Shallow Neural Network

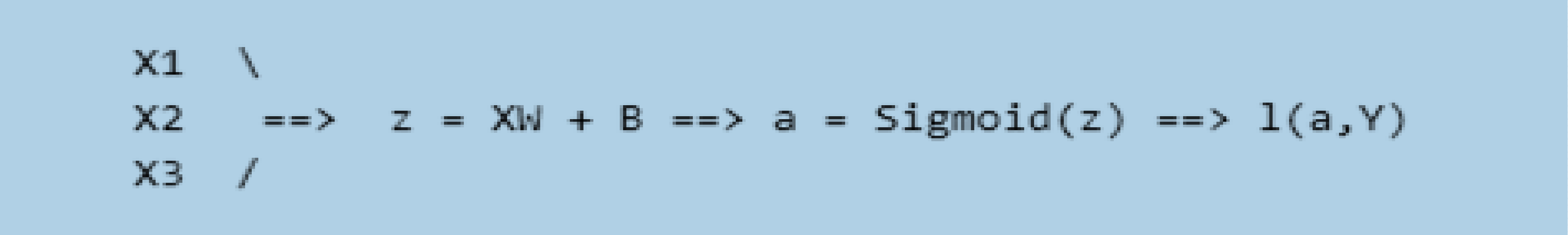
lab3

Eng: Samar Shaaban

Vectorization

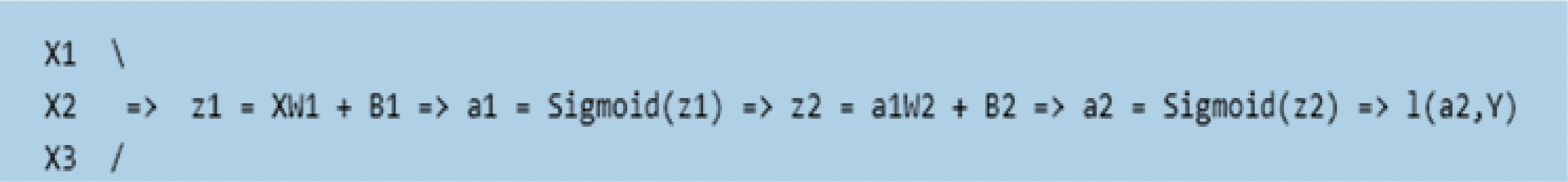
- Deep learning shines when the dataset are big. However for loops will make you wait a lot for a result. That's why we need vectorization to get rid of some of our for loops.
- NumPy library (dot) function is using vectorization by default.
- The vectorization can be done on CPU or GPU. But its faster on GPU.
- Whenever possible avoid for loops.
- Most of the NumPy library methods are vectorized version.

- In logistic regression we had:



```
X1  \
X2   ==>  z = XW + B ==> a = Sigmoid(z) ==> l(a,Y)
X3  /
```

