

# **Introduction to neural networks using python.**

**John Pazarzis, 2017**

Python.....	3
Installation for MS Windows.....	4
Executing python.....	5
The most useful python elements.....	6
Variable types.....	7
Numeric types (int, float, long).....	7
String.....	8
List.....	8
Dictionary.....	10
Tuple.....	11
Functions.....	11
Artificial Neural networks (ANN).....	13
Why we need A NN.....	13
Biological neural networks (BNN).....	15
The simplest ANN possible.....	16
Adding the bias node.....	17
Adding multiple input values to the Single layer ANN.....	19
Adding the Threshold.....	20
A Single Layer ANN with threshold.....	21
Linear Classification.....	22
Solving the binary logical functions using Neural Networks.....	23
Multiple layer ANN.....	25
XOR operator.....	25
Back-propagation Algorithm.....	26
Sigmoid Function.....	29
The Back-propagation process.....	31
Using a neural network to classify iris.....	32
Bibliography.....	37

# Python

Python is an open source, dynamic object-oriented language which excels in expressibility and cleanness of protruded code. It is one of the most commonly used generic purpose languages and has become one of the most popular choices for a very wide range of application types. Some of the companies and organizations who are heavy users of python are the following:

- Google
- Facebook
- Dropbox
- Yahoo
- NASA
- IBM
- Bank Of America

Python is one of the easiest languages for the beginner, who can start writing useful scripts in a matter of days as its excels in detail hiding and expressibility while the comprehensive standard library and extremely wide universe of third party libraries provide a very rich ecosystem that can solve a wide range of problems with minimal effort. Code written in python can be executed in virtually all the modern computers solving very well the problem of platform compatibility. Although it is true that python does not produce very fast code, this problem can be solved by C extensions in cases where performance is a bottleneck and consists a first priority.

Python has become one of the top choices when it comes to scientific programming and it provides an extensive set of related open source libraries that can easily be used by non professional programmers who still need to develop

code to solve mathematical and scientific problems. Some of the most important libraries are numpy, scipy and matplotlib, pandas, anacoda, pybrain, networkx and many more

## Installation for MS Windows

- Python
  - Point your browser to <https://www.python.org/downloads/windows/>
  - Select the Windows x86-64 executable installer for:  
Python 3.6.1 - 2017-03-21
  - Install it by double clicking of the downloaded file.
- Add python and pip to path.
  - Open Control Panel\All Control Panel Items\System | advanced system settings | Environment Variables
  - Append the following to the path environment variable:  
;C:\Python34;C:\Python34\Scripts
- Install nosetests by running: pip install nose from the command line.
- Install git from the following link: <https://git-scm.com/download/win>

Following the above instructions you will have the python ,pip , nosetests and git available in your system.

Pip is a python utility that allows for very simple installation of third party libraries with a single command. For example to install the widely used http wrapper **requests** you simply execute the following command:

```
pip install requests
```

Nosetests is a utility to automatically run unit tests from the command line. What it does is that it discovers all the classes that extend a specific test class (unittest.TestCase) and it executes all the tests that are implemented in it.

Git is a very extensively used version control system that makes it easy to download source code from third party repositories and of course to create your own.

## Executing python

There are two main ways to write and execute python code. We can either write the code in an editor, create the necessary unit tests and execute it from the command line passing the script through the python interpreter or we can use an interactive environment ((like the build in python interpreter or the more capable ipython) to execute our code in a line by line fashion. In this document I will use both approaches depending in the problem to solve. For now let's use the python interpreter started from the command prompt by typing python which will display the following prompt:

```
Python 2.7.6 (default, Oct 26 2016, 20:30:19)
[GCC 4.8.4] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At this point we can enter our commands and execute them through interpreter which will print the output in the screen.

For example giving the following command:

```
>>> 6 * 12
```

Will print the following output:

```
72
```

We can create a variable by following the following syntax:

```
a = 123.12
```

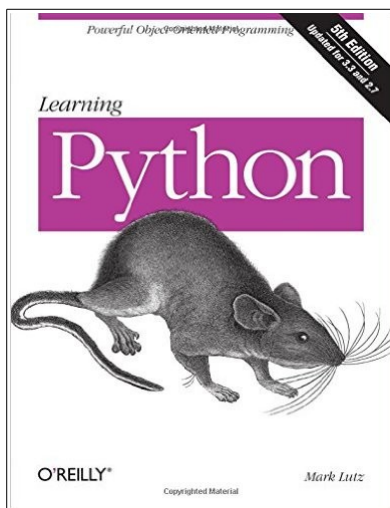
or

```
b = 'hello world'
```

## The most useful python elements

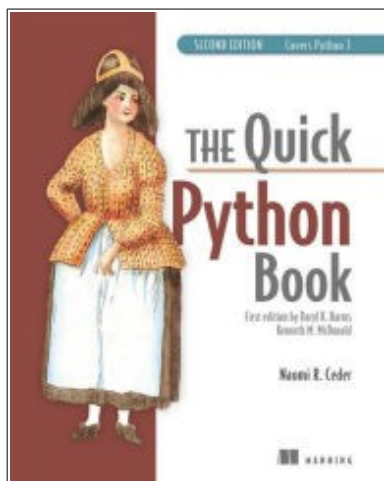
The purpose of this document is not to cover a detailed introduction to python, something that is served by a very wide range of books, videos and websites but to provide some very generic, basic and gentle presentation of the most necessary topics that are needed to follow the neural network material that is covered in the following chapters.

For a systematic and thorough introduction to python the reader can refer in the following books:



### Learning Python by Mark Lutz

This is one of the best “first” books in python starting from the very basics and continuing to more advanced topics covering most of the knowledge that is going to be needed by an application developer.



### The Quick Python Book by Naomi R. Ceder

Also an excellent introduction to python that does not cover as much ground as the previous one but it still covers the topic extensively.

## Variable types

A python variables is created automatically when an object is assigned to a name using the equal sign (=) . Some examples of assignment are the following:

```
i = 100      # Integer.
f = 1000.0   # Floating point.
city = "New York" # String.
```

In python we can assign multiple variables in the same line of code:

