W1)
    self.z2 = self.activationSigmoid(self.z)
    self.z3 = np.dot(self.z2, self.W2)
    o = self.activationSigmoid(self.z3)
    return o
```

```
#
  def backwardPropagate(self, X, y, o):
    self.o_error = y - o
    self.o_delta = self.o_error*self.activationSigmoidDer(o)
    self.z2_error = self.o_delta.dot(self.W2.T)
    self.z2_delta = self.z2_error*self.activationSigmoidDer(self.z2)
    self.W1 += X.T.dot(self.z2_delta)
    self.W2 += self.z2.T.dot(self.o_delta)

  def trainNetwork(self, X, y):
    o= self.feedForward(X)
    self.backwardPropagate(X, y, o)
#
  def saveSumSquaredLossList(self,i,error):
    lossFile.write(str(i)+","+str(error.tolist())+'\n')

  def saveWeights(self):
    np.savetxt("weightsLayerl.txt", self.W1, fmt="%s")
    np.savetxt("weightsLayer2.txt", self.W2, fmt="%s")

  def predictOutput(self):
    print ("Predicted XOR output data based on trained weights: ")
    print ("Expected (X1-X3): \n" + str(xPredicted))
    print ("Output (Y1): \n" + str(self.feedForward(xPredicted)))
```

Creating a class and defining its methods gives us a number of advantages. When instances are created, they are automatically encapsulated with its associated methods. Fragments of code that are likely to be used many times are defined once as a function with a name. These fragments can be executed by simply a call to the function. If the details of internal operations in a function get modified, the calls can be made as before without being concerned about those details as long as the function objective remains the same.

Step 6: Create an instance of the Neural_Network class. It is now a working object

```
myNeuralNetwork = Neural_Network()
myNeuralNetwork.inputLayerSize
myNeuralNetwork.ouputLayerSize
myNeuralNetwork.hiddenLayerSize
myNeuralNetwork.W1
myNeuralNetwork.W2
```

```
myNeuralNetwork.inputLayerSize
3
myNeuralNetwork.outputLayerSize
1
```

```
myNeuralNetwork.hiddenLayerSize
4
myNeuralNetwork.W1

array([[ 1.03205222, -1.01923292, -0.94697756,  0.16023102],

       [-0.29517126, -0.37420777, -2.10696714, -1.29781689],

       [ 2.00248949,  0.01310485, -0.62581832, -0.67745429]])

myNeuralNetwork.W2

array([[ 1.08758167],

       [-1.17630573],

       [-0.52214085],
```

Step 7: Network Training Epochs

```
myNeuralNetwork = Neural_Network()
trainingEpochs = 10000
for i in range(trainingEpochs):
  print ("Epoch * " + str(i) + "\n")
  print ("Network Input : \n" + str(X))
  print ("Expected Output of XOR Gate Neural Network: \n" + str(y))
  print ("XOR Gate Actual Output: \n"+ str(myNeuralNetwork.feedForward(X)))
  Loss = np.mean(np.square(y - myNeuralNetwork.feedForward(X)))
  myNeuralNetwork.saveSumSquaredLossList(i,Loss)
  print ("Sum Squared Loss: \n" + str(Loss))
  print ("\n")
  myNeuralNetwork.trainNetwork(X, y)
```

Final results after the epochs:

```
Epoch * 9999

Network Input :
[[0. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 1.]
 [1. 0. 0.]
 [1. 0. 1.]
 [1. 1. 0.]
 [1. 1. 1.]]
Expected Output of XOR Gate Neural Network:
[[0.]
```

```
 [1.]
 [1.]
 [0.]
 [1.]
 [0.]
 [0.]
 [1.]]
XOR Gate Actual Output:
[[0.02364826]
 [0.98423801]
 [0.98484896]
 [0.00801493]
 [0.98487698]
 [0.00922088]
 [0.01546563]
 [0.99845478]]
Sum Squared Loss:
0.00020709719265751375
```

The model performance appears to be very good, if we look at the expected output versus the actual output. Let us look at the side by side as shown in Table 3.

Table 3: Expected versus Actual Output of the Neural Network Model

Expected Output	**Actual Output**	**Rounded Values of Output**
0	0.02364826	0
1	0.98423801	1
1	0.98484896	1
0	0.00801493	0
1	0.98487698	1
0	0.00922088	0
0	0.01546563	0
1	0.99845478	1

Sum Squared Loss value provides further confirmation of this out. The expected versus actual output values are very closes to each other. There is hardly any difference.

Step 8: Post Epoch Results

