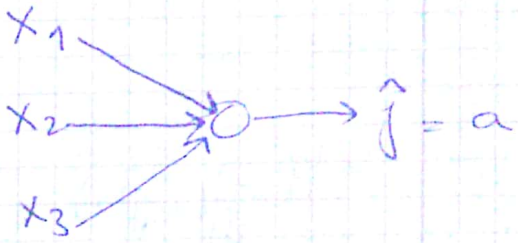
Week # 3:-

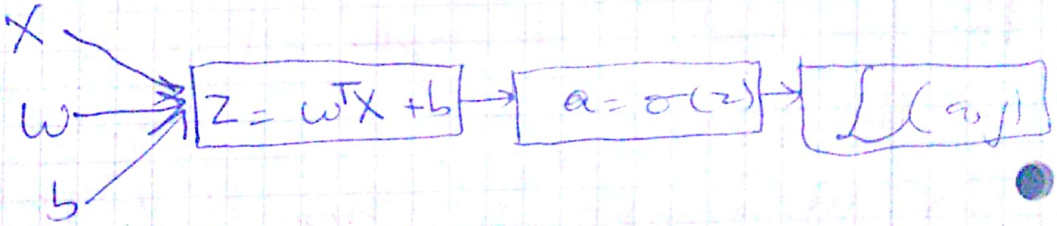
Shallow Neural Network:-

Neural network overview:-

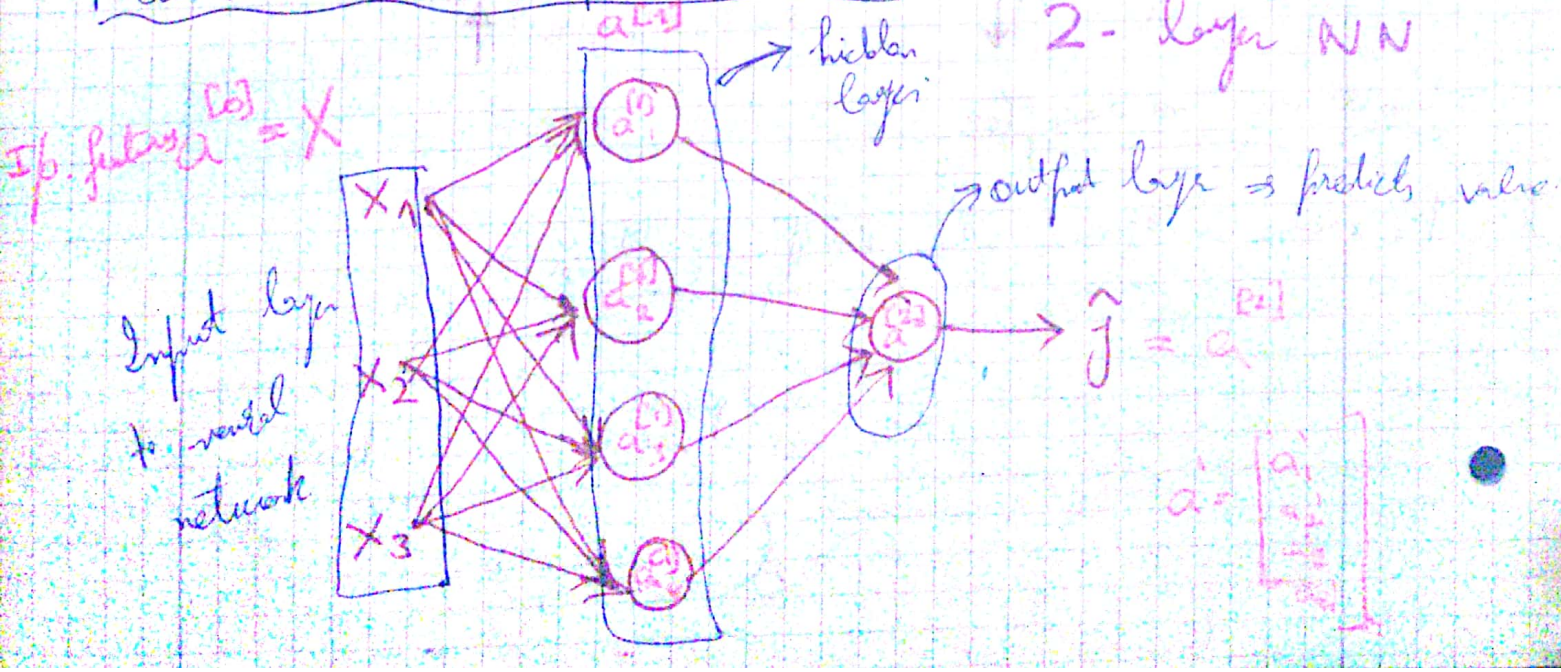
What is neural network



Neural network can be formed by stacking together a lot of sigma-id units



Neural Network Representation:-



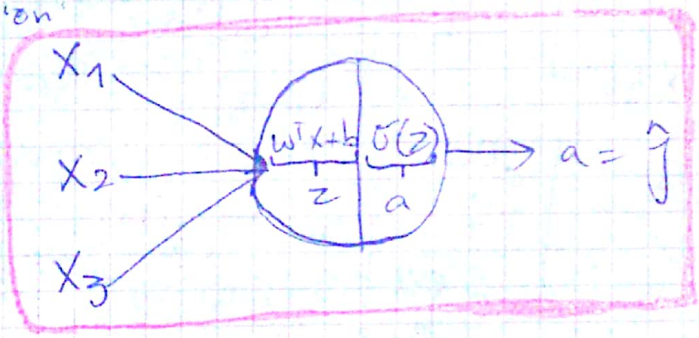
$W^{[1]}$ here will be 4×2 matrix. 4 comes from the fact that we have 4 units inside the hidden layers and 3 comes from the facts that we have 3 features.

$b^{[1]} \in 4 \times 1$

$W^{[2]} \in \mathbb{R}^{1 \times 4}$ $\quad b^{[2]} \in \mathbb{R}^{1 \times 1}$

Computing a Neural Network output :-

- Now let's recall we said logistic regression consists of 2 steps of computation first you compute z then you compute sigmoid as activation function.



- In the last 2 layered network, let's look at the first node first layer.

Same as above figure it will compute:

$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$ then $a_1^{[1]} = \sigma(z_1^{[1]})$ ⇒ activation.

So Convention here is

$a_i^{[l]}$ ← layer

↖ node in layer

- Second node in first layer.

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

3rd node 1st layer

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

4th node 1st layer

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]} \; , \; a_{61}^{[1]} = \sigma(z_4^{[1]})$$

So now as we have finished our first layer computation now to implement it we will not use for loops we will use vectorization

→ stack w's in a matrix.