```
a, b, c = 1.2, 'test', 3
```

A variable can be deleted using the del statement as it can be seen in the following example:

```
i = 100
print i
del i
print I
```

Running this program will raise an exception:

```
100
Traceback (most recent call last):
  File "/home/ipazarzis/samples/junk1.py", line 4, in <module>
    print i
NameError: name 'i' is not defined
```

## Numeric types (int, float, long)

Unlike to languages like C++ or Java, in python we do not explicitly specify the numeric type that we want to use, instead we simply assign a value to variable and the environment automatically decides what kind of type to use. There is no maximum value for a long value which is only restricted by the hardware where the program is running. The following code shows some uses of numeric types:

```
x = 12
y = 28.2
z = x + y
print z # prints 40.2
print x / y # prints 0.425531914894
print 3 / 2 # prints 1
print 3 / 2. # prints 1.5
```

Running this program produces the following output:

```
40.2
0.425531914894
1
1.5
```

## String

Strings in python can be created by enclosing characters in quotes (single or double quotes can be used as far as the same type is used in both ends). The string interface is very comprehensive and allows many different operations like it can be seen in the following code:

```
s1 = "hello"
s2 = "world"
s = s1 + ' ' + s2
print s # prints (hello world)
s = '({} {})'.format(s1, s2)
print s
print s[1] # prints h
print s1.upper() # prints HELLO
print s2.lower() # prints world
print (s1 + ' ' + s2).title() # prints Hello World
```

Running this program produces the following output:

```
hello world
(hello world)
h
HELLO
world
Hello World
```

## List

Creates a collection of objects that can be iterated and provides a rich interface of function to manipulate it. The following script displays some common uses of list:

```
items = ['test1', 123, 'test2']
print 'Number of items before the append: {}'.format(len(items))
items.append('test3')
print 'Number of items after the append: {}'.format(len(items))
print 'Printout of all items..'
for item in items:
    print '\t', item
print
only_strings = [item for item in items if type(item) is str]
print 'Printout of all strings only..'
for item in only_strings:
    print '\t', item
print
```



Running this program produces the following output:

```
Number of items before the append: 3
Number of items after the append: 4
Printout of all items..
    test1
    123
    test2
    test3

Printout of all strings only..
    test1
    test2
    test3
```

What makes list very useful is the great flexibility that it provides in the way that we can refer to one of its elements by using a powerful indexing mechanism.

Assuming a list like a following:

```
x = [1, 'a', 'this is a test', 123.12, 98]
```

we can access any of its elements by passing its zero-based index enclosed within brackets, for example:

```
x[2]
```

points to the variable:

```
'this is a test'
```

We can access the last item of the list by using -1 as the index:

```
x[-1]
```

points to the variable:

```
98
```

Note that the following expression is equivalent:

```
x[4]
```

To access a subset of the list we use the following notation:

```
x[1: 3]
```

which points to:

```
['a', 'this is a test']
```

Note that the ending index (3 in this case) designates the last element of the array which will not be included in the returned substring.

The following code shows some of the uses of the indexes:

```
x = [1, 'a', 'this is a test', 123.12, 98]
print x[2:] # ['this is a test', 123.12, 98]
print x[:3] # [1, 'a', 'this is a test']
print x[2:-1] # ['this is a test', 123.12]
```

which creates the following output:

```
['this is a test', 123.12, 98]
[1, 'a', 'this is a test']
['this is a test', 123.12]
```

## Dictionary

As in any other language, a python dictionary is an associative array (also known as a hash). It allows the user to associate a **key** to **value** and allow retrieval based on the former. The value can be of any possible type while the key must be a hashable type. The following code shows how a dictionary can be used in python:

```
capitals = {
    'Canada': 'Ottawa',
    'USA': 'Washington, D.C.',
    'Mexico': 'Mexico City'
}
print capitals['USA']
countries = capitals.keys()
print countries
print capitals.values()
for country, capital in capitals.items():
    print country, capital
```

Running this program produces the following output:

```
Washington, D.C.
['Canada', 'USA', 'Mexico']
['Ottawa', 'Washington, D.C.', 'Mexico City']
Canada Ottawa
USA Washington, D.C.
```

## Tuple

A tuple is an immutable list meaning that once it is created we can no longer append or remove items from it. It is defined like a list but instead of square brackets we use parentheses.

The code that we used above for the list works exactly the same if we use a tuple instead:

```
x = (1, 'a', 'this is a test', 123.12, 98)
print x[2:] # ('this is a test', 123.12, 98)
print x[:3] # (1, 'a', 'this is a test')
print x[2:-1] # ('this is a test', 123.12)
```

Note that the returned value of :

```
print x[2:]
```

is now a tuple instead of a list, meaning that it is not possible to add a new item to it..

A useful property of a tuple is that If all the items it contains are immutable then it can be used as a key for dictionary.

## Functions

A python function receives specific input from its user, performs some processing and finally exists returning a value as it can be seen in the following example:

```
def Add(a, b):
    return a + b
```

The Add function defines an entry point that received two arguments and returns their sum.

The following syntax is used when we do not know in advance how many arguments to expect:

```
def CalculateAverage(*n):
    return sum(n) * 1. / len(n)
```

This function receives any number of arguments, calculates its average using the

build in functions sum and len and returns their average. A user of this function can do the following:

```
x = CalculateAverage(1, 2, 10)
```

which sets the value of x to 4.333. What is happening here is that the \*n input argument of the CalculateAverage function is passed as a tuple aggregating the values that were used. Within the body of the function we can use the n as any other tuple.

Similarly, we can pass named parameters as can be see in the following example:

```
def PrintCityCountry(**kwargs):  
    for key, value in kwargs.items():  
        print key, value
```

Which when called as follows:

```
PrintCityCountry(city='New York', country='USA')
```

Creates the following output:

```
city New York  
country USA
```

# Artificial Neural networks (ANN)

## Why we need A NN

The most common case when it comes to computer programming, has to do with solving a problem using an algorithm that consists of a decision tree resembling the necessary steps to evaluate a solutions. Following this approach, the solution to a problem can be implemented as a model that expects a certain number of parameters that are processed and based on them the solution is returned to the user. A trivial example of this type of a “model” can be seen in the following python function that expects a distance measured in centimeters and converts it to inches:

