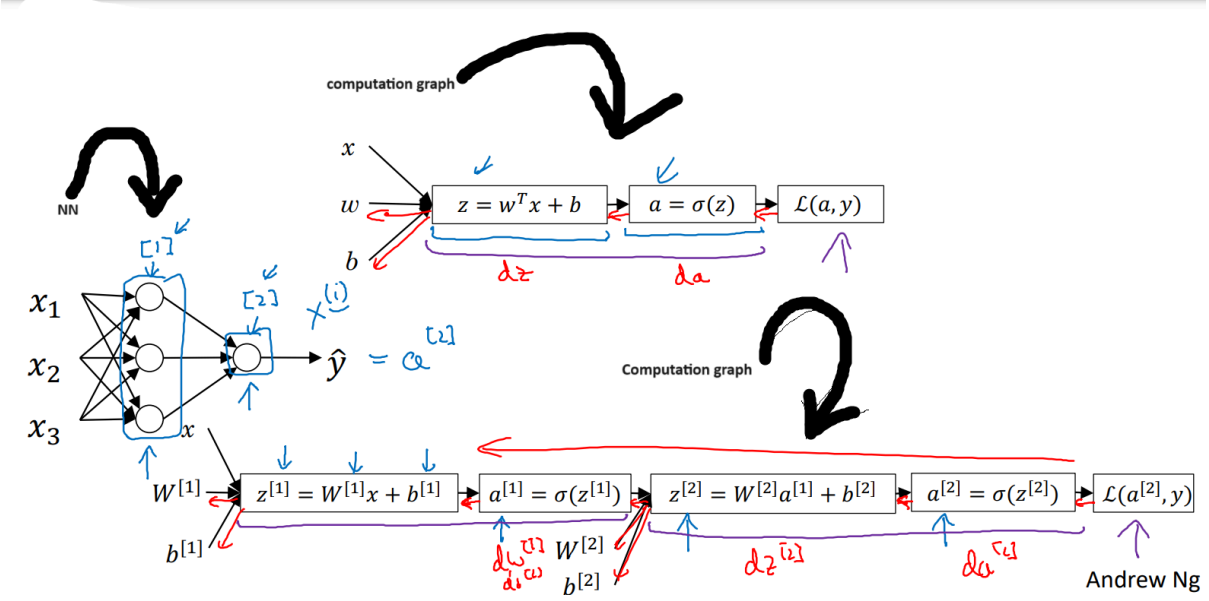
Week 3

Neural network

In the below figure it is shown how Neural networks take input for the hidden layers and the representations of the variables w, b, a, x for the diff layers;