$$\vec{z}^{[1]} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]} x + b_2^{[1]} \\ w_3^{[1]} x + b_3^{[1]} \\ w_4^{[1]} x \times b_4^{[1]} \end{bmatrix} = \begin{bmatrix} — \; w_1^{[1]T} \; — \\ — \; w_2^{[1]T} \; — \\ — \; w_3^{[1]T} \; — \\ — \; w_4^{[1]T} \; — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

4×3

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

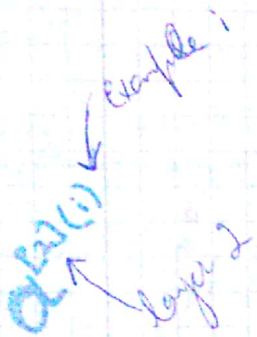
→ Now for the second layer

→ $z^{[2]} = \underset{(1\times4)}{w^{[2]}} \; \underset{(4\times1)}{a^{[1]}} + \underset{(1\times1)}{b^{[2]}}$

→ $\underset{(1,1)}{a^{[2]}} = \sigma(\underset{(1,1)}{z^{[2]}})$

Vectorizing across multiple examples:

- Let's see how to vectorize across multiple training examples.



$$X \longrightarrow a^{[2]} = \hat{y}$$

$$x^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$$

$$x^{(2)} \longrightarrow a^{[2](2)} = \hat{y}^{(2)}$$

$$\vdots$$

$$x^{(m)} \longrightarrow a^{[2](m)} = \hat{y}^{(m)}$$

- ✱ for i=1 to m:

$$z^{[1](i)} = w^{[1]} x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

$$z^{[2](i)} = w^{[2]} a^{[1](i)} + b^{[2]}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

- We want to vectorize the whole computation in order to get rid of the main loop ✱

→ So Recall, our training examples stoked in a matrix like this

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$(n_x \times m)$

Now the vectorized version $X = a^{[0]}$; $X^{(1)} = a^{[0](1)}$

$A^{[0]}$

$\rightarrow Z^{[1]} = W^{[1]} X + b^{[1]} \leftarrow W^{[1]} A^{[0]} + b^{[1]}$

$\rightarrow A^{[1]} = \sigma(Z^{[1]})$

$\rightarrow Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$

$\rightarrow A^{[2]} = \sigma(Z^{[2]})$

training examples →

$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$; $A^{[1]} = \begin{bmatrix} | & & | \\ a^{[1](1)} & \cdots & a^{[1](m)} \\ | & & | \end{bmatrix}$