```
def CentimetersToInches(x):  
    return x / 2.54
```

To convert centimeters to inches using this function can be done as follows:

```
distance_in_centimeters = 198.12  
distance_in_inches = CentimetersToInches(distance_in_centimeters)
```

After executing these two lines of code the variable `distance_in_inches` will be assigned the value 78.0 which is the equivalent distance in inches.

Although this example is trivial, it is still representative of the majority of programs that are used today. Building extremely complex systems, like a relational database, an editor or a word processor relies in this type of entry points that can be imagined as deterministic black boxes, that know how to handle their input and

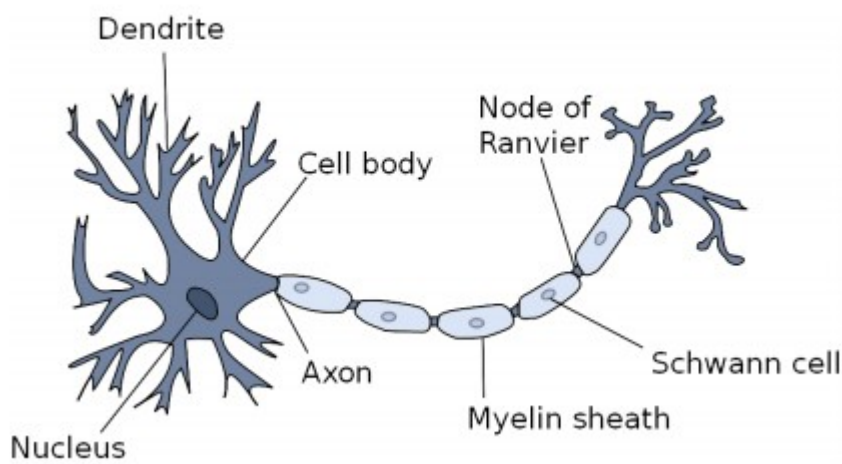
process it accordingly.

Of course, the development of the centimeters to inches converter, assumes that we know the underlined logic which is used to describe the solution in a language that can be executed by the computer, which will always follow exactly the same steps and return the same result when fed by the same data.

This programming paradigm, is applicable when we know exactly the required logic; still are many cases where although we know the parameters and the result values we do not know the logic to calculate them based on the input and these are the type of problems where artificial intelligence is applicable. Let's say that we want to develop a program that will be able to recognize handwritten input. How can we recognize a character that can be written in so many different ways? It is virtually impossible to create a geometrical model to recognize all the possible formats and shapes that handwritten text can take. One of the ways to solve this problem is by using a neural network which need to be trained based on a mapping of many handwritten characters (pattern) to their corresponding letter (target). The process of handwriting recognition is described as **classification** since it tries to detect specific classes of data, in other words each letter of the alphabet consists a specific class that each pattern will match or not.

## Biological neural networks (BNN)

Artificial neural networks are inspired by the biological neural networks and more precisely from the human brain. A simplified view of a biological neural network can be seen in the following image:

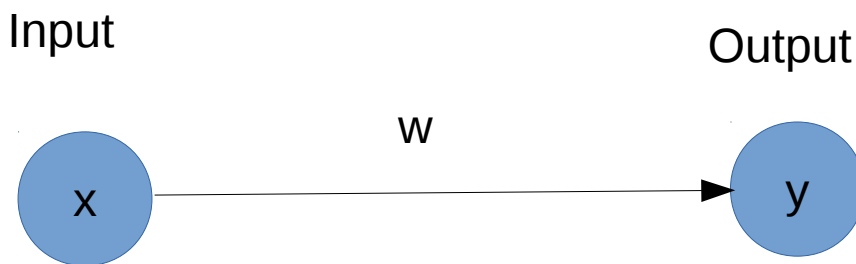


The neural cell that we see in this picture accepts signals from the environment through the dendrites that are connected to it and “fires” transmitting a new signal over its axon reaching other neurons in turn. It is important to realize that all the information transmitted by dendrites and neurons consist of a very low analog electrical signals.

The fact that BNN are based on analog signals can present some challenges when we try to simulate them on a digital computer and we will see soon how to go around this problem by using an activation function.

## The simplest ANN possible

The simplest possible ANN consists of one input and one output neural as we can see in the following image:



Where  $x$  is the value of the input,  $w$  is the weight of the connection and  $y$  the output which is given by the following formula:

$$y = x * w$$

