DEEP LEARNING

Deep Neural Networks

Data and Tasks
What kind of data and tasks have DNNs been used for?

One common example is digit classification using the MNIST dataset

MNIST images are vectorized using the pixel values of each cell
Matrix having pixel values will be of size 28x28 (as MNIST images are of the size 28x28)

255	183	95	8	93	196	253
254	154	37	...	28	172	254
252	221	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	198	253
252	250	187	178	195	253	253

Each pixel can range from 0 to 255. Standardize pixel values by dividing with 255

1	0.72	0.37	0.03	0.36	0.77	0.99
1	0.60	0.14	...	0.11	0.67	1
0.99	0.87	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	0.78	0.99
0.99	0.98	0.73	0.69	0.76	0.99	0.99

Now, flatten the matrix to convert into a vector of 784 (28x28)

Convert all images to vectors of order $\mathbb{R}784$

Let's look at the data along with the labels (Multi-Class Classification)

28x28 images	Vectorized form	Class Label	Class Labels - One Hot Representation
	[1.00, 0.72, … 0.99]	0	[1,0,0,0,0,0,0,0,0,0]
	[1.00, 0.85, … 1.00]	1	[0,1,0,0,0,0,0,0,0,0]
	[1.00, 0.76, … 1.00]	2	[0,0,1,0,0,0,0,0,0,0]
	[0.99, 0.82, … 1.00]	3	[0,0,0,1,0,0,0,0,0,0]
	[0.73, 0.81, … 0.67]	4	[0,0,0,0,1,0,0,0,0,0]
	[1.00, 1.00, … 0.99]	5	[0,0,0,0,0,1,0,0,0,0]
	[0.84, 0.72, … 0.99]	6	[0,0,0,0,0,0,1,0,0,0]
	[0.33, 0.52, … 1.00]	7	[0,0,0,0,0,0,0,1,0,0]
	[0.85, 0.72, … 0.99]	8	[0,0,0,0,0,0,0,0,1,0]
	[0.84, 0.92, … 0.99]	9	[0,0,0,0,0,0,0,0,0,1]

Another example would be the Indian Liver Patient classification problem. There are only two possible outcomes, hence it is a Binary-Class classification task

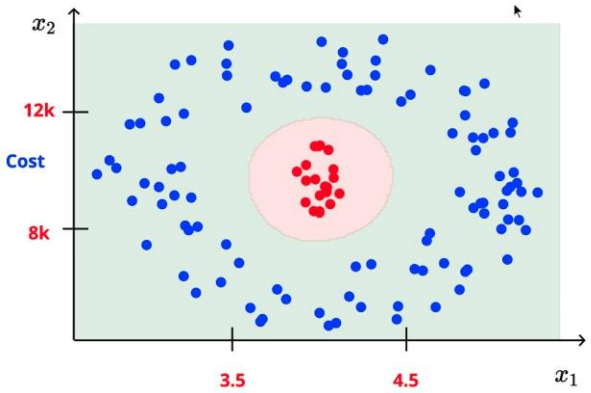
An example for regression would be Housing Price Prediction, where instead of predicting a discrete output, the prediction is a real-number or continuous value (decimals, fractions etc)

Model

A Simple Deep Neural Network

How to build complex functions using Deep Neural Networks

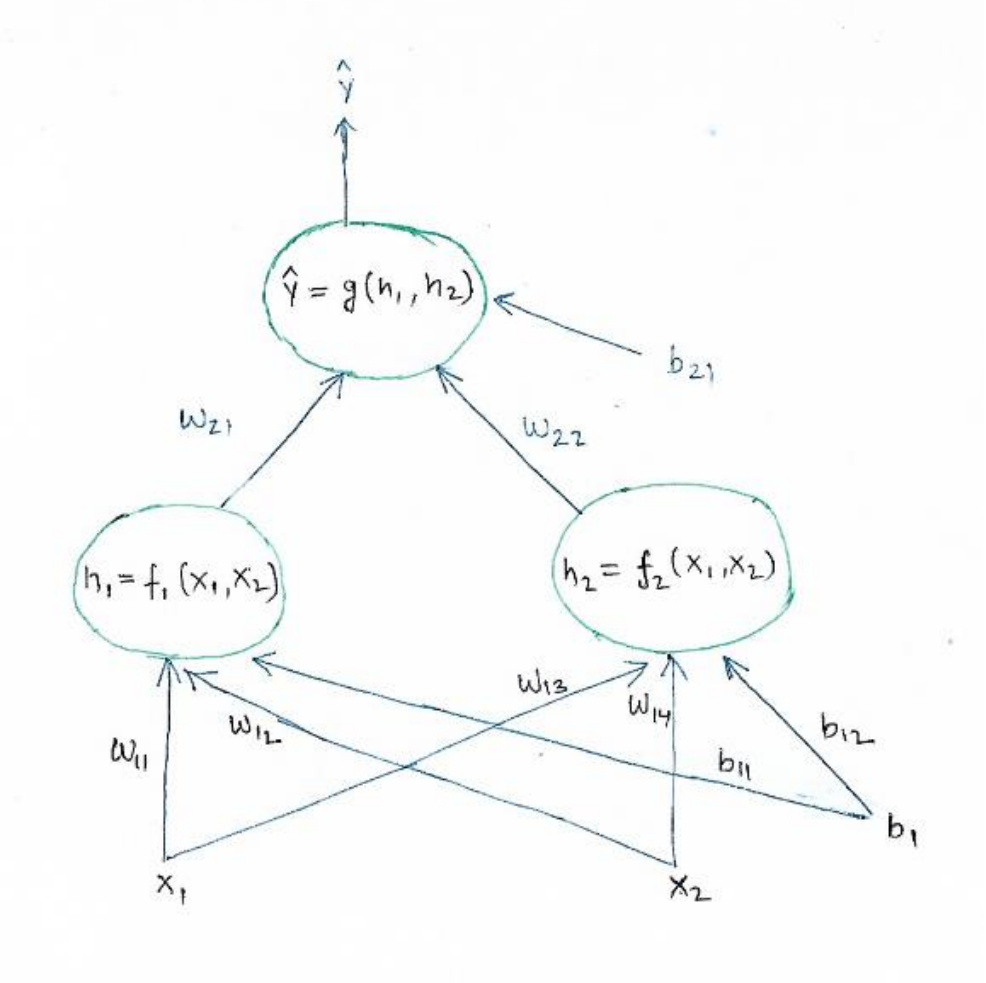
Consider the previously used example of mobile phone like/dislike predictor with the variables Screen-size and Cost. It has a complex decision boundary as shown here



With a single sigmoid neuron, it is impossible to obtain this shape, regardless of how we vary the parameters w & b, as the sigmoid neuron can only produce a shape ranging from s-shaped to flat.

The formula is $\hat{y} = f(x_1, x_2)$ *or* $\hat{y} = \frac{1}{1 + e^{-(w_1 * x_1 + w_2 * x_2 + b)}}$

Now, let us consider a Deep Neural Network for the same mobile phone like/dislike predictor



Breaking down the model:
x1 = Screen-Size, x2 = Cost

Output of FIRST NEURON

$$h_1 = f_1(x_1, x_2)$$

If we apply the sigmoid function to the inputs x_1 and x_2 with the appropriate weights w_{11}, w_{12} and bias b_1 we would get an output h_1, which would be some real value between 0 and 1. The sigmoid output for the first neuron h_1 will be given by the following equation,

$$h_1 = \frac{1}{1+e^{-(w_{11}*x_1+w_{12}*x_2+b_1)}}$$

a. Here, w11 and w12 are the weights of x1 and x2 corresponding to the first neuron h1
b. b11 is the corresponding bias