↓ hidden units

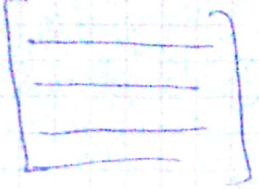
- Corresponds to the activation value of 1st hidden unit on first training example in 1st layer
- Corresponds to the activation value of 2nd hidden unit on first training example in 1st layer
- activation value ⇒ 3rd hidden unit, 1st layer (hidden) 1st training example

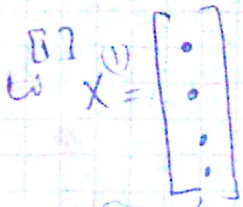
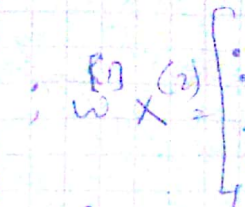
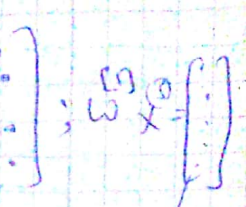
Explanation for vectorized implementation:-

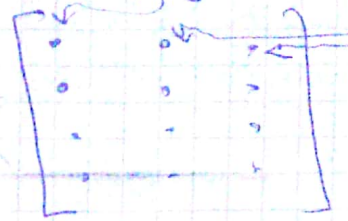
Let's say we have 3 training examples then computation would be

$Z^{[1](1)} = W^{[1]} x^{(1)} + b^{[1]}$, $Z^{[1](2)} = W^{[1]} X^{(2)} + b^{[1]}$, $Z^{[1](3)} = W^{[1]} x^{(3)} + b^{[1]}$

for the sake of simplicity let's say that all b's are zero

So $W^{[1]}$ = 

$W^{[1]} X^{(1)}$ =  ; $W^{[1]} X^{(2)}$ =  ; $W^{[1]} X^{(3)}$ = 

$W^{[1]}$  = 

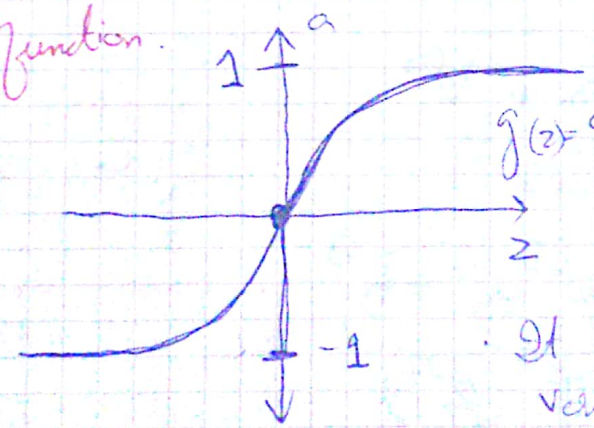
$= \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = Z^{[1]}$

$W^{[1]} X^{(i)} = Z^{[1](i)}$

Activation functions:-

So far, we have been using sigmoid activation function, now we look other options too.

→ other function that we can use is tangent hyperbolic function.



$g(z) = a = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$

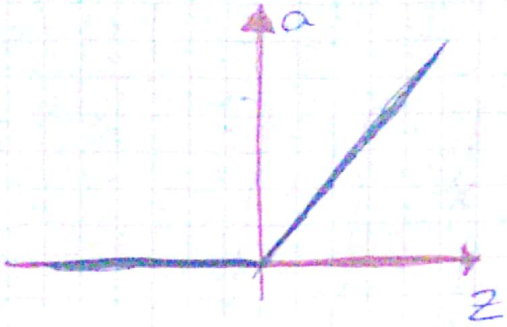
· It is the shifted version of our sigmoid function.

(a) "tanh" works better (always) with the exception of the output layer where you want the probability to be between 0 and 1.

So in 2 layered NN $g^{[1]}(z^{[1]}) = \tanh(z^{[1]})$

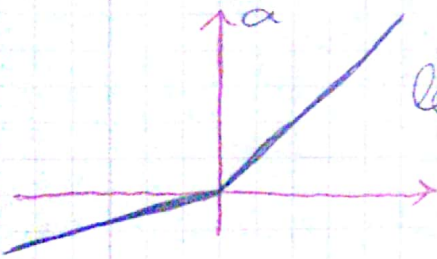
$g^{[2]}(z^{[2]}) = \sigma(z^{[2]})$.