This trivial neural network can be used to solve the conversion of centimeters to inches that we solve before with a python function by assigning to  $w$  the value 0.3937 (which is given by the  $1 / 2.54$ ).

Using this network we can now calculate the conversion to inches which will result in the same logic as the code that we have used before in the python example.



## Adding the bias node

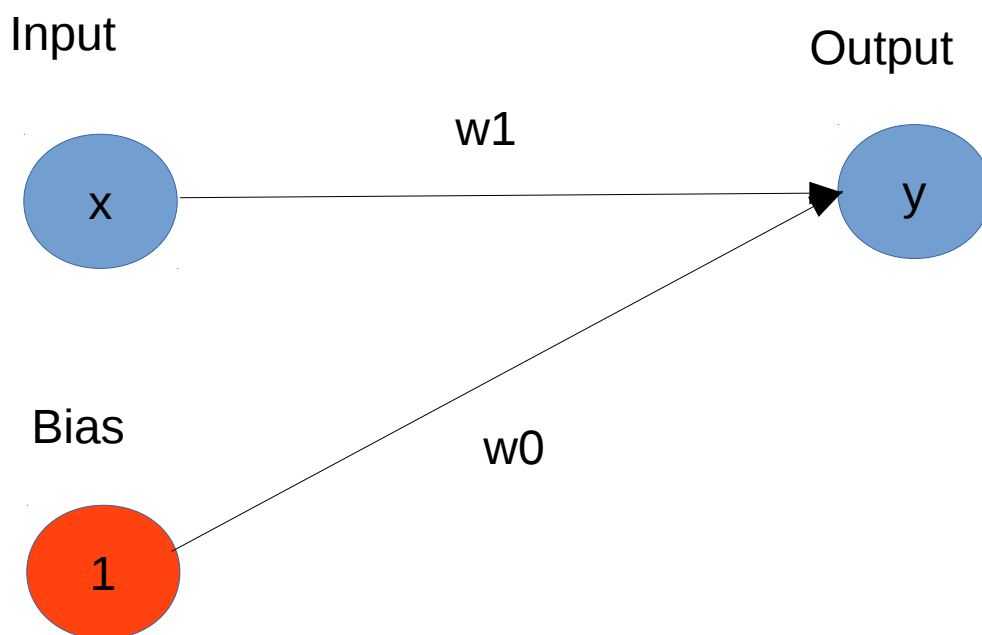
The centimeters to inches conversion ANN that we describe in the previous example can solve a problem that fits a very specific description which can be expressed as linear function which passes from the origin similar to the following:

$$y = a * x$$

The next problem we need to solve is how to describe the solution of any linear relation using an ANN. Since generic formula for a linear function is the following:

$$y = a * x + b$$

it is easy to realize that we can represent its solution using the following ANN:

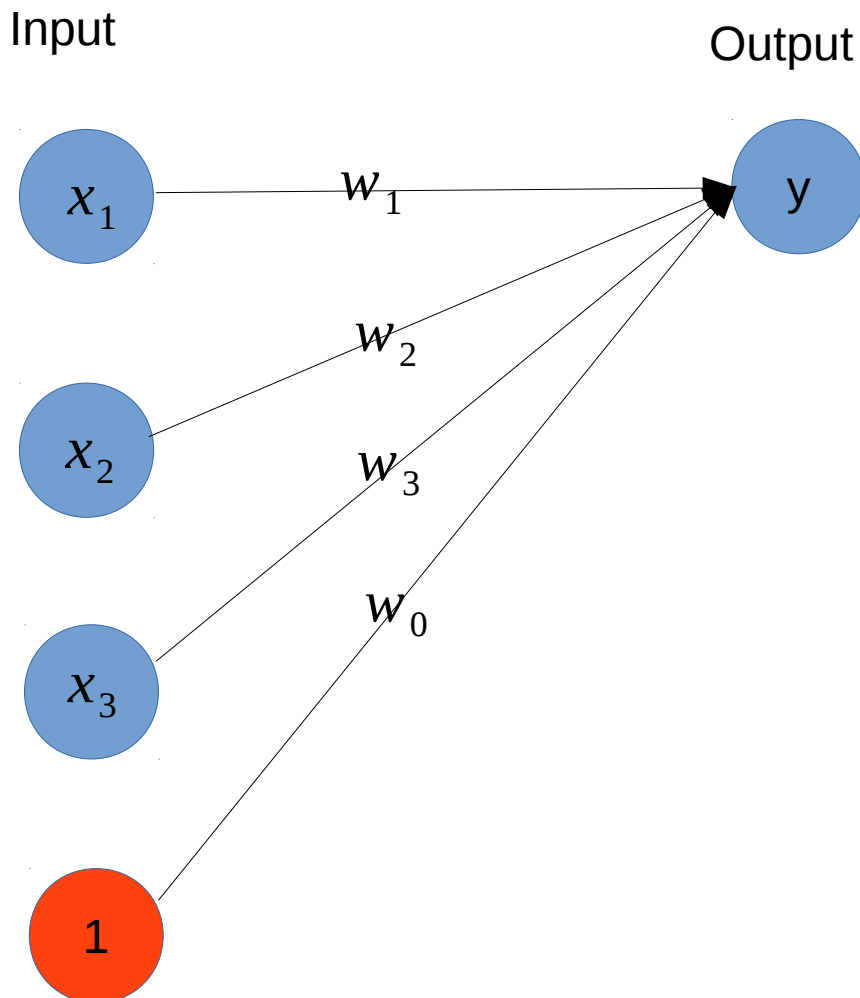


The value of the bias is always 1 so we can now express the ANN by the following formula:

$$y = w_0 + w_1 * x$$

## Adding multiple input values to the Single layer ANN

We can now extend our ANN to support multiple input values to handle multiple dimension problems as follows:

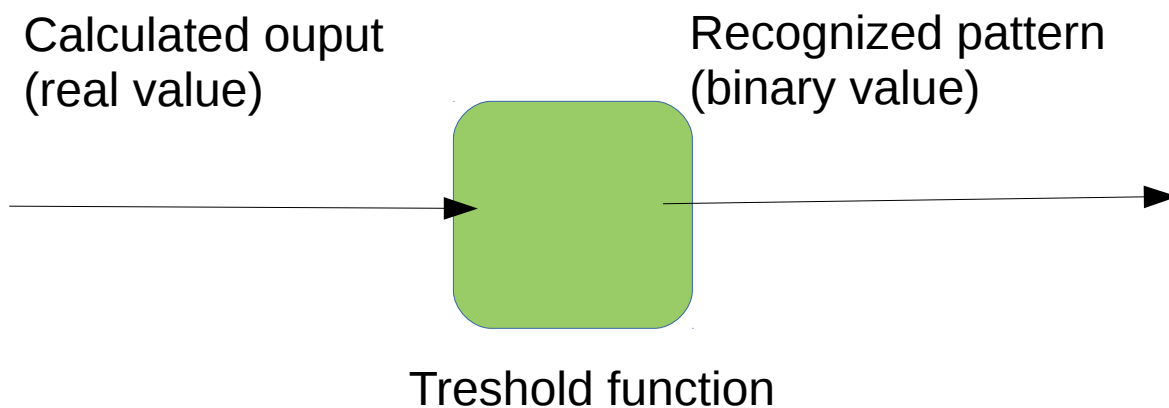


The single-layer multiple input ANN can be expressed as follows:

$$y = w_0 + w_1 * x_1 + w_2 * x_2 + w_3 * x_3$$

## Adding the Threshold

The Single layer ANN as we have seen so far receives its input as real numbers and provides its output as a real number as well. Assuming that we need to recognize a binary pattern we need to convert the output from a real to a binary value, usually either a 0 or a 1. To do so we need to pass the output value through a threshold function that will convert it to a binary as it can be visualized in the following picture:



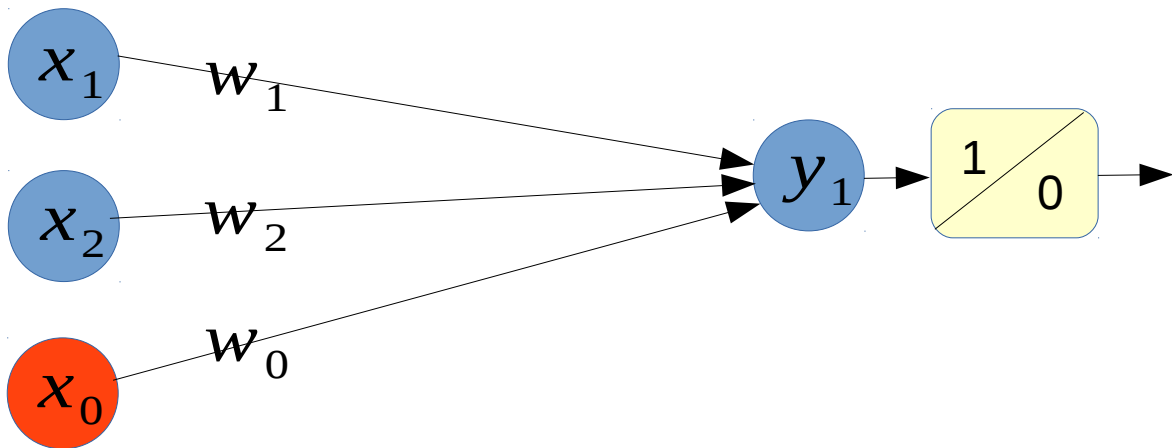
The threshold function can be expressed as follows:

$$y = \begin{cases} 1 & \text{if } u \geq \theta \\ 0 & \text{if } u < \theta \end{cases}$$

Where  $u$  is the output that was calculated and  $\theta$  the threshold value to use.

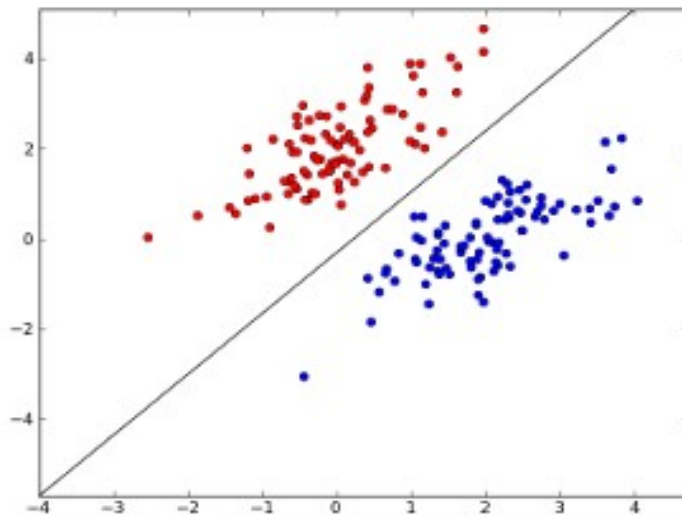
## A Single Layer ANN with threshold

The following picture describes a Single layer ANN with a threshold function that can be used to identify linear classified binary patterns:



## Linear Classification

Problems similar to the AND or OR logical operators can be solved by a linear classifier which is a mathematical model applicable if the separation among the classes can be described by a straight line, a plane or a multi-plane. The following graph shows two types of data that are separated by a straight line:



in this case we can use a Single Layer neural network to calculate the slope and intercept of the line.

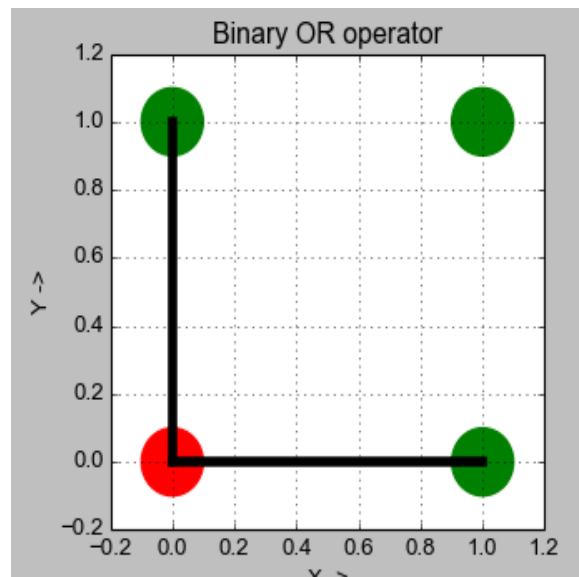
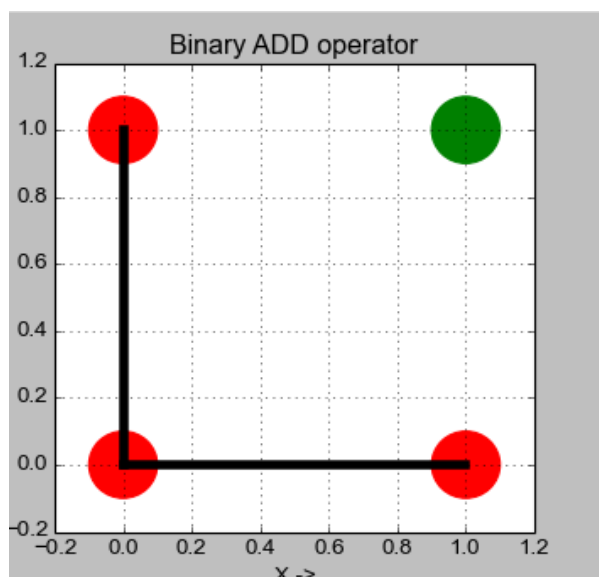
## Solving the binary logical functions using Neural Networks

Among the most simple examples of applications of neural networks is the solution of the binary logical functions AND, OR and NOT. Representing true as 1 and false as 0 the logical functions AND and OR results in the following mapping:

AND					OR				
1	AND	1	=	1	1	OR	1	=	1
1	AND	0	=	0	1	OR	0	=	1
0	AND	1	=	0	0	OR	1	=	1
0	AND	0	=	0	0	OR	0	=	0

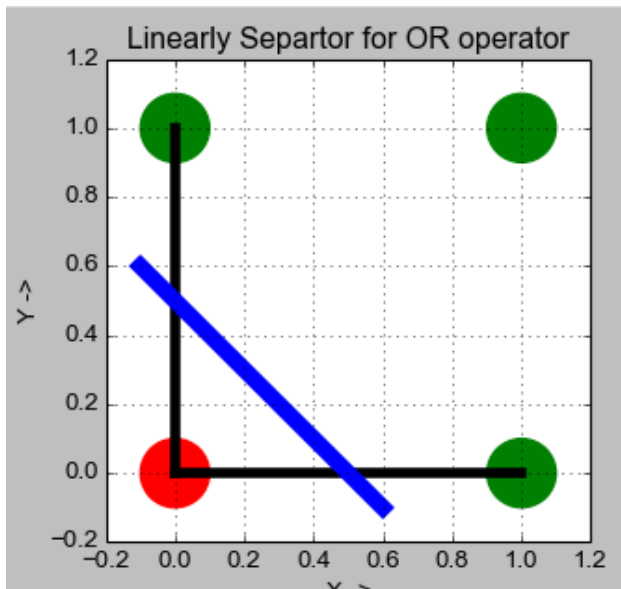