Output of SECOND NEURON

$$h_2 = f_2(x_1, x_2)$$

As explained for first neuron similarly for second neuron

$$h_2 = \frac{1}{1+e^{-(w_{13}*x_1+w_{14}*x_2+b_2)}}$$

Here, w13 and w14 are the weights of x1 and x2 corresponding to the second neuron h2

a) b12 is the corresponding bias

OUTPUT NEURON

$$\hat{y} = g(h_1, h_2)$$

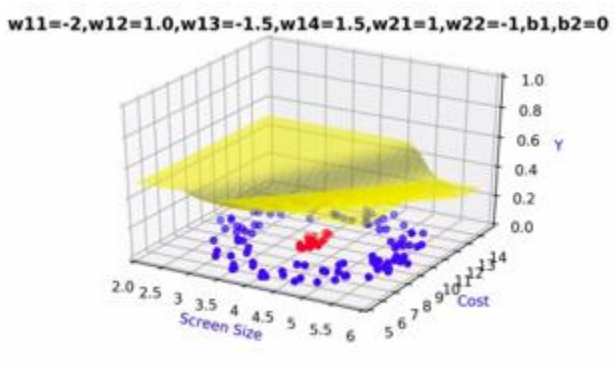
$$\hat{y} = \frac{1}{1+e^{-(w_{21}*h_1+w_{22}*h_2+b_3)}}$$

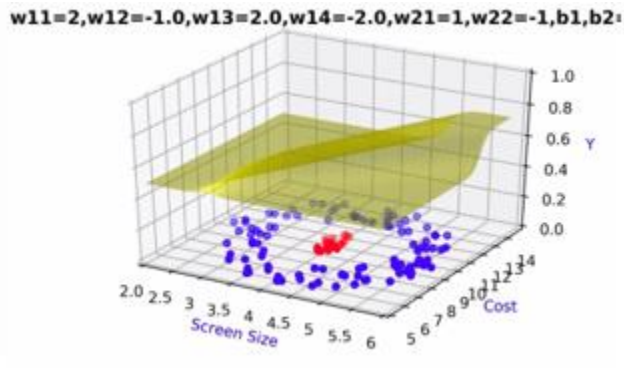
$$= \frac{1}{1+e^{-(w_{21}*(\frac{1}{1+e^{-(w_{11}*x_1+w_{12}*x_2+b_1)}})+w_{22}*(\frac{1}{1+e^{-(w_{13}*x_1+w_{14}*x_2+b_2)}})+b_3)}}$$

a. Here, w21 and w22 are the weights of h1 and h2 corresponding to the output neuron ŷ
b. b21 is the corresponding bias

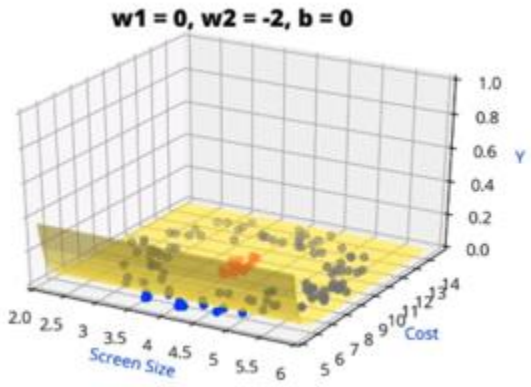
From this configuration, we have 9 parameters (w11, w12, w13, w14, w21, w22, b1, b2, b3), which allow for a much more complex decision boundary than a single sigmoid neuron with 3 parameters

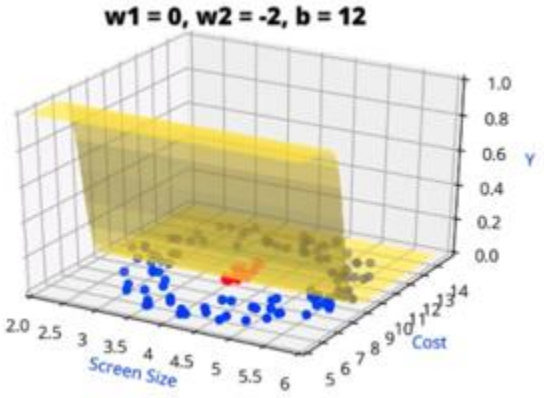
This model creates output something like this





While if we check sigmoid output is specific to S shape
Sigmoid decision boundary, can range from s-shaped to flat, based on w and b values

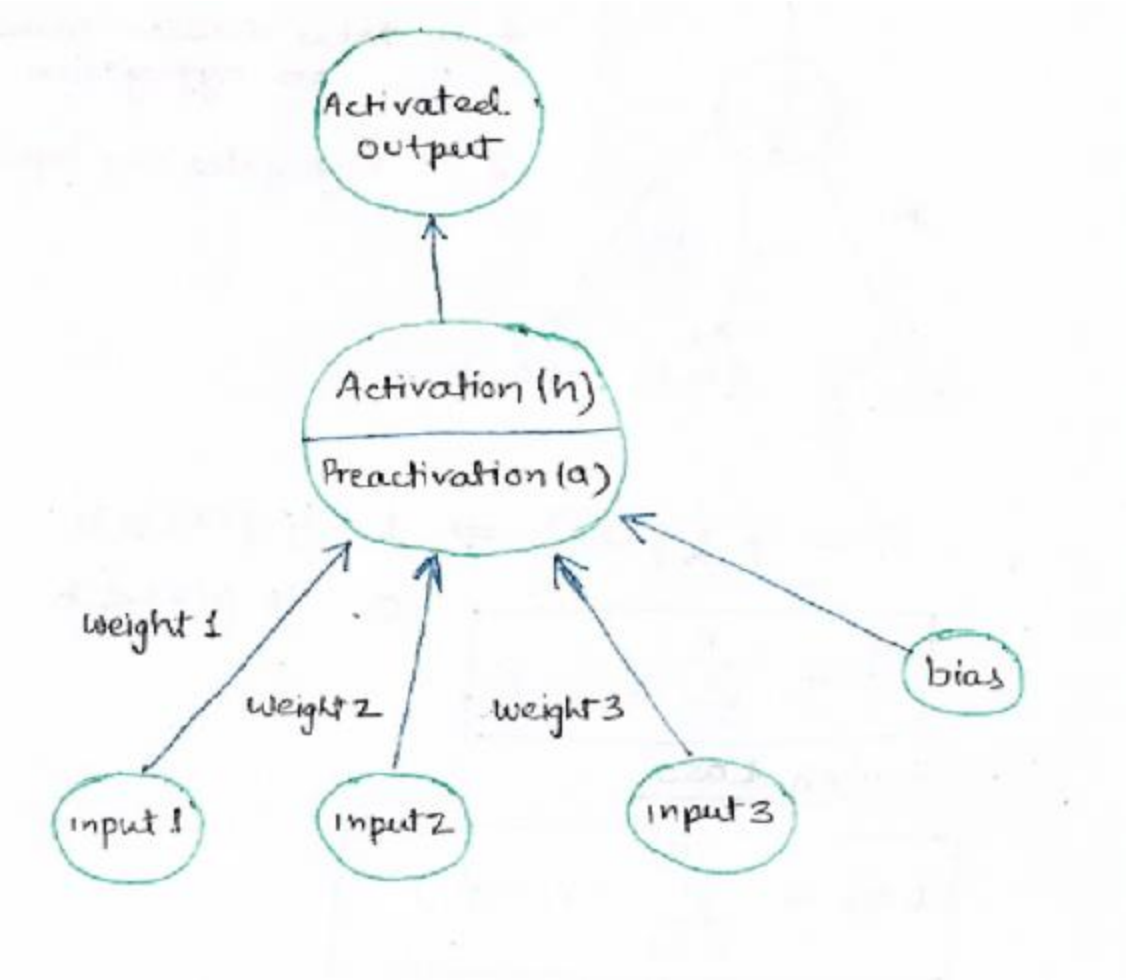




So, we can observe as we make model with multiple sigmoid with addition of layers the model is able to learn more complex algorithm now.

