Laboratory 1 – Neural Networks Basics

In this class we are going to learn the basics of neural networks and their underlying principles.

Objectives of this class:

1. *Understand and code a simple neuron*
2. *Understand how a neuron learns*
3. *Understand its limitations*

Start the class by opening your preferred Python text editor (we recommend Visual Studio) and import the library Numpy.

```python

import numpy as np

```

Now, let's code our first neuron. As you might remember from the lectures, the structure of a neuron is as follows

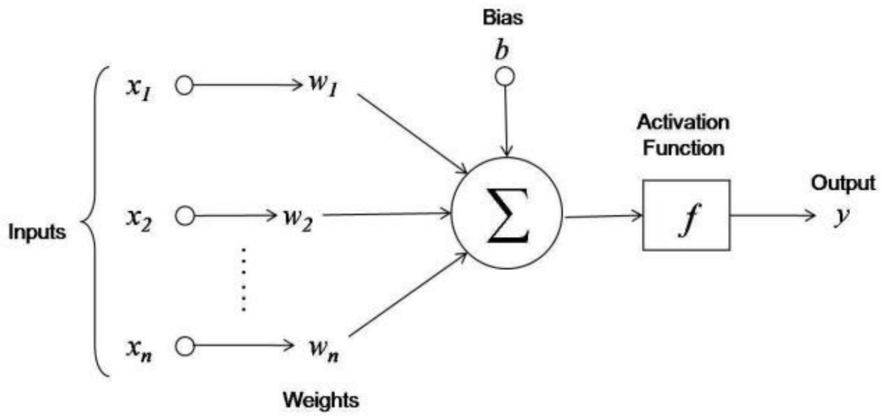


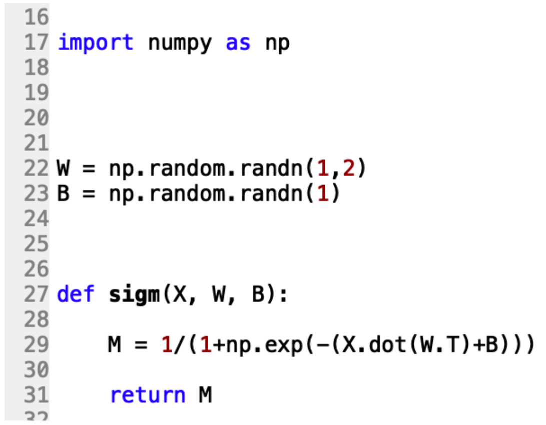
Figure 1: A Neuron. Source: https://towardsdatascience.com/statistics-is-freaking-hard-wtf-is-activation-function-df8342cdf292

The function $f(\cdot)$ represents the core of the neuron and we will see later the effects of the choice of this function. For the time being, let's consider the sigmoid function you have seen during class:

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

Where $w = (w_1, w_2, \dots, w_n)$ and $x = (x_1, x_2, \dots, x_n)$ represent the n-dimensional vectors for the weights and the inputs, b is the bias and $(\cdot)^T$ is the transpose operator.

Now write in your text editor the code required to simulate a neuron with $n = 2$, a sigmoid activation function and random weights. You will need to make use of the methods "np.random.randn" and of ".T" and ".dot" (respectively, picking a random number according to the gaussian distribution, the transpose operator and the dot product).



```

import numpy as np




W = np.random.randn(1,2)
B = np.random.randn(1)



def sigm(X, W, B):

    M = 1/(1+np.exp(-(X.dot(W.T)+B)))

    return M
```

Although the results might appear quite underwhelming at this point, this really represents the core of a neural network and of deep learning. In order to see how the learning process works, we need to setup the weights of this neuron to perform some simple tasks. Let's briefly recall the logical operators. Logical, or Boolean, functions are defined as $g: \{0,1\}^n \rightarrow \{0,1\}$. So, for instance a two dimensional (i.e., $n = 2$) logical function would have the form $g(x_1, x_2) = y$ with $x_1, x_2, y \in \{0,1\}$. In this example we have the following tables (also called "truth tables") associated with function OR and function AND:

OR		
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

AND		
x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

By this point our neuron is not yet able to learn. In order to make our unit smarter, we are going to make use of the gradient descent method to update the weights, depending on the choice of an error function. For this laboratory, we will consider the squared error function.