```
myNeuralNetwork.saveWeights()
myNeuralNetwork.predictOutput()
```

```
Predicted XOR output data based on trained weights:
Expected (X1-X3):
[0. 0. 1.]
Output (Y1):
[0.98423891]
```

The complete listing of the Python code for this example:

```
import numpy as np

# X = input of our 3 input XOR gate
# set up the inputs of the neural network
X = np.array(([0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]), dtype=float)
print(X)
#y = output of neural network
y= np.array(([0],[1],[1],[0],[1],[0],[0],[1]), dtype=float)
print(y)
#
#what value we want to predict
xPredicted = np.array(([0,0,1]), dtype=float)
print("xPredicted", xPredicted)
X = X/np.amax(X, axis=0) # maximum of X input array
#maximum of xPredicted (our input data for the prediction).
xPredicted = xPredicted/np.amax(xPredicted, axis=0)
#
#set up our Loss file for graphing
lossFile = open("SumSguaredLossList.csv", "w")

class Neural_Network(object):
   def __init__(self):
       self.inputLayerSize= 3 # X1,X2,X3
#
       self.outputLayerSize= 1
#
       self.hiddenLayerSize= 4 # Size of the hidden layer
       self.W1 = np.random.randn(self.inputLayerSize,self.hiddenLayerSize)
       self.W2 = np.random.randn(self.hiddenLayerSize,self.outputLayerSize)
#
   def activationSigmoid(self, s):
       return 1/(1+np.exp(-s))
#
   def activationSigmoidDer(self, s):
       return s * (1-s)
#
   def feedForward(self, X):
      self.z = np.dot(X, self.W1)
      self.z2 = self.activationSigmoid(self.z)
      self.z3 = np.dot(self.z2, self.W2)
      o = self.activationSigmoid(self.z3)
      return o
#
```

```
   def backwardPropagate(self, X, y, o):
       self.o_error = y - o
       self.o_delta = self.o_error*self.activationSigmoidDer(o)
       self.z2_error = self.o_delta.dot(self.W2.T)
       self.z2_delta = self.z2_error*self.activationSigmoidDer(self.z2)
       self.W1 += X.T.dot(self.z2_delta)
       self.W2 += self.z2.T.dot(self.o_delta)

   def trainNetwork(self, X, y):
      o= self.feedForward(X)
      self.backwardPropagate(X, y, o)
#
  def saveSumSquaredLossList(self,i,error):
     lossFile.write(str(i)+","+str(error.tolist())+'\n')

 def saveWeights(self):
      np.savetxt("weightsLayerl.txt", self.W1, fmt="%s")
      np.savetxt("weightsLayer2.txt", self.W2, fmt="%s")

  def predictOutput(self):
      print ("Predicted XOR output data based on trained weights: ")
      print ("Expected (X1-X3): \n" + str(xPredicted))
      print ("Output (Y1): \n" + str(self.feedForward(xPredicted)))

myNeuralNetwork = Neural_Network()
trainingEpochs = 10000
for i in range(trainingEpochs):
    print ("Epoch * " + str(i) + "\n")
    print ("Network Input : \n" + str(X))
    print ("Expected Output of XOR Gate Neural Network: \n" + str(y))
    print ("XOR Gate Actual Output: \n"+ str(myNeuralNetwork.feedForward(X)))
    Loss = np.mean(np.square(y - myNeuralNetwork.feedForward(X)))
    myNeuralNetwork.saveSumSquaredLossList(i,Loss)
    print ("Sum Squared Loss: \n" + str(Loss))
    print ("\n")
    myNeuralNetwork.trainNetwork(X, y)

myNeuralNetwork.saveWeights()
myNeuralNetwork.predictOutput()
```