The next step would be figuring out how to choose the best configuration of the DNN for our task, this is called Hyperparameter Tuning.
For now, we can rest easy knowing that by the **Universal Approximation Theorem** we will be able to approximate any kind of function with our Neural Network

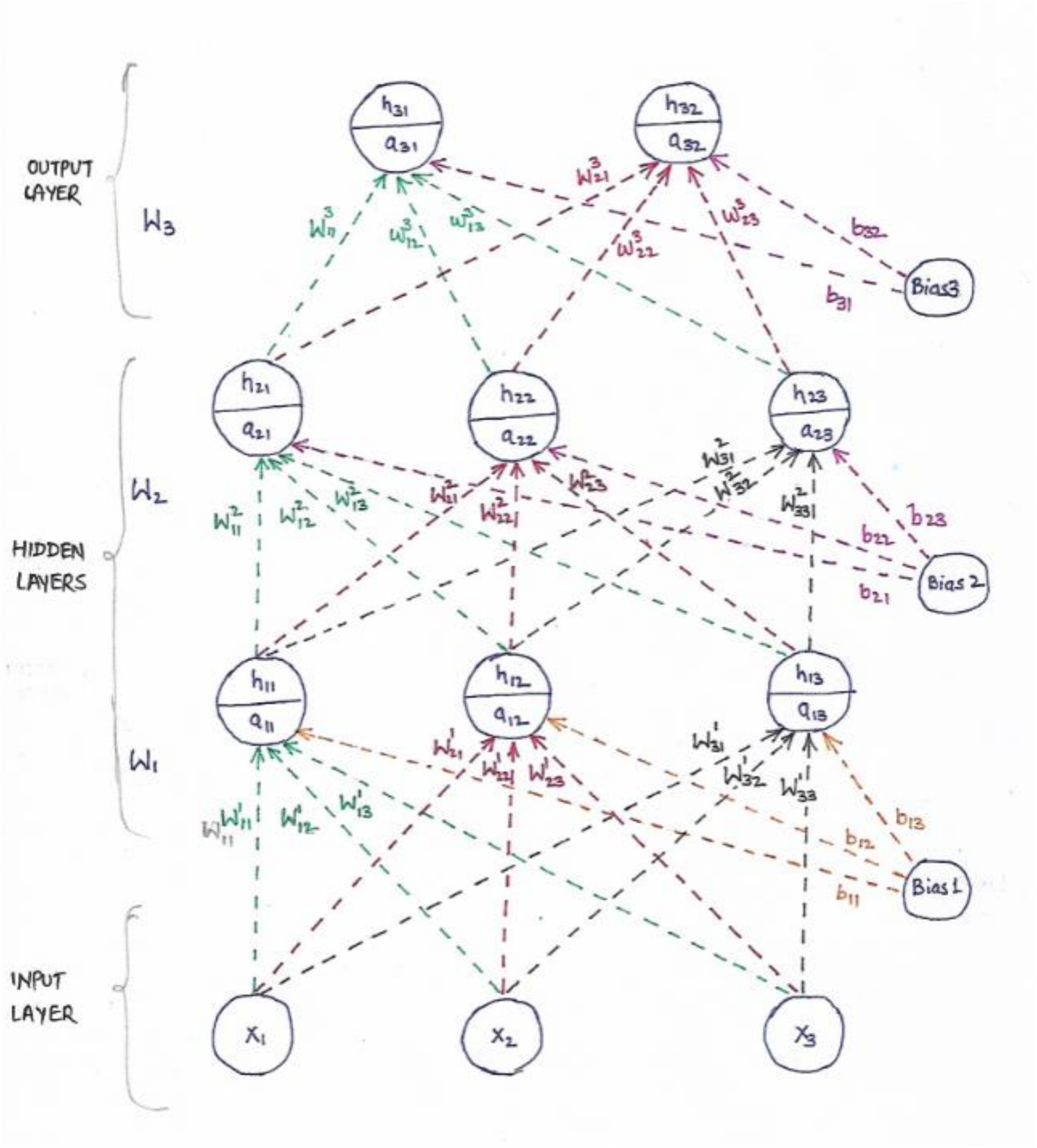
Let's check the structure of neuron again



Let us break down the terms

a. Let us refer to the layer being referenced
b. Pre-activation function $a_i = \Sigma(input * weights) + bias$
c. Activation function $h_i = \frac{1}{1+e^{-(a_i)}}$ a
d. Here, the activation function is the sigmoid function.
e. The construction of a Neural network is a simple stacking of these neurons in layers, one on top of the other
f. The outputs of one layer of neurons become the inputs for the next layer.
g. The cycle of pre-activation and activation repeats itself from the input layer, till we reach the output layer and obtain the desired function

Now if we break down the structure on neural networks



Let's break down some of the terms used:

Let's understand the network neuron by neuron. Consider the first neuron present in the first hidden layer. The first neuron is connected to each of the inputs by weight W_1.

Going forward I will be using this format of the indices to represent the weights and biases associated with a particular neuron,

W (Layer number) (Neuron number in the layer) (Input number)

b (Layer number) (Bias number associated for that input)

W_{111}— Weight associated with the first neuron present in the first hidden layer connected to the first input.

W_{112}— Weight associated with the first neuron present in the first hidden layer connected to the second input.

b_{11} — Bias associated with the first neuron present in the first hidden layer.
b_{12} — Bias associated with the second neuron present in the first hidden layer.
….
Here W_1 a weight matrix containing the individual weights associated with the respective inputs. The pre-activation at each layer is the weighted sum of the inputs from the previous layer plus bias. The mathematical equation for pre-activation at each layer 'i' is given by,

$$a_i(x) = W_i h_{i-1}(x) + b_i$$

The activation at each layer is equal to applying the sigmoid function to the output of pre-activation of that layer. The mathematical equation for the activation at each layer 'i' is given by,

$$h_i(x) = g(a_i(x))$$

where 'g' is called as the activation function

Finally, we can get the predicted output of the neural network by applying some kind of activation function (could be softmax depending on the task) to the pre-activation output of the previous layer. The equation for the predicted output is shown below,

$$f(x) = h_L = O(a_L)$$

where 'O' is called as the output activation function

Understanding the Computations in a Deep Neural Network

First, we will focus on first layer
Let's look at the computations inside a DNN

$$W_1 = \begin{bmatrix} w_{1\,1\,1} & w_{1\,1\,2} & . & . & . & w_{1\,1\,99} & w_{1\,1\,100} \\ w_{1\,2\,1} & w_{1\,2\,2} & . & . & . & w_{1\,2\,99} & w_{1\,2\,100} \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ w_{1\,10\,1} & w_{1\,10\,2} & . & . & . & w_{1\,10\,99} & w_{1\,10\,100} \end{bmatrix} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_{100} \end{bmatrix}$$

Here, the **pre-activation values are as follows** →

$$a_{11} = w_{111} * x_1 + w_{112} * x_2 + w_{113} * x_3 + b_{11}$$
$$a_{12} = w_{121} * x_1 + w_{122} * x_2 + w_{123} * x_3 + b_{12}$$
$$a_{13} = w_{131} * x_1 + w_{132} * x_2 + w_{133} * x_3 + b_{13}$$

These values are just the individual rows of the dot-product between W1 and X plus the bias vector
Thus, a1 is given by