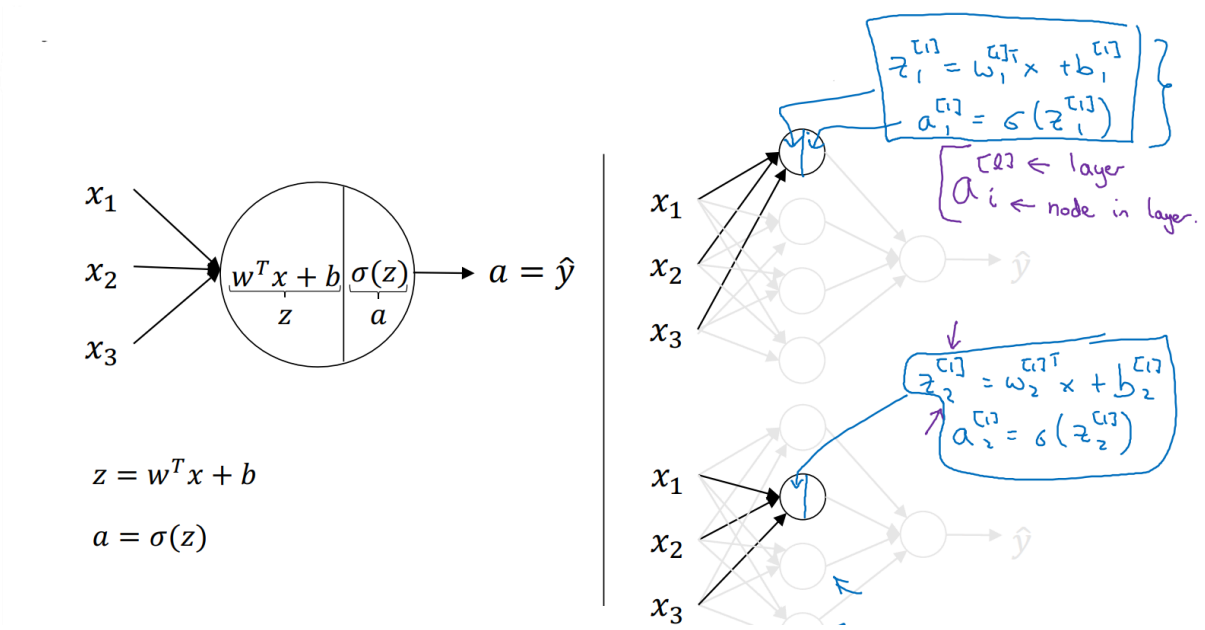

In the NN in this fig, the hidden layers and the output layer form the layer-1 and 2 respectively.This NN is aka 2 layer NN.The  correct formulae of z,a,etc. are given in later part

The inputs x1,x2,x3 are altogether taken and denoted by x.
In the bottommost computation graph, x, w[1], b[1] from the layer 1 compute z[1] and a[1] which then compute z[2] and a[2].
a[1] is a linear vector of a_1[1],a_2[1],a_3[1], for the NN in the fig. And its dimension are (3,1). The input layer has a[0]=x , 0 is subscript for input layer.

Also, w[1] and b[1] has dimension ,(4,3) and (4,1) resp. Whereas w[2] and b[2] have (1,4) and (1,1). The 4 in w[1] is no of neurons in

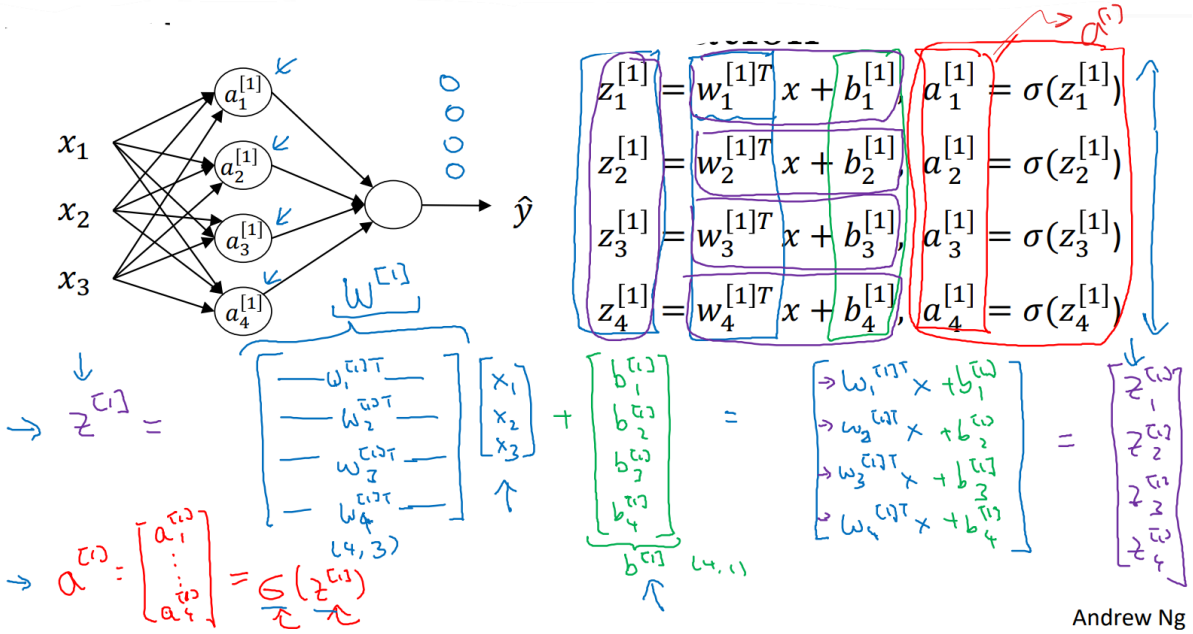
layer 1, and 3 is for the Neurons in previous layer, lly, for the other 3 w[2],b[1],b[2]
As learnt in logistic regressions, given x,w and b as input for computing z,a, and loss; this is NOT applicable for 2 layer NN.
Superscripts are vvi in NN