Table 1

the graphic representations of the AND and OR functions are the following:



Note that both of these graphs share a common characteristic: in both cases it is

possible to draw a straight line that clearly separates the two possible classes.



If for example the in the following graph of the OR operator the blue line divides the plane to two areas, each one containing the same classes of data (only green and only red).

To discover the separating line that divides the classification of the OR operator to TRUE and FALSE (or 1 and 0) we can use the simplest form of a neural network which consists of a set of input parameters and a single output. In our case the input must consist from the the two binary values of x and y and the output of a single binary value.

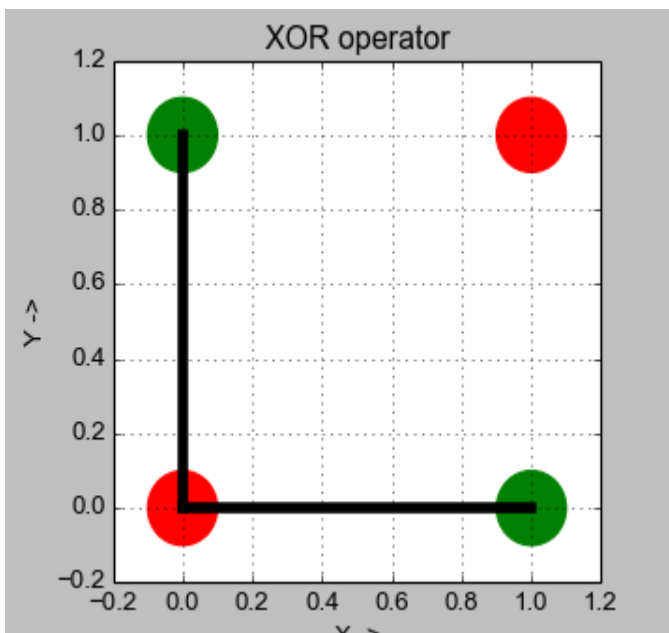


## Multiple layer ANN

### XOR operator

The advent of multilayer neural networks sprang from the need to implement the XOR logic gate. Early perceptron researchers ran into a problem with XOR. The same problem as with electronic XOR circuits: multiple components were needed to achieve the XOR logic. With electronics, 2 NOT gates, 2 AND gates and an OR gate are usually used. With neural networks, it seemed multiple perceptrons were needed (well, in a manner of speaking). To be more precise, abstract perceptron activities needed to be linked together in specific sequences and altered to function as a single unit. Thus were born multi-layer networks.

Why go to all the trouble to make the XOR network? Well, two reasons: (1) a lot of problems in circuit design were solved with the advent of the XOR gate, and (2) the XOR network opened the door to far more interesting neural network and machine learning designs.

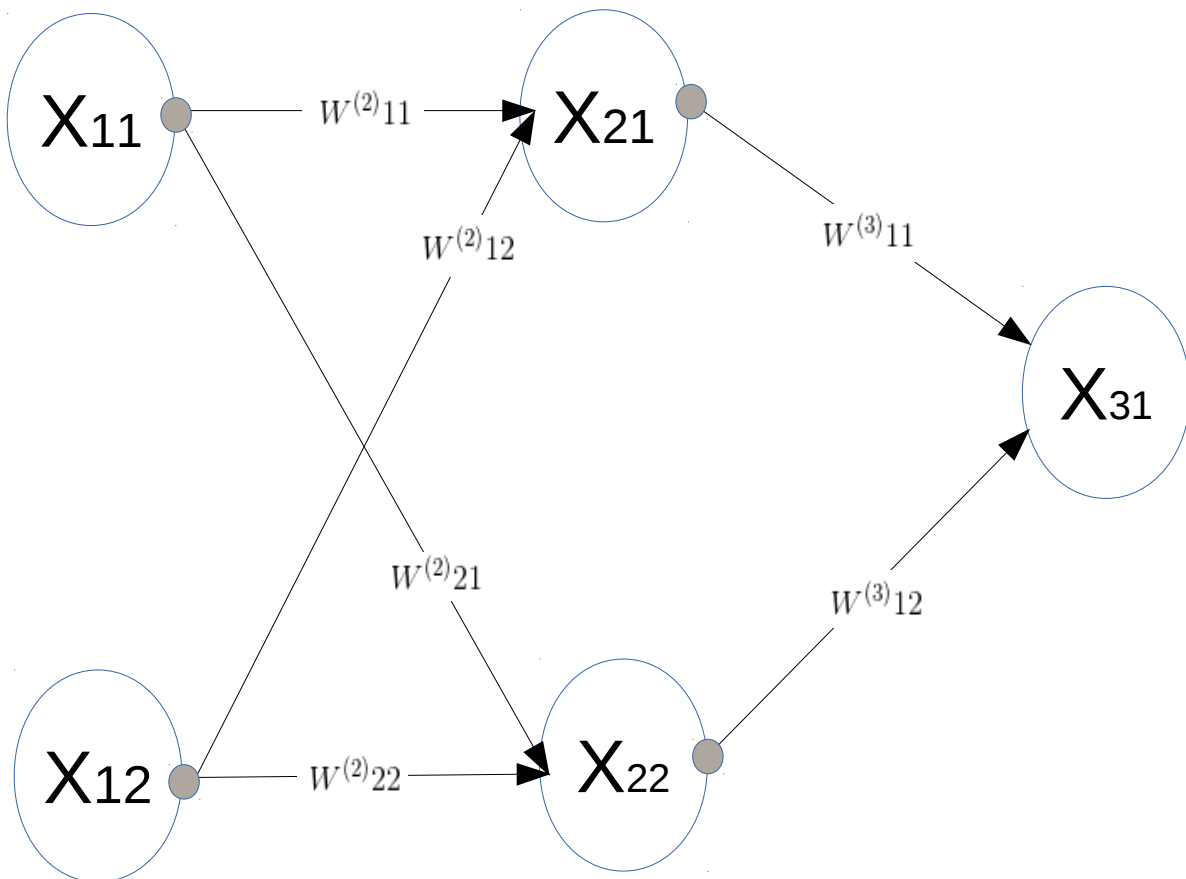


This is the graphical representation of the XOR operator. It is easy to realize by a visual inspection that it is impossible to separate the two classes (red and green) by a straight line.

## Back-propagation Algorithm

The objective of the back-propagation algorithm is to distribute the errors

Let's assume the following perceptron:



The input to this network is provided by its first layer ( $x_{11}$ ,  $x_{12}$ ) and its output by the last layer ( $X_{31}$ ). As can be seen in the picture, every node is connected with all the nodes in the subsequent layer. Each connection is assigned a weight, which usually is a small value between 0 and 1.

The value of each node in any layer except the input is calculated by summing the

products of the weight of each backward connection by the value of the connected node.

Having said this, we can calculate the values of the given network by the following calculations:

Hidden Layer

$$X_{21} = W_{11}^{(2)} * X_{11} + W_{12}^{(2)} * X_{12}$$

$$X_{22} = W_{21}^{(2)} * X_{11} + W_{22}^{(2)} * X_{12}$$

Output Layer

$$X_{31} = W_{11}^{(3)} * X_{21} + W_{12}^{(3)} * X_{22}$$

## Numerical example

Let the input values of the example perceptron be as follows:

$$X_{11} = 3.5$$

$$X_{12} = 2.8$$

while the connection weights are as follows:

Hidden layer

$$W_{2(11)} = 0.23$$

$$W_{2(12)} = 0.18$$

$$W_{2(21)} = 0.47$$

$$W_{2(22)} = 0.14$$

Output layer

$$W_{3(11)} = 0.56$$

$$W_{3(12)} = 0.71$$

Calculating the values of the hidden nodes