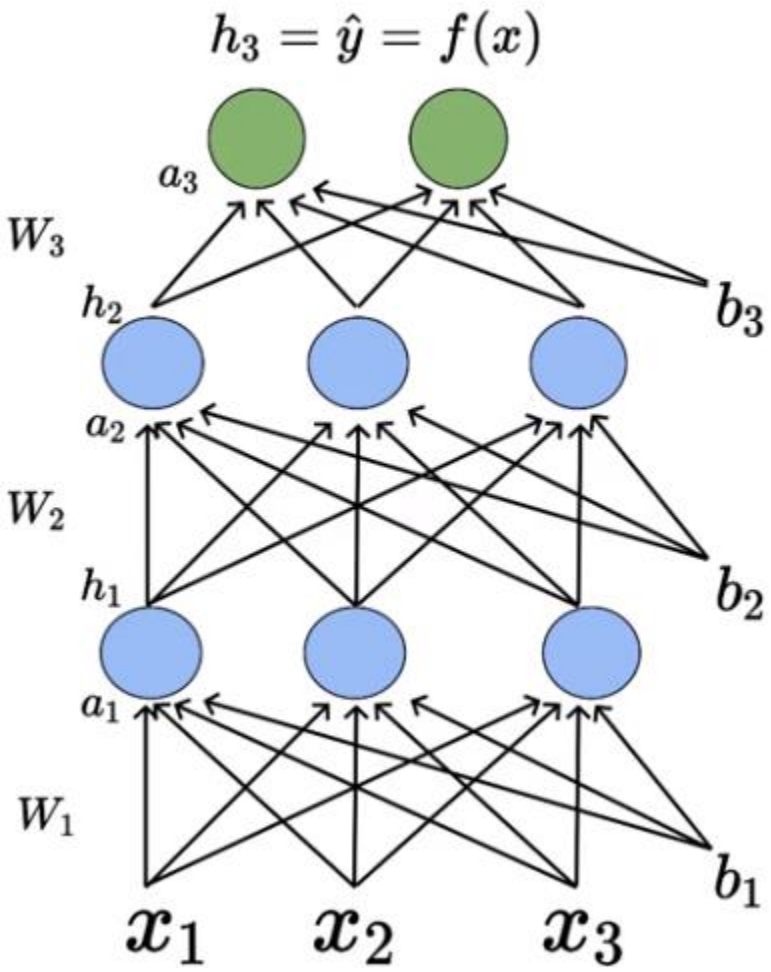
$$a_1 = W_1 * x + b_1$$

Where,
W_1 is a matrix containing the individual weights associated with the corresponding inputs and b_1 is a vector containing (b_{11}, b_{12}, b_{13}, …., b_{10}) the individual bias associated with the sigmoid neurons.
The activation for the first layer is given by,

$$h_1 = g(a_1)$$

Where 'g' represents the sigmoid function.
Remember that a_1 is a vector of 10 pre-activation values, here we are applying the element-wise sigmoid function on all these 10 values and storing them in another vector represented as h_1. Similarly, we can compute the pre-activation and activation values for 'n' number of hidden layers present in the network.

Output Layer of DNN
So far, we have talked about the computations in the hidden layer. Now we will talk about the computations in the output layer.



We can compute the pre-activation a_3 at the output layer by taking the dot product of weights associated W_3 and activation of the previous layer h_2 plus bias vector b_3.

$$W_3 = \begin{bmatrix} w_{3\,1\,1} & w_{3\,1\,2} & . & . & . & w_{3\,1\,10} \\ w_{3\,2\,1} & w_{3\,2\,2} & . & . & . & w_{3\,2\,10} \\ w_{3\,3\,1} & w_{3\,3\,2} & . & . & . & w_{3\,3\,10} \\ w_{3\,4\,1} & w_{3\,4\,2} & . & . & . & w_{3\,4\,10} \end{bmatrix} \qquad h_2 = \begin{bmatrix} h_{2\,1} \\ h_{2\,2} \\ . \\ . \\ h_{2\,10} \end{bmatrix}$$

$$a_{3\,1} = w_{3\,1\,1} * h_{2\,1} + w_{3\,1\,2} * h_{2\,2} + w_{3\,1\,3} * h_{2\,3} + ... + w_{3\,1\,10} * h_{2\,10} + b_{31}$$

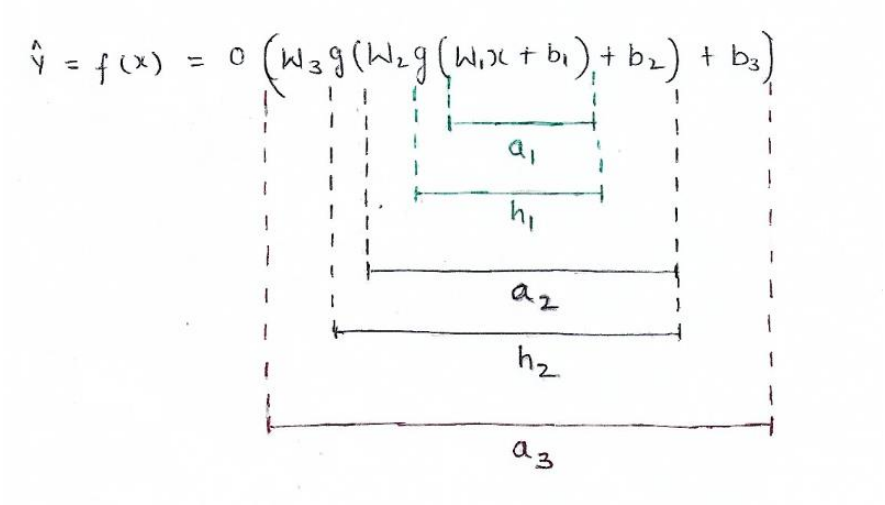
$$a_{3\,2} = w_{3\,2\,1} * h_{2\,1} + w_{3\,2\,2} * h_{2\,2} + w_{3\,2\,3} * h_{2\,3} + ... + w_{3\,2\,10} * h_{2\,10} + b_{32}$$

$$a_{3\,3} = w_{3\,3\,1} * h_{2\,1} + w_{3\,3\,2} * h_{2\,2} + w_{3\,3\,3} * h_{2\,3} + ... + w_{3\,3\,10} * h_{2\,10} + b_{33}$$

$$a_{3\,4} = w_{3\,4\,1} * h_{2\,1} + w_{3\,4\,2} * h_{2\,2} + w_{3\,4\,3} * h_{2\,3} + ... + w_{3\,4\,10} * h_{2\,10} + b_{34}$$

$$a_3 = W_3 h_2 + b_3$$

To find out the predicted output from the network, we are applying some function (which we don't know yet) to the pre-activation values.



$$\hat{y}_1 = O(a_{31}) \qquad \hat{y}_2 = O(a_{32})$$

These two outputs will form a probability distribution that means their summation would be equal to 1.

The output Activation function is chosen depending on the task at hand, can be softmax or linear.

Output Activation function is chosen depending on the task at hand (can be a softmax, linear)

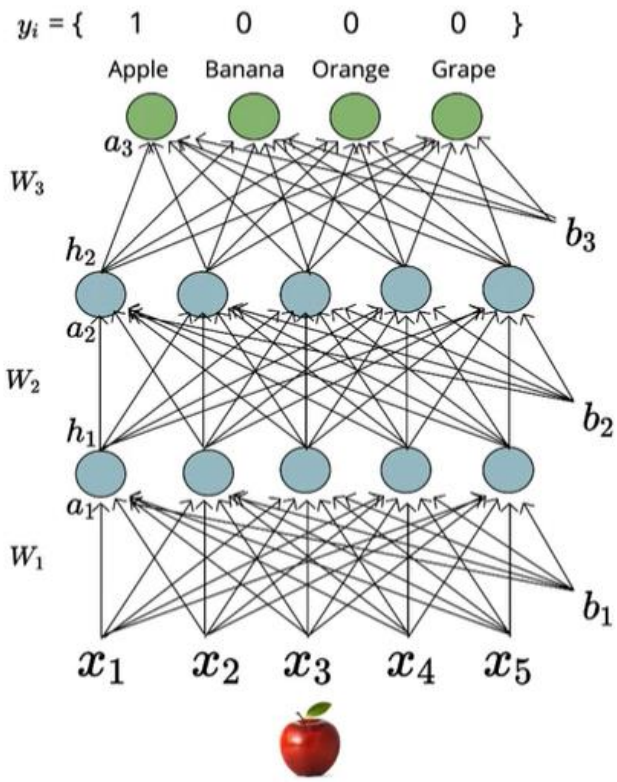
Softmax Function

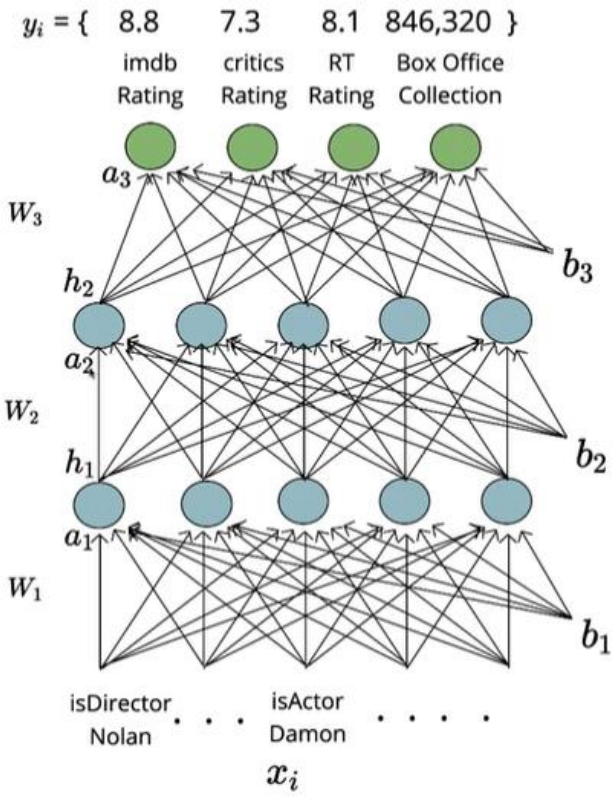
We will use the Softmax function as the output activation function. The most frequently used activation function in deep learning for classification problems.