- In neural networks with one hidden layer we will have

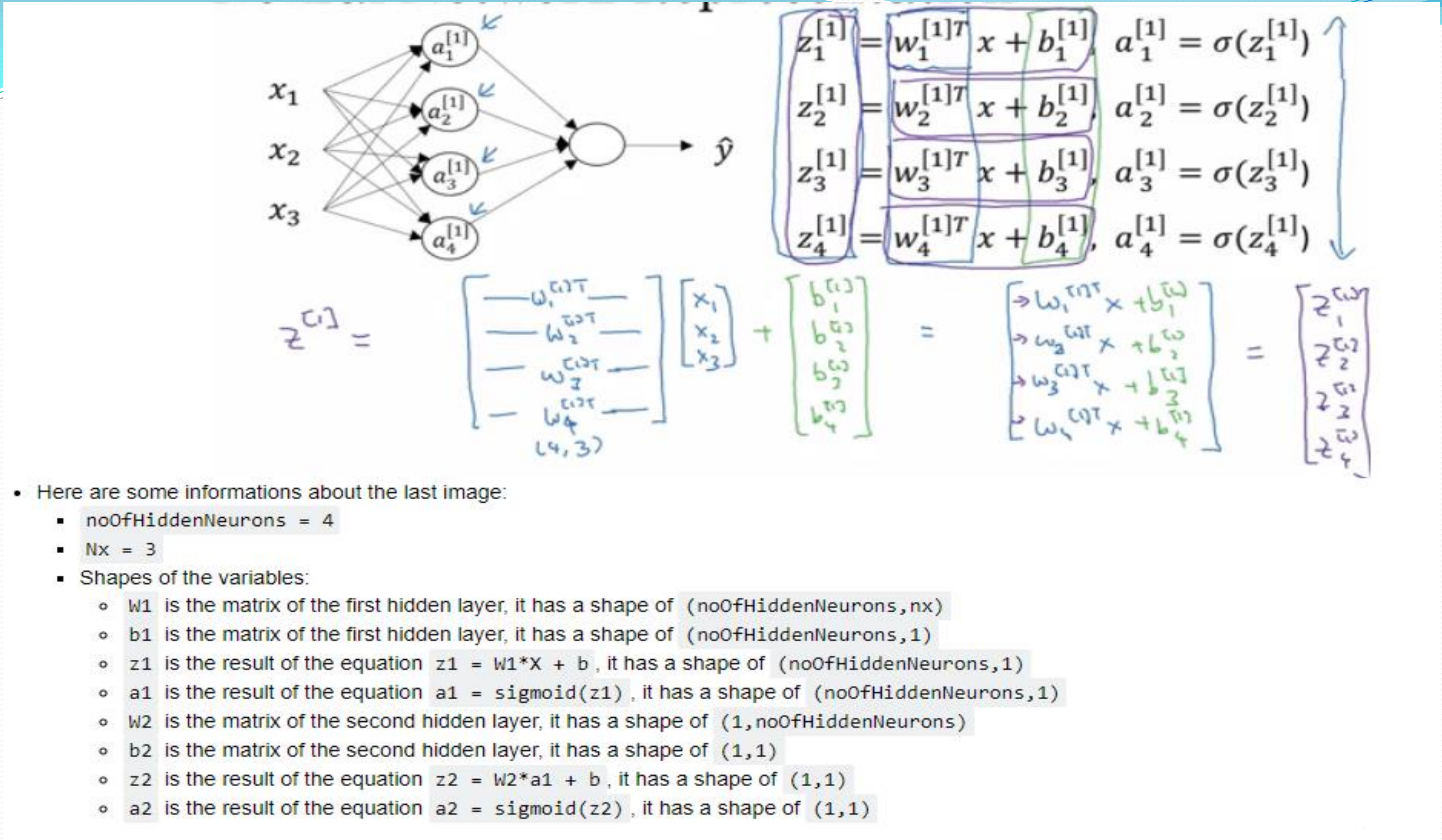


```
X1  \
X2    =>  z1 = XW1 + B1 => a1 = Sigmoid(z1) => z2 = a1W2 + B2 => a2 = Sigmoid(z2) => l(a2,Y)
X3  /
```

  - X is the input vector (X1, X2, X3), and Y is the output variable (1x1)
  - NN is stack of logistic regression objects.

Neural Network Representation

- We will define the neural networks that has one hidden layer.
- NN contains of input layers, hidden layers, output layers.
- Hidden layer means we cant see that layers in the training set.
- $a0 = x$ (the input layer)
- $a1$ will represent the activation of the hidden neurons.
- $a2$ will represent the output layer.
- We are talking about 2 layers NN. The input layer isn't counted.



$$z_1^{[1]} = w_1^{[1]T}x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T}x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T}x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T}x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

$$z^{[1]} = \underbrace{\begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix}}_{(4,3)} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} \rightarrow w_1^{[1]T}x + b_1^{[1]} \\ \rightarrow w_2^{[1]T}x + b_2^{[1]} \\ \rightarrow w_3^{[1]T}x + b_3^{[1]} \\ \rightarrow w_4^{[1]T}x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