One other function is ReLU (Rectified linear unit function)



$a = \max(0, z)$

One disadvantage of Relu is that, it has derivative zero with z being less than zero.



Leaky Relu $\Rightarrow a = \max(0.01z, z)$

→ Why do you need a non-linear activation functions?

From the algorithm that we established earlier

Given X:

1. $Z^{[1]} = W^{[1]} X + b^{[1]}$
2. $a^{[1]} = g^{[1]}(Z^{[1]})$ $\quad g(z) = z$
3. $Z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$
4. $a^{[2]} = g^{[2]}(Z^{[2]})$

Let's think for a moment that the function $g(z)$ is the Identity function that is: whatever goes in comes out

Then:

$$a^{[1]} = z^{[1]} = w^{[1]} x + b^{[1]}$$

$$a^{[2]} = z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = w^{[2]} \underbrace{\left(w^{[1]} x + b^{[1]} \right)}_{a^{[1]}} + b^{[2]}$$

$$= \underbrace{\left(w^{[2]} w^{[1]} \right)}_{w'} x + \underbrace{\left(w^{[2]} b^{[1]} + b^{[2]} \right)}_{b'}$$

$$\boxed{a^{[2]} = w' x + b'}$$

- If we were to use Identity activation function then our neural network will be using linear function and outputting it i.e. linear output

→ So later when we will use network with many hidden layers, then with identity activation function we will be outputting linear function's output and it's like not using a network with with many layers.

So linear hidden layer is more or less useless.

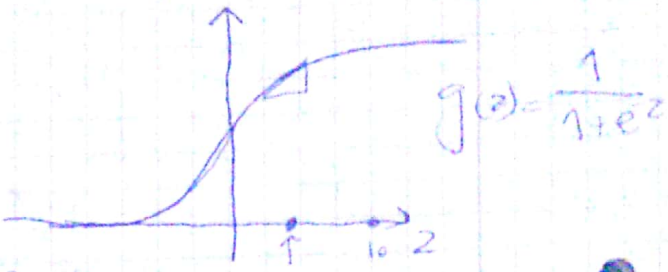
→ There is one place, where you might use linear activation function $g(z) = z$ and that is if you are doing machine learning on a regression problem. I.e. if $y \in \mathbb{R}$. not only from 0 to 1

- In this case we use Identity or linear activation function only in the output layer and hidden layers still use Relu or tanh.

Derivatives of activation functions:-

When you implement back propagation you compute derivatives of activation functions, so let's take a look at our choices of activation functions and how you can compute the slopes of these functions.

Let's see sigmoid function:



$\frac{d}{dz}(g(z))$ = slope of $g(z)$ at z

let's take a look at this point

So lets take derivative

$$g'(z) \quad \frac{d}{dz}\left[\frac{1}{1+e^{-z}}\right] = \frac{-1(-e^{-z})}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right)$$

→ look at the power rule of calculus

So $\frac{d}{dz} g(z) = g(z)(1 - g(z)) \Rightarrow a(1-a)$

Let's take a look at $z = 10 \Rightarrow g(z) \approx 1$

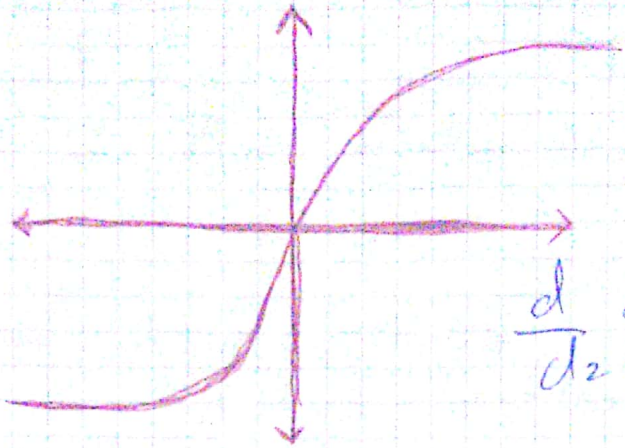
$$\frac{d}{dz} g(z) \approx 1(1-1) \approx 0$$