$$X_{21} = 0.23 * 3.5 + 0.18 * 2.8 = 1.31$$

$$X_{22} = 0.47 * 3.5 + 0.14 * 2.8 = 2.04$$

Calculating the output layer

$$X_{31} = 0.56 * 1.31 + 0.71 * 2.04 = 2.18$$

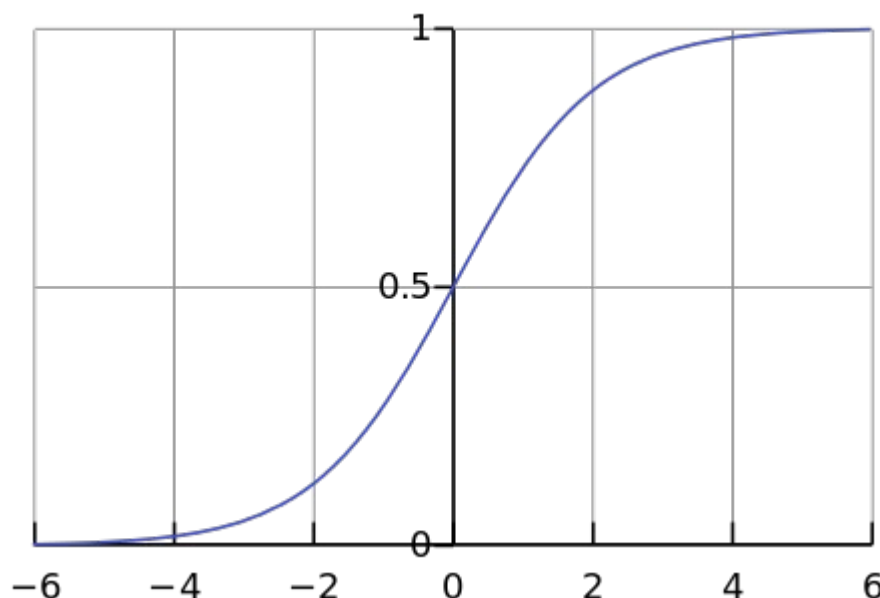
## Sigmoid Function

The process of calculating the value of each node based on the nodes that exist in the previous layers is called feed-forward. Applying the feed-forward calculations all the way to the output layer will assign a value to every node in the network. Assuming that we know in advance the expected pattern (target) for the input used, makes it possible to calculate the error that corresponds to the weights of the network. Since we have started with randomly assigned weights we expect the result to be random as well and completely wrong. The objective of the neural network training is to adjust the weights of the synapses in such a way that the produced error will be minimal.

Before we delve into the details of the training process we need to add another important component in the calculation of the node values. The most common technique used to polarize the value of a node, is to pass it through an activation function which for the purpose of this document will be the sigmoid function which is given by the following formula:

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

Which can be plotted as follows:



The interesting thing about the sigmoid function is that it as the x increases it reached asymptotically the value of 1 while as it decreases it reaches 0.

Another useful characteristic of the sigmoid is that it is very easy to caculate its derivate can be seen here:

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

$$\frac{d}{dx} f(x) = \frac{e^x * (1 + e^x) - e^x * e^x}{(1 + e^x)^2}$$

$$\frac{d}{dx} f(x) = \frac{e^x}{(1 + e^x)^2} = f(x)(1 - f(x))$$

One of the reasons why we pick sigmoid for the back-propagation algorithm lies exactly in the simplicity of the calculation of the derivate that is used for the error adjustments and we will shortly see why this is the case.

At this point we will use the sigmoid to transform the output of each node, meaning the its calculation now requires the following calculations:

$$\text{net}_j = \sum_{i \in I} (o_i \cdot w_{i,j})$$

$$\text{Value}_j = \text{Sigmoid}(\text{net}_j)$$

## The Back-propagation process

### Calculate error for output layer

To back propagate the error that was calculated by subtracting the target minus the observed value in the output layer we use the following formula:

$$Error_i = Output_i * (1 - Output_i) * (Target_i - Output_i)$$




### Calculate error for a hidden layer

Unlike the output layer we can't calculate these directly (because we don't have a Target), so we Back Propagate them from the output layer (hence the name of the algorithm). This is done by taking the Errors from the output neurons and running them back through the weights to get the hidden layer errors as described in the following equation:

$$Error_h = \sum Error_i * W_{ih}$$

The training process of the ANN consists of subsequent passes of all the available training data through a feed forward – back propagation process until we reach the desired level of error.

## Using a neural network to classify iris.

Versicolor	Setosa	Virginica
		