- Here are some informations about the last image:
  - `noOfHiddenNeurons = 4`
  - `Nx = 3`
  - Shapes of the variables:
    - `W1` is the matrix of the first hidden layer, it has a shape of `(noOfHiddenNeurons,nx)`
    - `b1` is the matrix of the first hidden layer, it has a shape of `(noOfHiddenNeurons,1)`
    - `z1` is the result of the equation `z1 = W1*X + b`, it has a shape of `(noOfHiddenNeurons,1)`
    - `a1` is the result of the equation `a1 = sigmoid(z1)`, it has a shape of `(noOfHiddenNeurons,1)`
    - `W2` is the matrix of the second hidden layer, it has a shape of `(1,noOfHiddenNeurons)`
    - `b2` is the matrix of the second hidden layer, it has a shape of `(1,1)`
    - `z2` is the result of the equation `z2 = W2*a1 + b`, it has a shape of `(1,1)`
    - `a2` is the result of the equation `a2 = sigmoid(z2)`, it has a shape of `(1,1)`

Activation functions

- So far we are using sigmoid, but in some cases other functions can be a lot better.
- Sigmoid can lead us to gradient decent problem where the updates are so low.
- Sigmoid activation function range is [0,1] **A = 1 / (1 + np.exp(-z))** # Where z is the input matrix
- Tanh activation function range is [-1,1] (Shifted version of sigmoid function)
  - In NumPy we can implement Tanh using one of these methods: A = (np.exp(z) - np.exp(-z)) / (np.exp(z) + np.exp(-z)) # Where z is the input matrix
  - Or A = np.tanh(z) # Where z is the input matrix
- It turns out that the tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer.

Activation functions

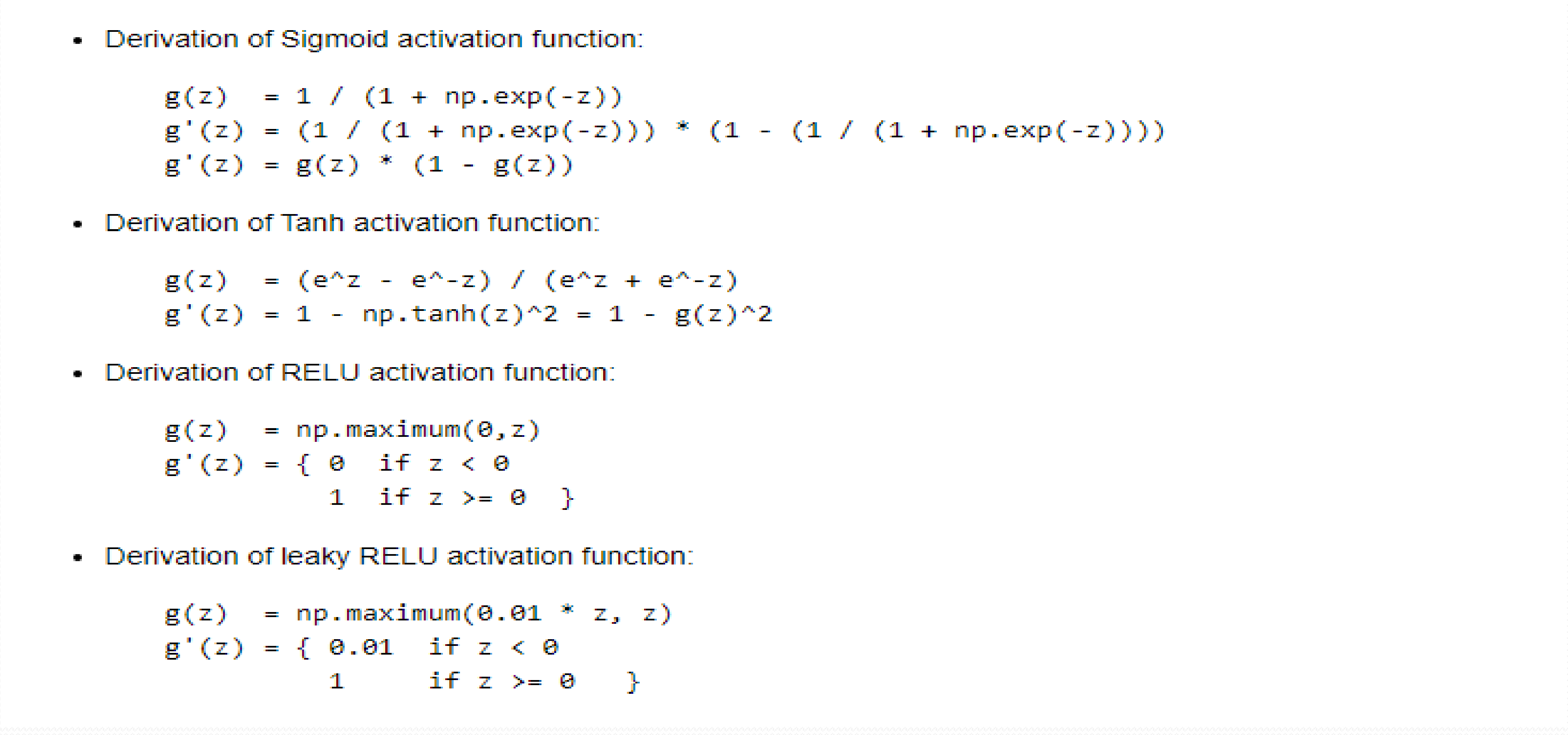
- Sigmoid or Tanh function disadvantage is that if the input is too small or too high, the slope will be near zero which will cause us the gradient decent problem.
- One of the popular activation functions that solved the slow gradient decent is the RELU function. RELU = max(0,z) # so if z is negative the slope is 0 and if z is positive the slope remains linear.
- So here is some basic rule for choosing activation functions, if your classification is between 0 and 1, use the output activation as sigmoid and the others as RELU.
- Leaky RELU activation function different of RELU is that if the input is negative the slope will be so small. It works as RELU but most people uses RELU. Leaky_RELU = max(0.01z,z) #the 0.01 can be a parameter for your algorithm.