and its clear from the figure that as z increases rate of change decreases

⇒ Conversely if $z = -10$ $\quad g(z) \approx 0$

$$\frac{d}{dz} g(z) \approx 0 \cdot (1-0) \approx 0$$

Let's now look at the tanh activation function



$g(z) = \tanh(z)$

$\frac{d}{dz} g(z)$ = slope of $g(z)$ at z

$z = 10 \quad \tanh(z) \approx 1$

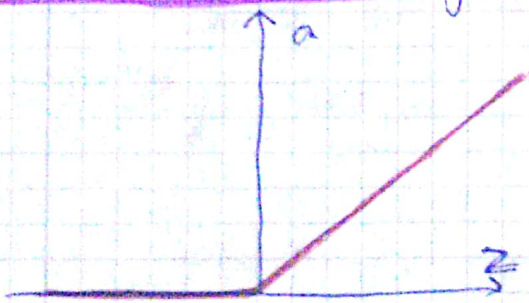
$g'(z) \approx 0$

$z = -10 \quad \tanh(z) \approx -1$

$g'(z) \approx 0$

$g'(z) = \frac{d}{dz} g(z) = 1 - (\tanh(z))^2$

ReLU and Leaky ReLU

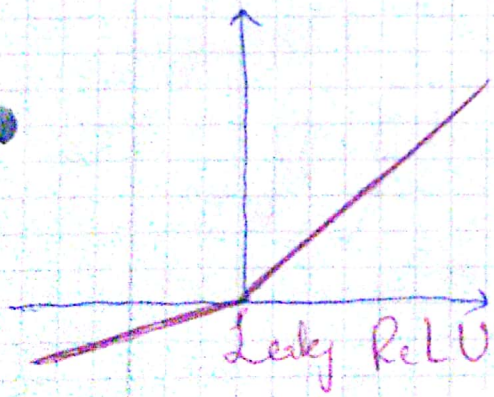


$g(z) = \max(0, z)$

$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$

undefined if $z = 0$

but when implementing in software at zero we take $g'(z) = 1$ or 0 doesn't matter.



$g(z) = \max(0.01z, z)$

$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{cases}$

* Probability of $g(z)$ outputting 0.000000 is very low.

Gradient Descent for Neural network :-

Let's dive into, how to implement gradient descent on neural network with one hidden layer.

⇒ Neural network with single hidden layer has parameter $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$

→ If you have n_x or alternatively $n^{[0]}$ input features and $n^{[1]}$ hidden units and $n^{[2]}$ output units

then $w^{[1]} \in \mathbb{R}^{(n^{[1]} \times n^{[0]})}$ $\quad w^{[2]} \in \mathbb{R}^{(n^{[2]} \times n^{[1]})}$

$b^{[1]} \in \mathbb{R}^{(n^{[1]} \times 1)}$ $\quad b^{[2]} \in \mathbb{R}^{(n^{[2]} \times 1)}$

Assuming we are doing binary classification.

Cost function is : $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]})$

$$= \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y)$$

($\hat{y} = a^{[2]}$)

Now to train the parameters, we need to implement gradient descent.

Repeat {

Compute predictions $(\hat{y}^{(i)}, i = 1, \ldots m)$.

$dw^{[1]} = \frac{\partial J}{\partial w^{[1]}}$, $db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$

$w^{[1]} = w^{[1]} - \alpha\, dw^{[1]}$

$b^{[1]} = b^{[1]} - \alpha\, db^{[1]}$ $\quad \alpha :=$ learning rate

Forward propagation:

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$$

Backward propagation:

$$dZ^{[2]} = A^{[2]} - Y \qquad Y = [y^{(1)}, y^{(2)}, \dots y^{(m)}]$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis=1, keepdims=True)$$

$$dZ^{[1]} = \underbrace{W^{[2]T} dZ^{[2]}}_{\in (n^{[1]} \times m)} * g^{[1]\prime}(Z^{[1]}) \in (n^{[1]}, m)$$

(* : element-wise product)

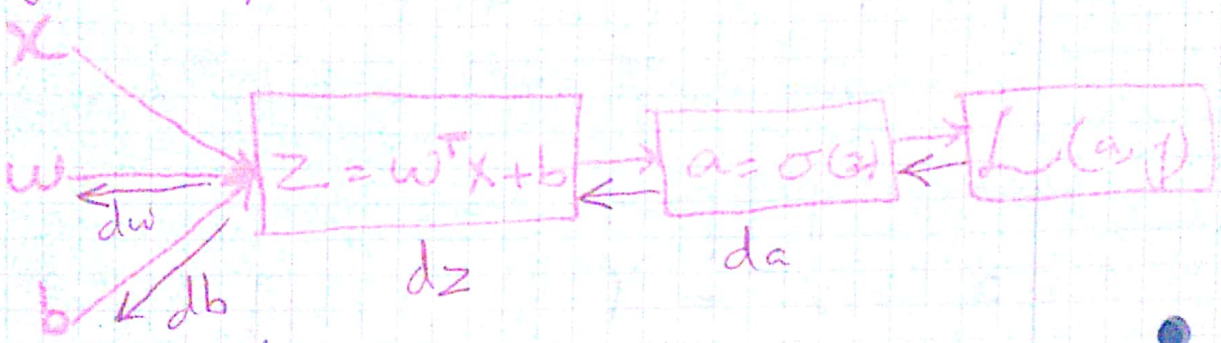
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis=1, keepdims=True)$$