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad for\ i = 1.....k$$

In the Softmax function, the output is always positive, irrespective of the input.

Consider the following multi-class classification problem





In first example we are trying to find probability values while 2nd case is regression problem where we need to predict values.

Consider first example, In last layer, we compute a3 = W3h2 + b3
We need to apply a function to ŷi = O(a3i) such that the 4 outputs form a probability distribution.

Output Activation Function has to be chosen such that output is probability

As we discussed before equation will be

$$W_3 = \begin{bmatrix} w_{3\,1\,1} & w_{3\,1\,2} & . & . & . & w_{3\,1\,10} \\ w_{3\,2\,1} & w_{3\,2\,2} & . & . & . & w_{3\,2\,10} \\ w_{3\,3\,1} & w_{3\,3\,2} & . & . & . & w_{3\,3\,10} \\ w_{3\,4\,1} & w_{3\,4\,2} & . & . & . & w_{3\,4\,10} \end{bmatrix} \qquad h_2 = \begin{bmatrix} h_{2\,1} \\ h_{2\,2} \\ . \\ . \\ h_{2\,10} \end{bmatrix}$$

$$a_3 = W_3 * h_2$$

$$\hat{y}_1 = O(a_{31}) \qquad \hat{y}_2 = O(a_{32}) \qquad \hat{y}_3 = O(a_{33}) \qquad \hat{y}_4 = O(a_{34})$$

Let's assume a3 = [3 4 10 3]

Take each entry and divide by the sum of all entries to get a probability distribution

$$\widehat{y_1} = \frac{3}{(3+4+10+3)} = 0.15$$

$$\widehat{y_2} = \frac{4}{(3+4+10+3)} = 0.20$$

$$\widehat{y_3} = \frac{10}{(3+4+10+3)} = 0.50$$

$$\widehat{y_4} = \frac{3}{(3+4+10+3)} = 0.15$$

However, this will not work if any of the entries are negative

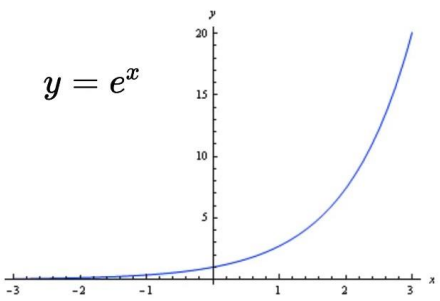
So, we consider the softmax function

SOFTMAX function

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad for\ i = 1.....k \qquad y = e^x$$

Note: the output of ex is always positive, irrespective of the input

$y = e^x$



This property is important to counter the negative-value shortcoming that we observed in the previous example

Now, let us illustrate the SoftMax function at the last layer of our Neural Network

$$a = [a_1\ a_2\ a_3\ a_4]$$

$$softmax(a) = \left[\frac{e^{a_1}}{\sum_{j=1}^{k} e^{a_j}}\ \frac{e^{a_2}}{\sum_{j=1}^{k} e^{a_j}}\ \frac{e^{a_3}}{\sum_{j=1}^{k} e^{a_j}}\ \frac{e^{a_4}}{\sum_{j=1}^{k} e^{a_j}}\right]$$

Applying the Softmax Function

Raising the numerators to ex ensures that they are all positive
The denominator is just the sum of all the values raised to ex
$softmax(a_i)\ is\ the\ i^{th} element\ of\ the\ softmax\ output$
So, for our multi-class fruit classifier, the equations are as follows

Layer	Pre-activation	Activation/Output
Hidden Layer 1	$a_1 = W_1 * x + b_1$	$h_1 = g(a_1)$
Hidden Layer 2	$a_2 = W_2 * h_1 + b_2$	$h_2 = g(a_2)$
Output Layer	$a_3 = W_3 * h_2 + b_3$	$\hat{y} = softmax(a_3)$

By applying the softmax function we would get a predicted probability distribution and our true output is also a probability distribution, we can compare these two distributions to compute the loss of the network.

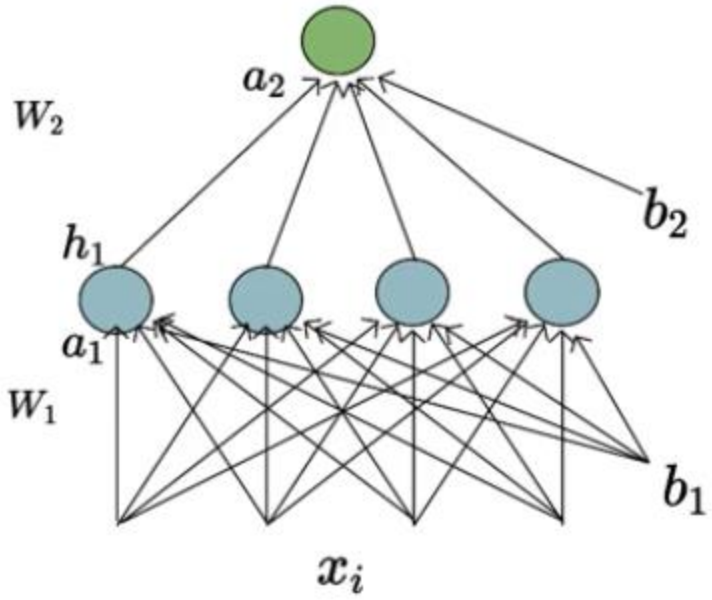
Process of tuning the DNN i.e. The No. of layers, No. of neurons in each layer, learning rate, batch size etc are together called Hyperparameter Tuning

Regarding regression output problem we will discuss later on.

LOSS FUNCTION

Loss function for Binary Classification

What is the loss function that you can use for a binary classification problem?
In normal cases, the number of neurons in the output layer would be equal to the number of classes, for example in below figure we have one neuron which means binary classification problem, although binary is two class problem but we can use **one neuron in output layer for binary classification problem**.
However, a shortcut in the case of binary classification would be to use only one output neuron that uses a sigmoid function. Here is a diagrammatic representation of that configuration



Here, ŷ = P(y = 1)
Therefore, we can obtain P(y = 0) = 1 – [P(y = 1)]
Consider the following values for the variables

$$b = [\ 0.5 \quad 0.3\ ]$$

$$W_1 = \begin{bmatrix} 0.9 & 0.2 & 0.4 & 0.3 \\ -0.5 & 0.4 & 0.3 & 0.3 \\ 0.1 & 0.1 & -0.1 & 0.2 \\ -0.2 & 0.5 & 0.5 & 0.7 \end{bmatrix}$$

$$W_2 = [0.5 \quad 0.8 \quad -0.6 \quad 0.3]$$

Input Given :→

$$x = [\ 0.3 \quad 0.5 \quad -0.4 \quad 0.3\ ] \qquad y = 1$$

Output :

$$a_1 = W_1 * x + b_1 = [\ 0.8 \quad 0.52 \quad 0.68 \quad 0.7\ ]$$

$$h_1 = sigmoid(a_1) = [\ 0.69 \quad 0.63 \quad 0.66 \quad 0.67\ ]$$

$$a_2 = W_2 * h_1 + b_2 = 0.948$$

$$\hat{y} = sigmoid(a_2) = 0.7207$$

Cross Entropy Loss:

$$L(\Theta) = \begin{cases} -log(\hat{y}) & \text{if } y = 1 \\ -log(1-\hat{y}) & \text{if } y = 0 \end{cases}$$