Why do you need non-linear activation functions?

- If we removed the activation function from our algorithm that can be called linear activation function.
- Linear activation function will output linear activations
  - Whatever hidden layers you add, the activation will be always linear like logistic regression (So its useless in a lot of complex problems).
- You might use linear activation function in one place - in the output layer if the output is real numbers (regression problem). But even in this case if the output value is non-negative you could use RELU instead

- In NN you will decide a lot of choices like:
  - No of hidden layers.
  - No of neurons in each hidden layer.
  - Learning rate. (The most important parameter)
  - Activation functions.
  - And others..
- It turns out there are no guide lines for that. You should try all activation functions for example

Derivatives of activation functions



- Derivation of Sigmoid activation function:

```
g(z)  = 1 / (1 + np.exp(-z))
g'(z) = (1 / (1 + np.exp(-z))) * (1 - (1 / (1 + np.exp(-z))))
g'(z) = g(z) * (1 - g(z))
```

- Derivation of Tanh activation function:

```
g(z)  = (e^z - e^-z) / (e^z + e^-z)
g'(z) = 1 - np.tanh(z)^2 = 1 - g(z)^2
```

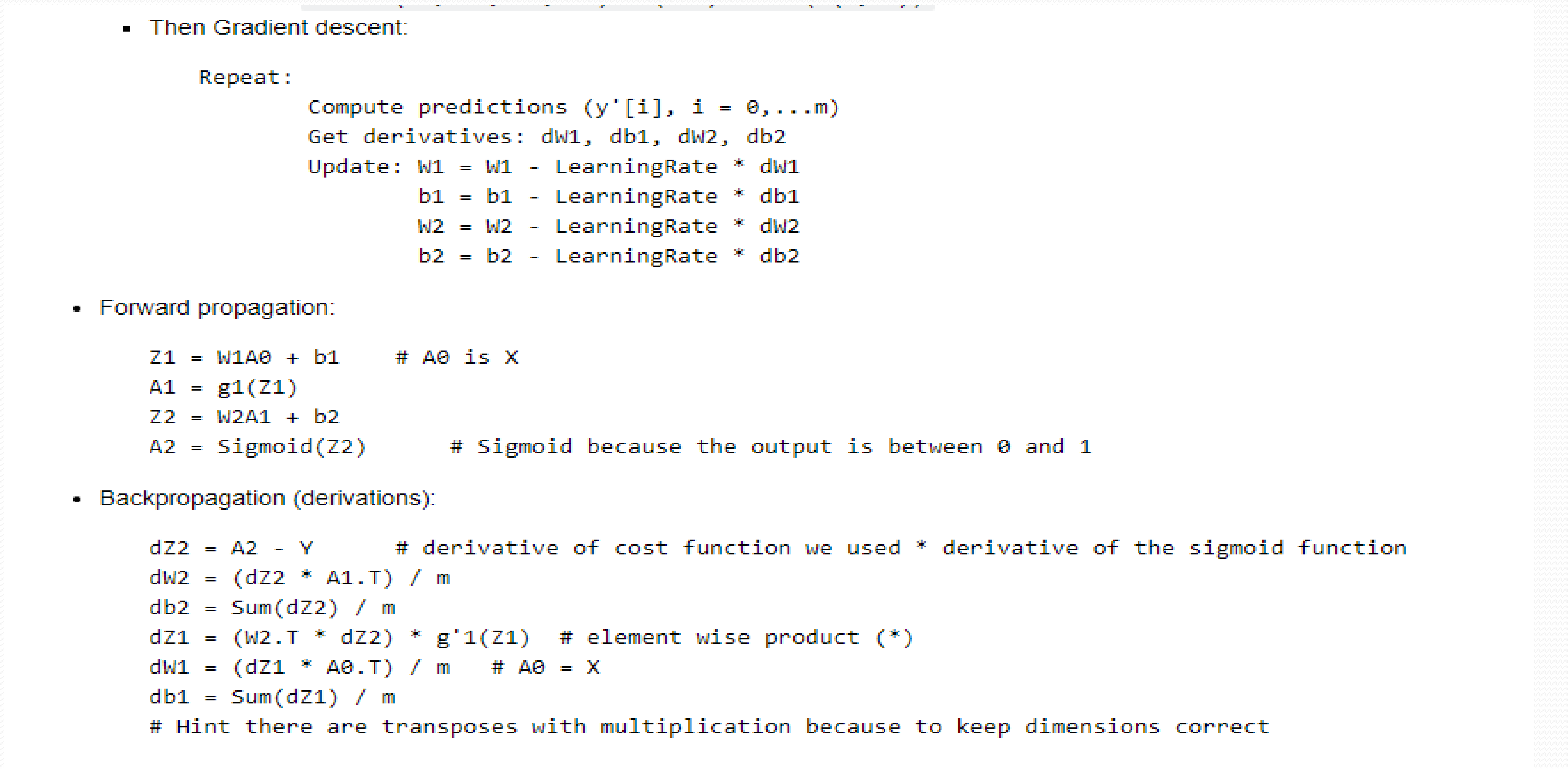