Back propagation Intuition:-

Let's see how we derived the equation on the last page we wrote

Logistic Regression:



$$L(a, y) = -y \log(a) - (1-y) \log(1-a)$$

$$\frac{d}{da} L(a, y) = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$dz = a - y$$

$$dz = da \cdot g'(z)$$

$$g'(z) = \sigma(z)$$

$$dw = dz \cdot x$$

$$db = dz$$

by chain rule

$$\frac{dL}{dz} = \frac{dL}{da} \cdot \left(\frac{da}{dz}\right), \quad \frac{da}{dz} = g'(z)$$

$$= \left(\frac{-y}{a} + \frac{1-y}{1-a}\right) a(1-a)$$

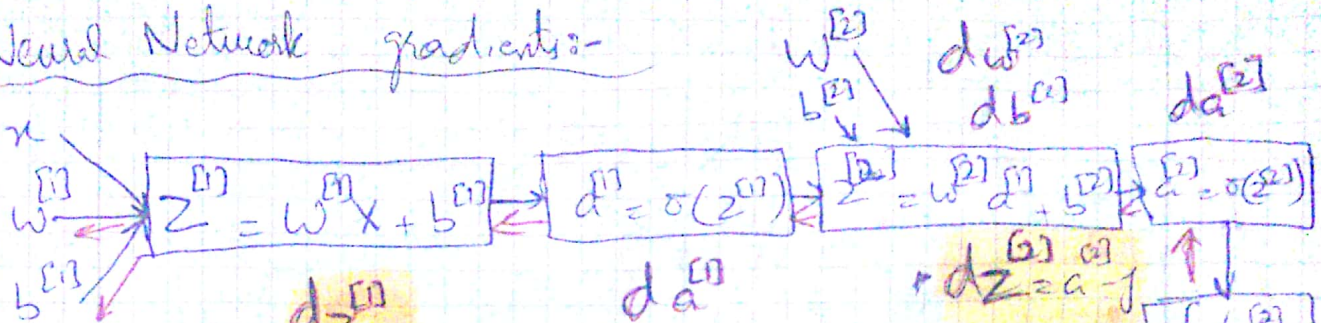
$$= -y(1-a) + (1-y)a$$

$$= -y + ay + a - ay$$

$$\frac{dL}{dz} = a - y = dz$$

This is the computation for just one step now we are going to do it with the neural network we have defined earlier with one hidden layer that means we are going to do these computation twice.

Neural Network gradients:-



$dz^{[2]} = a^{[2]} - y$

$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$

$da^{[1]}$

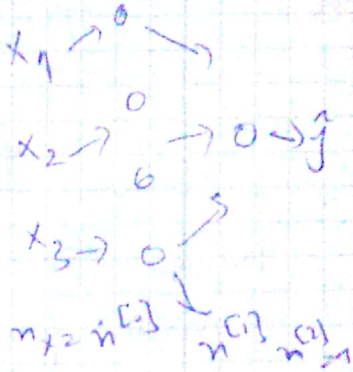
$dW^{[2]} = dz^{[2]} a^{[1]T}$

$db^{[2]} = dz^{[2]}$

"$dw = dz \cdot x$"

$W^{[2]}$ $(n^{[2]}, n^{[1]})$

$z^{[2]}, dz^{[2]}$ $(n^{[2]}, 1)$ $\rightarrow (1, 1)$



$dz^{[1]} = W^{[1]T} dz^{[1]} * g^{[1]'}(z^{[1]})$

$dW^{[1]} = dz^{[1]} \cdot x^T$ ← $a^{[0]T}$

$db^{[1]} = dz^{[1]}$

So summery of gradient descent:

$dz^{[2]} = a^{[2]} - y$

$dW^{[2]} = dz^{[2]} a^{[1]T}$

$db^{[2]} = dz^{[2]}$

$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$

$dW^{[1]} = dz^{[1]} x^T$

$db^{[1]} = dz^{[1]}$

W.r.t X we don't need to take derivatives because right now we are talking a problem of supervised learning and in these kind of problem input features remain same for particular example.