In this case $y = 1$
$True\ distribution$ [0 1]

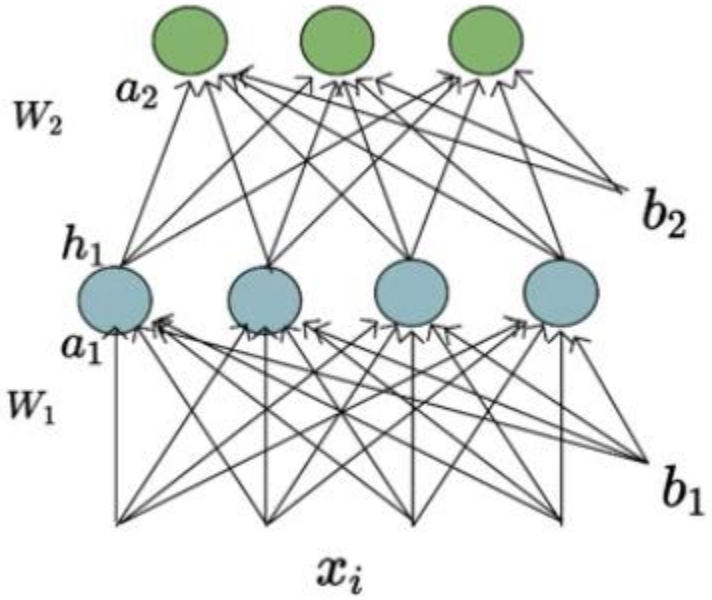
$Predicted\ distribution\ \hat{y}$ $[0.2793 \quad 0.7207]$
Cross Entropy Loss:
$L(\Theta) = (y)(-log\ (\hat{y})) + (1-y)(-log\ (\hat{y}))$

In this case, since y = 1
$L(\Theta) = -1 * log(0.7207)$
$L(\Theta) = 0.327$

So this is cross entropy for binary classification problem

Loss function for Multi-Class Classification

What is the loss function that you can use for a multi-class classification problem?
Here is an illustration of a sample multi-class classification Neural Network



Consider the following values for the parameters

$$b = [\ 0 \quad 0\ ]$$

$$W_1 = \begin{bmatrix} 0.1 & 0.3 & 0.8 & -0.4 \\ -0.3 & -0.2 & 0.5 & 0.5 \\ -0.3 & 0.1 & 0.5 & 0.4 \\ 0.2 & 0.5 & -0.9 & 0.7 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0.3 & 0.8 & -0.2 & -0.4 \\ 0.5 & -0.2 & -0.3 & 0.5 \\ 0.3 & 0.1 & 0.6 & 0.6 \end{bmatrix}$$

Consider Input value X and Y as following

$$x = [\ 0.2 \quad 0.5 \quad -0.3 \quad 0.3\ ] \qquad y = [\ 0 \quad 1 \quad 0\ ]$$

Output :

$$\begin{aligned} a_1 &= W_1 * x + b_1 = [\ -0.19 \quad -0.16 \quad -0.09 \quad 0.77\ ] \\ h_1 &= sigmoid(a_1) = [\ 0.45 \quad 0.46 \quad 0.49 \quad 0.68\ ] \\ a_2 &= W_2 * h_1 + b_2 = [\ 0.13 \quad 0.33 \quad 0.89\ ] \\ \hat{y} &= softmax(a_2) = [\ 0.23 \quad 0.28 \quad 0.49\ ] \end{aligned}$$

Cross Entropy Loss:

$$L(\Theta) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$$

Cross Entropy Loss

$L(\Theta) = -1 * log\ (0.28)$

$L(\Theta) = 1.2729$

Consider another case: →

$$x = [\ 0.6 \quad 0.4 \quad 0.6 \quad 0.1\ ] \qquad y = [\ 0 \quad 0 \quad 1\ ]$$

Output :

$$\begin{aligned} a_1 &= W_1 * x + b_1 = [\ 0.62 \quad 0.09 \quad 0.2 \quad -0.15\ ] \\ h_1 &= sigmoid(a_1) = [\ 0.65 \quad 0.52 \quad 0.55 \quad 0.46\ ] \\ a_2 &= W_2 * h_1 + b_2 = [\ 0.32 \quad 0.29 \quad 0.85\ ] \\ \hat{y} &= softmax(a_2) = [\ 0.2718 \quad 0.2634 \quad 0.4648\ ] \end{aligned}$$

Cross Entropy Loss:

$$L(\Theta) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$$

Cross Entropy Loss

$L(\Theta) = -1 * log\ (0.4648)$

$L(\Theta) = 0.7661$

LEARNING ALGORITHM

Can we use the same Gradient Descent algorithm?

Initialise w, b

Iterate over data:

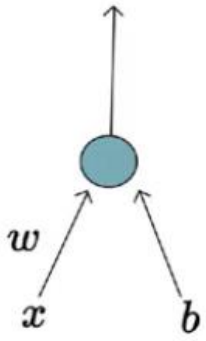
$compute\ \hat{y}$

$compute\ \mathscr{L}(w, b)$

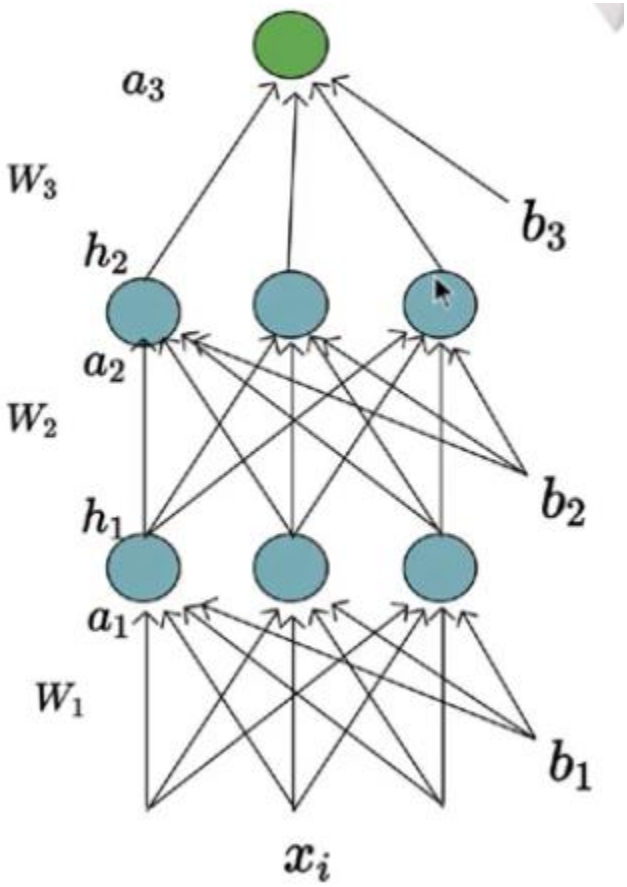
$w = w - \eta \Delta w$

$b = b - \eta \Delta b$

till satisfied



Earlier we have w, b for single neuron but here we have multiple weights



Initialise w, b

Iterate over data:

$compute\ \hat{y}$

$compute\ \mathscr{L}(w, b)$

$w_{111} = w_{111} - \eta\Delta w_{111}$

$w_{112} = w_{112} - \eta\Delta w_{112}$

....

$w_{313} = w_{313} - \eta\Delta w_{313}$

till satisfied

The algorithm

- Initialise: w111, w112, … w313, b1, b2, b3 randomly
- Iterate over data
- Compute ŷ
- Compute L(w,b) Cross-entropy loss function
  For each observation find the corresponding predicted distribution from the neural network and compute the loss using cross-entropy function.
- w111 = w111 - η**Δ**w111
- w112 = w112 - η**Δ**w112
- …
- w313 = w111 - η**Δ**w313
- bi = bi + η**Δ**bi
- Pytorch/Tensorflow have functions to compute $\frac{\delta l}{\delta w}\ and\ \frac{\delta l}{\delta b}$
- Till satisfied
  Based on the loss value, we will update the weights such that the overall loss of the model at the new parameters will be **less than the current loss** of the model.