## Vectorisation and neural network

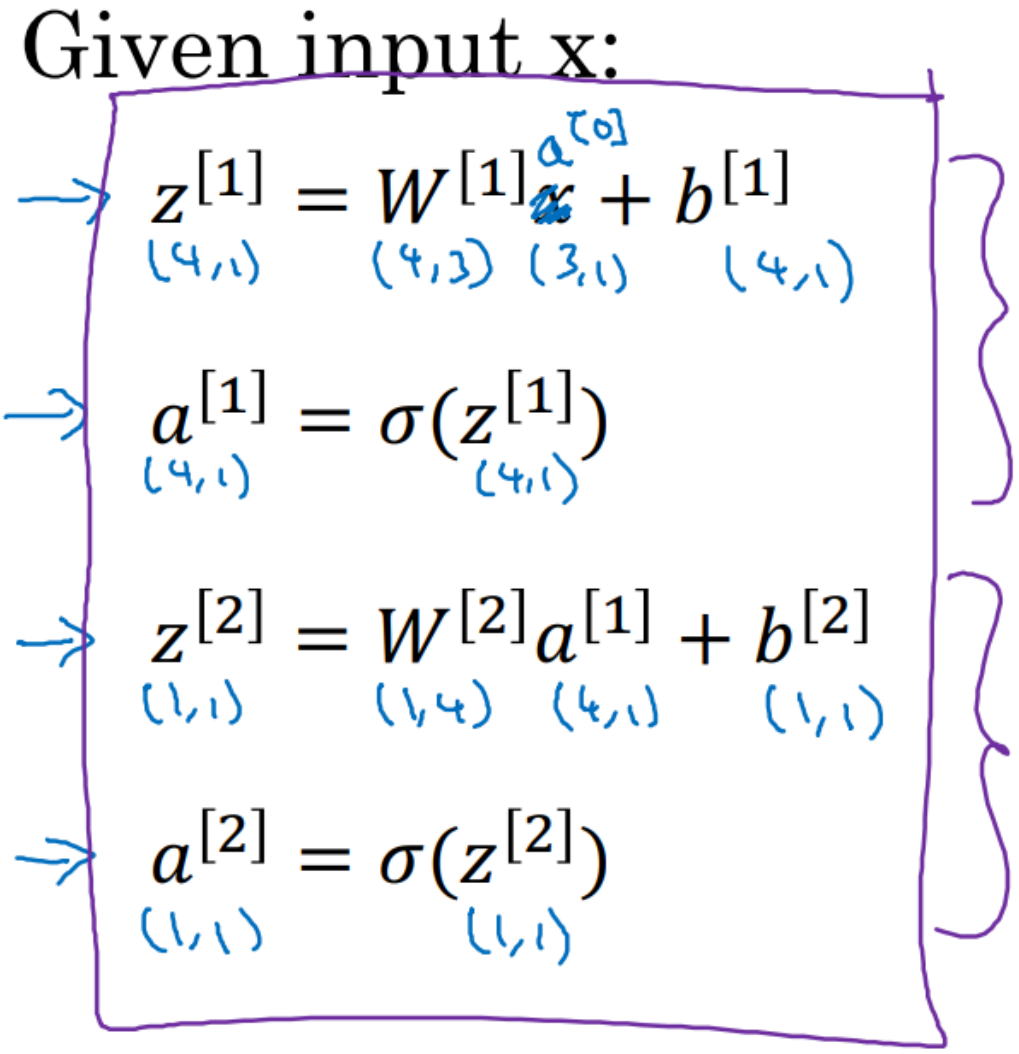



Each neuron  in hidden layer is divided in 2 parts - calculate z and a

formulas for z and a at each neuron is given in foll fig along with how vectorisation is carried out is also shown




Here, vectorization Is preferred as adding each formula is not feasible. For vectorization we need z vector,
And for that we need vectors of each of the variable present in formula of z ,i.e., wT, x, b
Now, trace the formula in vector form at bottom part Backwards, the Z vector is a column mx which equals wT*x+b matrix with superscript 1
wT*x+b with subscript 1 is Obtained from sum of wT*x and b matrices [ superscript 1 ]