Vectorized implementations

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$$

$$Z^{[1]} = W^{[1]}X + b^{[1]} \quad \text{forward step.}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

Now the same trick can be implemented on the back propagation step.

Vectorized back prop:-

$$dz^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis=1, keepdim=True)$$

$$dZ^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]}) \quad \text{(elementwise product)}$$

$$dW^{[1]} = \frac{1}{m} dz^{[1]} X^{T}$$

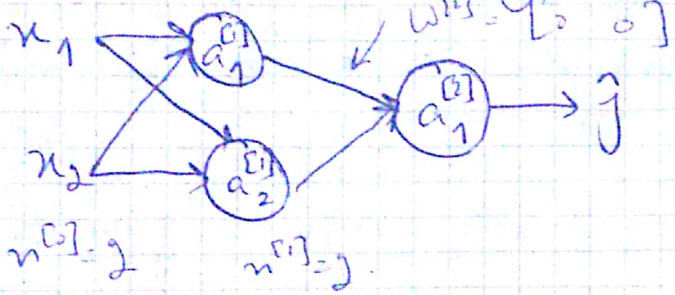
$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis=1, keepdim=True)$$

Initialization of parameters :-

- It is important to initialize your parameters randomly.

→ for neural network if we initialize all parameters to zero and then apply gradient descent, it won't work. let's find out why.

Let's consider the following.



* If we initialize weights to zero, each neuron in the first hidden layer will perform the same computation. So, even after multiple iterations each neuron in the layer will compute the same thing

$$W^1 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

turns out initializing bias term to zero is Ok!
but initializing W to zero creates problems.

because:

$$a_1^{[1]} = a_2^{[1]}$$ so in backpropagation $$dz_1^{[1]} = dz_2^{[1]}$$

So by symmetry $$dW = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$$

means $W = \begin{bmatrix} — \\ — \end{bmatrix}$ both will be same.

$$W^{[1]} = W^{[1]} - \alpha dW$$

as both are computing the same thing so there is no point in having multiple layers.

So solution to this is to initialize your parameters randomly.

$W^{[1]}$ = np.random.randn((2,2)) * 0.01

$b^{[1]}$ = np.zeros((2,1))

$W^{[2]}$ = initialize randomly

$b^{[2]}$ = 0

This is not very large because if 'z' is large then input to activation function will be large so derivative will be small and optimization algorithm will be slow

Backward Propagation Explained:

Using Chain rule to derive the derivatives of Ith layer in backward propagation.

* Courtesy tutor: Chandrashekar R

The Cost function for 1 data example is given as:

$$L(a,y) = -y\log(a) - (1-y)\log(1-a)$$

Total cost for all training examples:-

$$J(W^{[L]}, b^{[L]}, W^{[L-1]}, b^{[L-1]}, \ldots\ldots, W^{[1]}, b^{[1]}) = \frac{1}{m} \times \Sigma\left[L(a^{(i)}, y^{(i)})\right]$$

Please note that the cost J is a function of all parameters in all layers (not just the output layer) - i.e. $W^1, b^1, \ldots W^l, b^l$ $\forall$ l = 1 .... m. Goal of the gradient descent is to find the values of each of these parameters using derivatives, apply to weights until it converges.

As the output has Sigmoid activation, the cost function for that is:

$$L(a,y) = -(y\log(a) + (1-y)\log(1-a))$$

Using Chain rule of Calculus:-

$$\frac{dL}{dz^{[L]}} = \frac{dL}{da^{[L]}} * \frac{da^{[L]}}{dz^{[L]}}$$

$$\frac{dL}{da^{[L]}} = -\frac{y}{a^{[L]}} + \frac{(1-y)}{(1-a^{[L]})}$$

Simplifying this yields :-

$$\frac{dL}{da^{[L]}} = \left(a^{[L]} - y\right) / \left(a^{[L]}(1-a^{[L]})\right)$$

for sigmoid activation $a^{[L]} = \frac{1}{(1+e^{-z^{[L]}})}$

$$\frac{da^{[L]}}{dz^{[L]}} = -(1+e^{-z^{[L]}})^{-2} * (-1) * (e^{-z^{[L]}})$$

Simplifying, yields: $\boxed{\frac{da^{[L]}}{dz^{[L]}} = a^{[L]}(1-a^{[L]})}$

So Eqn (1)

$$\boxed{Dz^{[L]} = \frac{dL}{dz^{[L]}} = \frac{dL}{da^{[L]}} * \frac{da^{[L]}}{dz^{[L]}} = a^{[L]} - y}$$