The family of Iris flowers consists of three species (versicolor, setosa and virginica). A neural network can be used to identify the type of the flower based on the measurements of length, width of sepals, petals length and width of sepals and petals.

One of the most commonly used data base for neural networks for pattern recognition, is Fisher's Iris data base (Fisher, 1936) . The data set contains 3 classes of 150 instances where each class refers to a type of iris plant. One class is linearly separable from the other two; the latter are not linearly separable from each other. The data base contains the following attributes: 1). sepal length in cm 2). sepal width in cm 3). petal length in cm 4). petal width in cm 5). class: - Setosa - Versicolour - Virginica



The structure of the IRIS database can be seen in following table:

A

sepal_length	sepal_width	petal_length	petal_width	target
6.5	2.8	4.6	1.5	versicolor
6.6	2.9	4.6	1.3	versicolor
7	3.2	4.7	1.4	versicolor
6.3	2.5	4.9	1.5	versicolor
5.1	3.4	1.5	0.2	setosa
5.6	2.5	3.9	1.1	versicolor
5.5	3.5	1.3	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.6	3.2	1.4	0.2	setosa
7.4	2.8	6.1	1.9	virginica
6.6	3	4.4	1.4	versicolor
6.4	2.7	5.3	1.9	virginica
5.1	3.8	1.9	0.4	setosa

perceptron based feed forward, back propagation neural network seems to be the ideal solution to create a classifier for the iris dataset and we will use for this the network class that is part of the sample code.

We need to perform the following data preprocessing present it to the network:

### **Transform nominal target values to a numerical representation**

As you can see in the database subset, the target is described as a “name” that

designates the corresponding class and its value can be one of the following:

#### Available classes

- Setosa
- Virginica
- Versicolor

To make classification easier we need to substitute each class with a corresponding numerical representation that can be easily used to calculate estimation errors. The solution we are going to use is to represent each class as a vector of binary values as follows:

	T1	T2	T3
versicolor	1	0	0
setosa	0	1	0
virginica	0	0	1

By doing so we essentially transform each nominal value to a vector that will be used internally by the network which will be able to form an “opinion” expressed as a collection of three small real numbers ranging from 0 .. 1 and compare it against the predefined value.

#### Normalize input

Going through the values contained in the Iris db (we can call them pattern as well) we see that each column has a different range. This can mislead the

calculations of the NN which will put more weight to those who have larger values. To resolve this issue we need to normalize them before processing. Given the implementation of the NN that we will use, it is easy to see that the best approach will be to scale the values of the patterns in a 0 – 1 range.

To normalize the patterns we need to process each column individually and apply a normalization function to each specific data point.

There are several ways to do this and in the example we will use the simplest one, which is described in the following formula:

$$\text{NormalizedValue}(x) = (x - x_{\min}) / (x_{\max} - x_{\min})$$

Applying the preprocessing steps as described above we end up with the a dataset that looks like this:

```
[0.61, 0.33, 0.61, 0.58] [1, 0, 0]
[0.64, 0.37, 0.61, 0.50] [1, 0, 0]
[0.75, 0.50, 0.63, 0.54] [1, 0, 0]
[0.56, 0.21, 0.66, 0.58] [1, 0, 0]
[0.22, 0.58, 0.08, 0.04] [0, 1, 0]
[0.36, 0.21, 0.49, 0.42] [1, 0, 0]
[0.33, 0.62, 0.05, 0.04] [0, 1, 0]
[0.03, 0.37, 0.07, 0.04] [0, 1, 0]
[0.08, 0.50, 0.07, 0.04] [0, 1, 0]
[0.86, 0.33, 0.86, 0.75] [0, 0, 1]
[0.64, 0.42, 0.58, 0.54] [1, 0, 0]
[0.58, 0.29, 0.73, 0.75] [0, 0, 1]
```

Note that each row consists of two vectors: The first one contains the normalized

patterns (that now range from 0 to 1) and consists of four numbers each one corresponding to one of the features and the second vector represents the target which consists of three binary numbers from which the two are always 0 while only one is 1, following the representation as we defined above.

## Bibliography

Beazley, David, and Brian K Jones. 2013. *Python Cookbook: Recipes for Mastering Python 3*. Beijing; Köln [u.a.]: O'Reilly.

Hellmann, Doug. 2011. *The Python Standard Library by Example*. Upper Saddle River, NJ: Addison-Wesley.

Lutz, Mark. 2011. *Programming Python: [powerful Object-Oriented Programming]*. Sebastopol, Calif: O'Reilly.

2013. *Learning Python Powerful Object-Oriented Programming*. Beijing: O'Reilly.