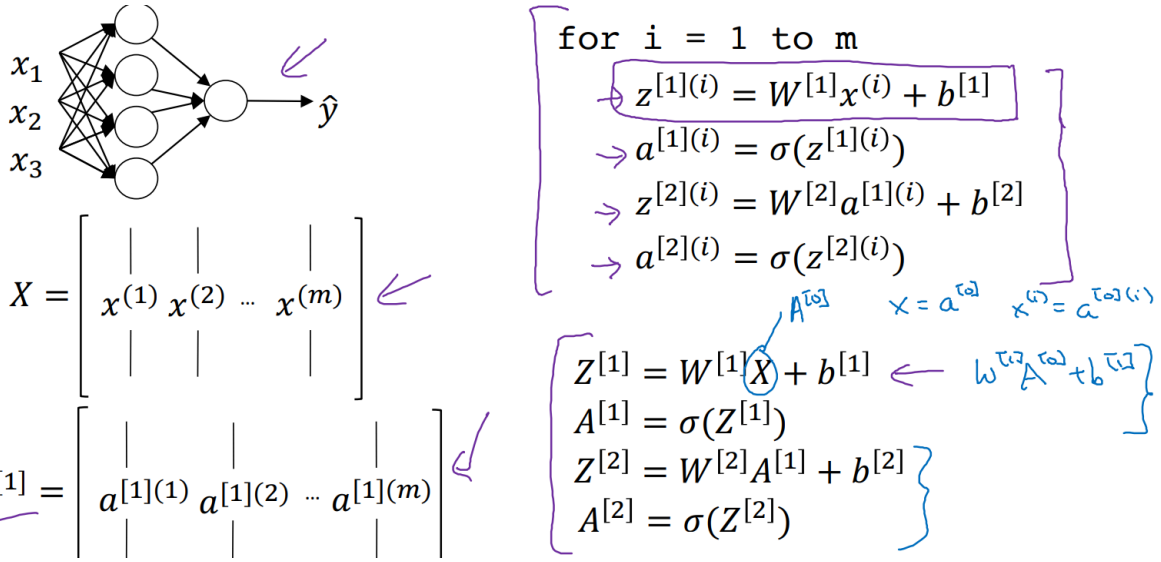



This pic gives the dimensions of each matrix present in the different formulas
Also x is replaced by a[0] cus they are same

**Vectorisation over multiple examples ,i.e., m training egs**

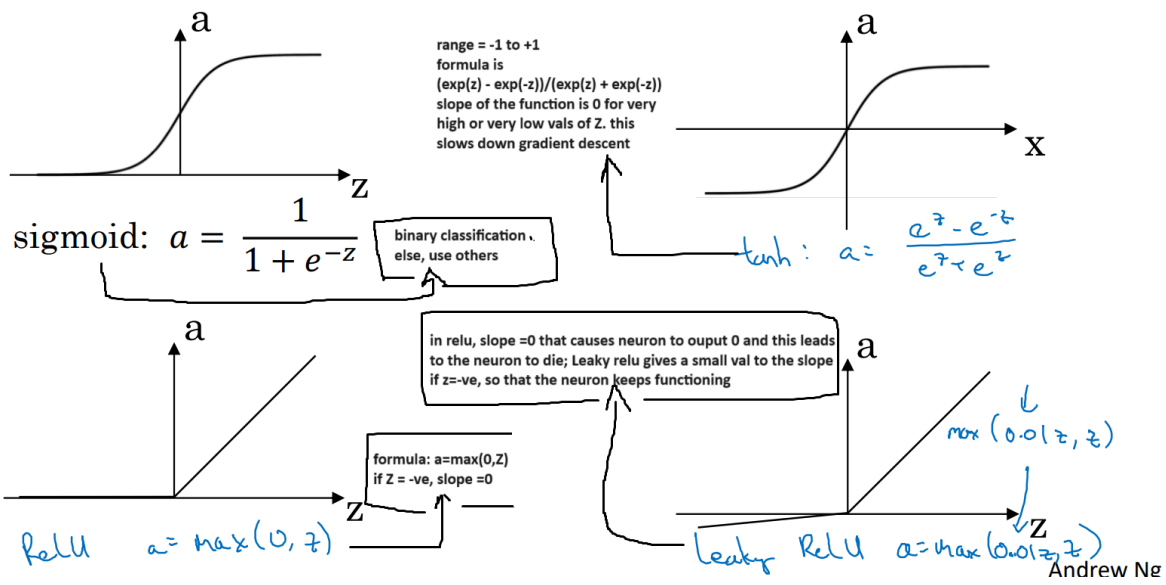
The figure below Give the for loop that Calculates the various variables for m examples
the X Matrix with order (n,m) gives a z matrix with the same order after formula of z applied
Similarly A matrix is obtained from its formula and is shown in the fig . this A matrix is an Activation function matrix wherein

horizontally training examples are obtained and as we go down vertically we get to m training eg, so , The 1st member of this matrix gives activation function for the first hidden unit of the 1st training eg




Not the superscripts of each LHS in the for loop, i is for egs and ranges from 1 to m
The main point of this image - instead of the for loop, the vectorisation formulas present below it are used

## Activation functions



Formula for leaky relu is a = max ( 0.01 * Z, Z )