- Number of epochs is reached (ie 1000 passes/epochs)
- Continue till Loss < ε (some defined value)

EVALUATION

How do you check the performance of a deep neural network?

Consider the Indian Liver Patient Diagnosis task

Age	Albumin	T_Bilirubin		y	ŷ
65	3.3	0.7		0	0
62	3.2	10.9		0	1
20	4	1.1	…… More features	1	1
84	3.2	0.7		1	0

The question here is, how do we resolve a probability distribution like [0.45 0.55] to a binary output
It is done by picking the class that corresponds to the highest probability, in the above case it is 1.
Predicted output is calculated by taking highest probability value as 1

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$
$$Accuracy = \frac{2}{4} = 50\%$$

For multiclass classification, the concept remains the same. The label is selected based on the highest value in the probability distribution.
The predicted label corresponds to the index of the highest value in the probability distribution (argmax)

Test Data	y	Predicted
	0	0
	1	7
	3	8
	5	5
	1	1

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$
$$Accuracy = \frac{3}{5} = 60\%$$

In addition to accuracy, we can also calculate the per-class accuracy. In this case, the accuracy of the class '1' is 50% and of class '5' is 100%
This allows us to analyze where the model is performing poorly, and enables us to take steps to improve the accuracy for the lagging classes, such as adding more images etc.

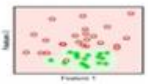
SUMMARY

Real inputs

$x_i \in \mathbb{R}$

- Binary Classification
- Multi-class classification
- Regression

$$\hat{y} = \frac{1}{1+e^{-(w_{21}*(\frac{1}{1+e^{-(w_{11}*x_1+w_{12}*x_2+b_1)}})+w_{22}*(\frac{1}{1+e^{-(w_{13}*x_1+w_{14}*x_2+b_1)}})+b_2)}}$$



Squared Error Loss :

$$L(\Theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{i=1}^{d}(\hat{y}_{ij} - y_{ij})^2$$

Cross Entropy Loss:

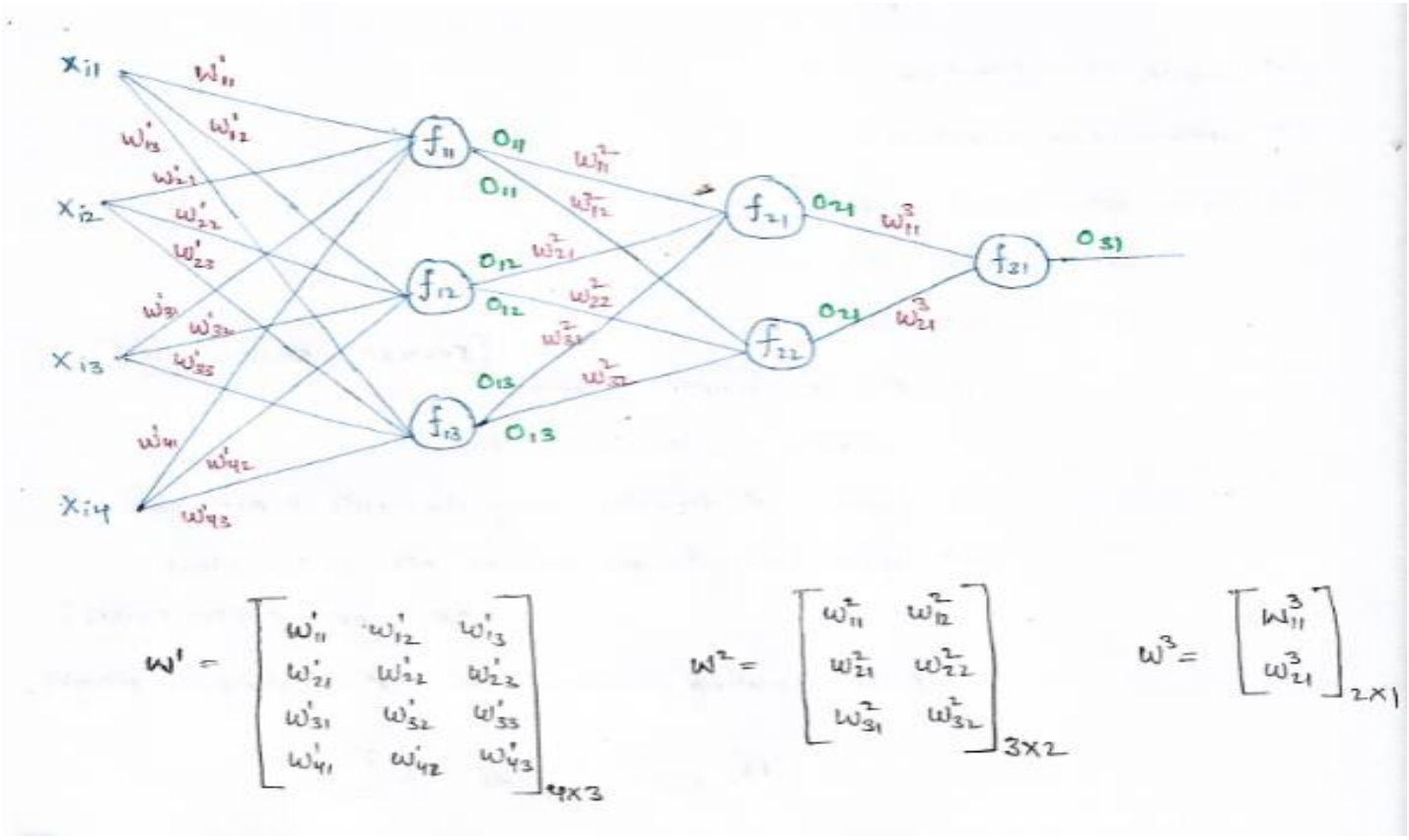
$$L(\Theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{i=1}^{d} y_{ij}\log(\hat{y}_{ij})$$

Gradient Descent with backpropagation

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

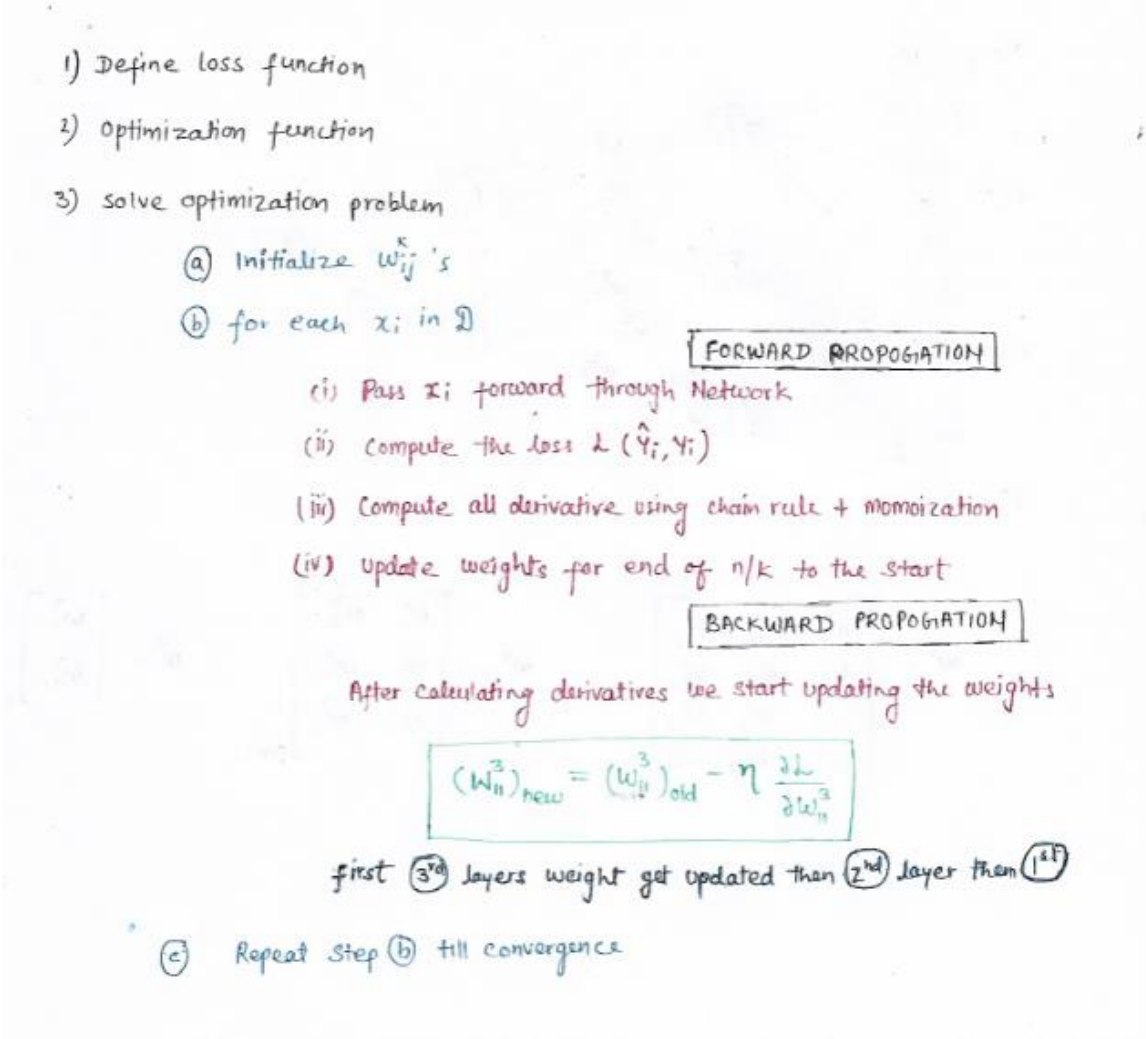
Understanding FORWARD PROPOGATION, BACKWARD PROPOGATION in depth