- Derivation of RELU activation function:

```
g(z)  = np.maximum(0,z)
g'(z) = { 0  if z < 0
          1  if z >= 0  }
```

- Derivation of leaky RELU activation function:

```
g(z)  = np.maximum(0.01 * z, z)
g'(z) = { 0.01  if z < 0
          1     if z >= 0   }
```

Gradient descent for Neural Networks



- Then Gradient descent:

```
Repeat:
        Compute predictions (y'[i], i = 0,...m)
        Get derivatives: dW1, db1, dW2, db2
        Update: W1 = W1 - LearningRate * dW1
                b1 = b1 - LearningRate * db1
                W2 = W2 - LearningRate * dW2
                b2 = b2 - LearningRate * db2
```

- Forward propagation:

```
Z1 = W1A0 + b1    # A0 is X
A1 = g1(Z1)
Z2 = W2A1 + b2
A2 = Sigmoid(Z2)      # Sigmoid because the output is between 0 and 1
```

- Backpropagation (derivations):

```
dZ2 = A2 - Y      # derivative of cost function we used * derivative of the sigmoid function
dW2 = (dZ2 * A1.T) / m
db2 = Sum(dZ2) / m
dZ1 = (W2.T * dZ2) * g'1(Z1)  # element wise product (*)
dW1 = (dZ1 * A0.T) / m   # A0 = X
db1 = Sum(dZ1) / m
# Hint there are transposes with multiplication because to keep dimensions correct
```

- Let’s run the notebook

- **Planar data classification with one hidden layer**