**Why do we need activation functions**

Without activation functions there will be linearity and
linear functions output real numbers but we want binary form
Also there will be no concept of hidden layers

Activation functions are not needed in neural networks that output real numbers like housing prices

## Derivations of activation function

-G Prime [ **g'(Z) ] :**
g prime equals the slope of the activation function
its the differentiation or rate of change of g ; g(Z) = a
here prime for the dash present on the g
We require the slope aka g prime for gradient descent

Direct formula for g'(Z) and g(Z);

1. **Sigmoid activation function:**

   **g(z) = 1 / (1 + exp(-z))**

   **g'(z) = g(z) * (1 - g(z))**
2. **Tanh activation function:**

   **g(z) = (exp(z) - exp(-z)) / (exp(z) + exp(-z))**

   **g'(z) = 1 - g(z)^2**
3. **ReLU (Rectified Linear Unit) activation function:**

   **g(z) = max(0, z)**

   **g'(z) = 0 if z < 0, 1 if z > 0**
4. **Leaky ReLU activation function:**

   **g(z) = max(0.01z, z)**

**g'(z) = 0.01 if z < 0, 1 if z > 0**

Vals of g prime lie *between* 0 and 1

*Recap formula-*
Gradient Descent Update/ Formulas for implementing gradient descent

W1 = W1 - learning_rate * dW1
B1 = B1 - learning_rate * dB1
W2 = W2 - learning_rate * dW2
B2 = B2 - learning_rate * dB2

The numbers below are the superscript
Forward Propagation:

1. Z1 = W1X + B1
2. A1 = activation_function(Z1)
3. Z2 = W2A1 + B2
4. A2 = G2(Z2)

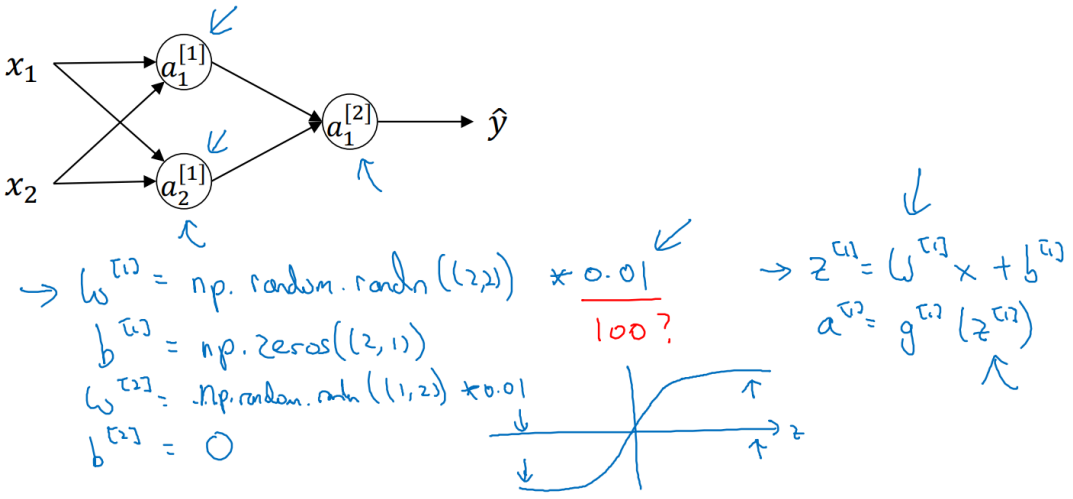
Backpropagation:

1. dZ2 = A2 - Y
2. dW2 = (1/m) * dZ2 * A1^T
3. dB2 = (1/m) * np.sum(dZ2, axis=1, keepdims=True)
4. dZ1 = W2^T * dZ2 * G1_prime(Z1)
5. dW1 = (1/m) * dZ1 * X^T
6. dB1 = (1/m) * np.sum(dZ1, axis=1, keepdims=True)

## Random initialisations of weights and b

on initialising w or weight and b with 0 ,the output that any neuron from Any layer gives is the same which is technically wrong
outputs from the neurones must be different ideally

For resolving this random initialisations are necessary
Note Initialising b with zero has no problem. it is w.




## So the formulae are

w = np.random.randn(shape) * 0.01
b = np.zeros(shape)
b2 =0 , as The values of b Must be different ,It is not necessary to keep any one of them as 0
The .01 constant is to hav a small val of w, as larger vals cause problem in Calculating Gradient descent which is slower if slope is small,
Slope is small if z is large which will be ofcourse if we have larger w