$$\frac{dL}{dw^{[L]}} = \frac{dL}{dz^{[L]}} * \frac{dz^{[L]}}{dw^{[L]}}$$

$$z^{[L]} = w^{[L]} * a^{[L-1]} + b^{[L]}$$

$$\frac{dz^{[L]}}{dw^{[L]}} = a^{[L-1]}$$

$$\frac{dL}{dw^{[L]}} = a^{[L-1]} * Dz^{[L]} = a^{[L-1]} * (a^{[L]} - y)$$

$$\boxed{\frac{dL^{[L]}}{dw^{[L]}} = Dw^{[L]} = a^{[L-1]} Dz^{[L]}}$$

$$\frac{dL^{[L]}}{db^{[L]}} = \frac{dL}{dz^{[L]}} * \frac{dz^{[L]}}{db^{[L]}} \quad \because \frac{dz^{[L]}}{db^{[L]}} = 1$$

So:

$$\boxed{Db^{[L]} = \frac{dL}{db^{[L]}} = dz^{[L]}}$$

This is for the first layer. Let's consider the l^{th} layer.

The key part is to apply chain rule to get the derivative of l^{th} parameter in terms of $(l+1)^{th}$

So we apply chain rule for parameters in l^{th} layer going through $(l+1)^{th}$ layer.

Applying chain rule:-

$$\frac{dL}{dw^{[l]}} = \frac{dL}{da^{[L]}} * \frac{da^{[L]}}{dz^{[L]}} * \frac{dz^{[L]}}{dz^{[L-1]}} * \cdots\cdots \frac{dz^{[l+1]}}{dz^{[l]}} * \frac{dz^{[l]}}{dw}$$

Let $Dz^{[l+1]} = (dL/da^{[L]}) * (da^{[L]}/dz^{[L]}) * \left(\frac{dz^{[L]}}{dz^{[L-1]}}\right) * \cdots * \frac{dz^{[l+2]}}{dz^{[l+1]}}$

(as per notation)

$$z^{[L+1]} = w^{[l+1]} * a^{[l]} + b^{[l+1]}$$

Expanding for $a^{[l]}$

$$z^{[L+1]} = w^{[l+1]} * g^{[l]}(z^{[l]}) + b^{[l+1]}$$

By notation

$$D_z^{[l]} = \frac{dL}{da^{[L]}} * \frac{da^{[L]}}{dz^{[L]}} * \cdots * \frac{dz^{[l+2]}}{dz^{[l+1]}} * \frac{dz^{[l+1]}}{dz^{[l]}}$$

$$= D_z^{[l+1]} * \frac{dz^{[l+1]}}{dz^{[l]}}$$

$$= D_z^{[l+1]} * w^{[l+1]} * g'^{[l]}(z^{[l]})$$

$$D_z^{[l]} = D_z^{[l+1]} * w^{[l+1]} * g'^{[l]}(z^{[l]}) \quad \checkmark$$

- $z^{[l]} = w^{[l]} * a^{[l-1]} + b^{[l]}$

$$\frac{dz^{[l]}}{dw^{[l]}} = a^{[l-1]}$$

Therefore $Dw^{[l]} = \frac{dL}{dw^{[l]}} = D_z^{[l]} * \frac{dz^{[l]}}{dw^{[l]}}$

$$Dw^{[l]} = D_z^{[l]} * a^{[l-1]} \quad \checkmark$$

- $\frac{dL}{db^{[l]}} = \frac{dL}{dw^{[l]}} = \frac{dl}{da^{[L]}} * \frac{da^{[L]}}{dz^{[L]}} * \cdots * \frac{dz^{[l+2]}}{dz^{[l+1]}} * \frac{dz^{[l+1]}}{dz^{[l]}} * \frac{dz^{[l]}}{db^{[l]}}$

$$\frac{dz^{[l]}}{db^{[l]}} = 1$$

Therefore $Db^{[l]} = \frac{dL}{db^{[l]}} = D_z^{[l]}$

-

So, for the I^{th} layer we have the following equations:-

$$Dz^{[l]} = \frac{dL}{dz^{[l]}} = D_z^{[l+1]} * W^{[l+1]} \circledast g'^{[l]}(z^{[l]})$$

(⊛ : element wise)

$$D\,w^{[l]} = \frac{dL}{dw^{[l]}} = D_z^{[l]} * \frac{dz^{[l]}}{dw^{[l]}}$$

$$Db^{[l]} = \frac{dL}{db^{[l]}} = Dz^{[l]}$$