$$E = (\hat{y} - o)^2$$

$$w_i' = w_i + \eta \frac{dE}{dw_i}$$

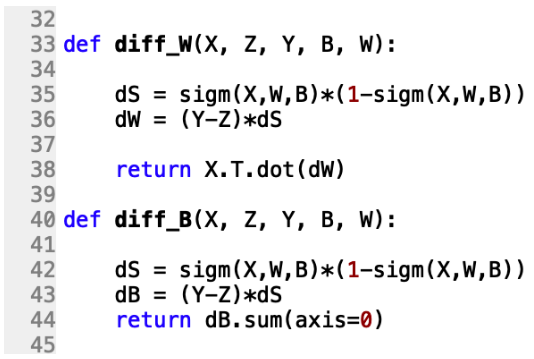
$$b' = b + \eta \frac{dE}{db}$$

Where, $\hat{y}$ represents the correct output, o is the current output of the neuron and η is the learning rate.

Exercise: derive an analytical expression for the formulas above, considering that the derivative the sigmoid can be expressed as,

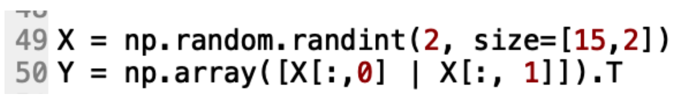
$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Once that is done, you can write the update rules for your weights and bias.



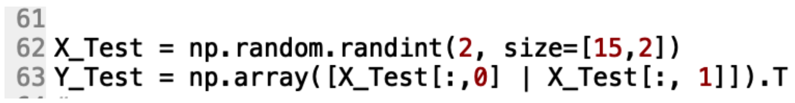
```python

def diff_W(X, Z, Y, B, W):

    dS = sigm(X,W,B)*(1-sigm(X,W,B))
    dW = (Y-Z)*dS

    return X.T.dot(dW)

def diff_B(X, Z, Y, B, W):

    dS = sigm(X,W,B)*(1-sigm(X,W,B))
    dB = (Y-Z)*dS
    return dB.sum(axis=0)

```

Now the final ingredients to teach our neuron are a training set and a test set. Make use of the function “np.random.randint()” to create 15 two-dimensional samples and their respective outputs (| represent the bitwise or operator in Python). This will be the set on which we will train our network.



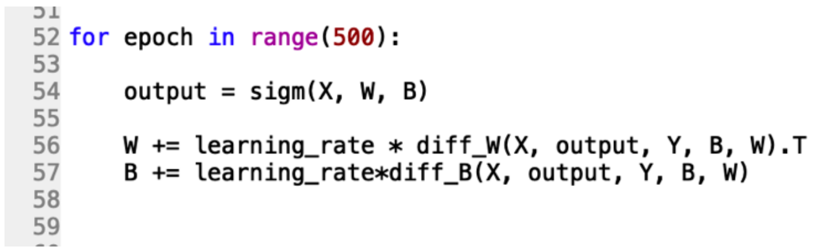
```python
X = np.random.randint(2, size=[15,2])
Y = np.array([X[:,0] | X[:, 1]]).T
```

Do the same for a testing set. This set will only be used to compute the error of our system and we will not perform any training on it (can you explain why?)



```python

X_Test = np.random.randint(2, size=[15,2])
Y_Test = np.array([X_Test[:,0] | X_Test[:, 1]]).T
```

Now, we can finally teach the neuron to emulate the OR function. Set the learning rate to a small number (0.01) and iterate over all the points in the training set (remember to update the weights!) for a large number of epochs (500 is a reasonable amount). To see the effects of the learning, check how the weights and the error change (use the function print) with each iteration. Also, play a bit with the learning rate and the number of iterations and see if you can make it learn faster and reduce the error further. Does the error go to zero? Why not? What could we do to make it exactly zero?



```
for epoch in range(500):

    output = sigm(X, W, B)

    W += learning_rate * diff_W(X, output, Y, B, W).T
    B += learning_rate*diff_B(X, output, Y, B, W)
```

Now, repeat the previous procedure also for the AND (the bitwise AND operator in Python is &) and the XOR (the bitwise AND operator in Python is ^).

XOR		
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Are we able to reproduce the XOR? Can you explain these results?

If you really wanted to solve the XOR just using one neuron, can you think of any solutions that we could apply?