Let’s discuss about neural networks



In above diagram W represents weights, F represents functions, O represents outputs
X represents inputs
Now when we pass input through this neural network in last we get output and from predicted and actual output are used to find LOSS and using optimization function we will reduce loss and in order to reduce loss we will adjust weights this is a continuous process and this is how training is performed now let's discuss this in details

STEPS for TRAINING



{One EPOCH: when we pass all the data point once through neural network}

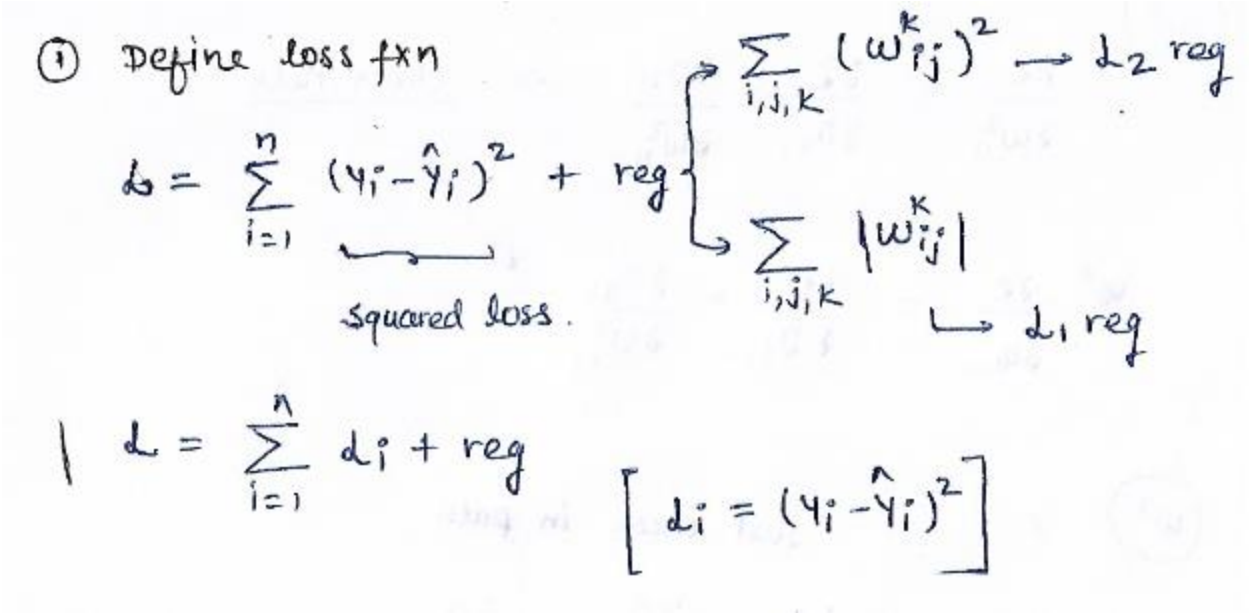
In above figure step 3.b is Epoch →

Forward propagation → Compute loss → Compute derivatives chain rule → update weight (Backward Propagation)

Go through above figure once again so that it will be easier to understand below remaining part.

Now let's discuss each step one by one

1) DEFINE LOSS FUNCTION
   For simplicity lets consider square loss function



2) OPTIMIZATION FUNCTION

② optimization

$$\min_{w_i} \sum_{i=1}^{n} (y_i - \underbrace{w^T x_i}_{\hat{y}_i})^2 + \text{regularizer}$$

from neural n/k $\hat{y}_i = f(w^T x_i)$

for linear regression f is identity

$$\min_{w_i} \sum_{i=1}^{n} (y_i - f(w^T x_i))^2 + \text{regularizer.}$$

↳ I : linear Regression

sigmoid : Log - Reg

Thresholding : Perceptron

$$w^* = \underset{w}{\text{argmin}} \sum_{i=1}^{n} (y_i - f(w^T x_i))^2 + \text{reg.}$$

3) SOLVE OPTIMIZATION PROBLEM

③ Solve optimize problem

Here we discussed How SGD works

ⓐ Initialization of weights w_i's

Random initialization

ⓑ $\nabla_w L = \begin{bmatrix} \partial L/\partial w_1 \\ \partial L/\partial w_2 \\ \vdots \\ \partial L/\partial w_4 \end{bmatrix}$ Vector derivative of L w.r.t. w

ⓒ $w_{new} = w_{old} - \eta \left[\nabla_w L\right]$ compute with w_{old} val

$$(w_i)_{new} = (w_i)_{old} - \eta \left[\frac{\partial L}{\partial w_i}\right]_{(w_i)\,old}$$

for iteratio = 1 to K

until converged.

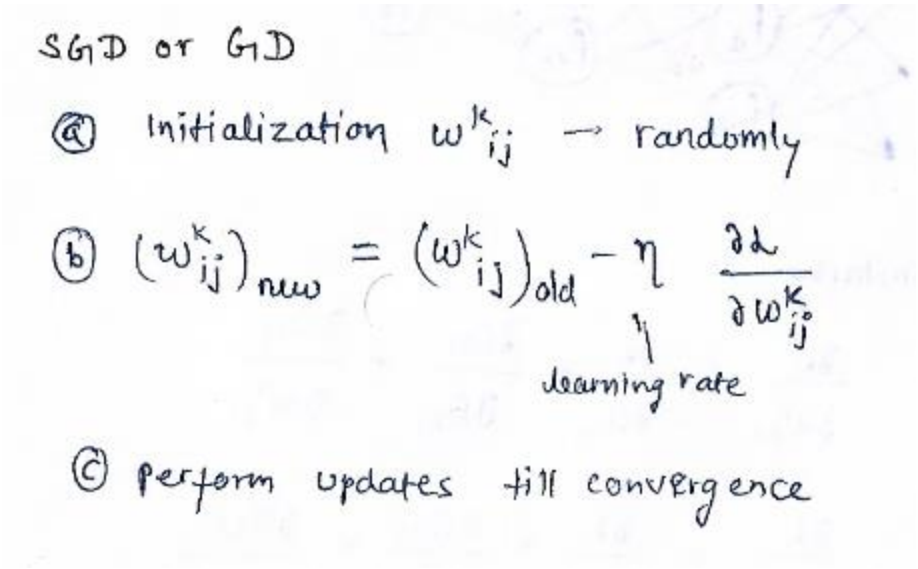
In standard gradient descent we compute for all values of input x_i's

In case of SGD

$\nabla_w L \approx$ one pt $\{x_i, y_i\}$ or

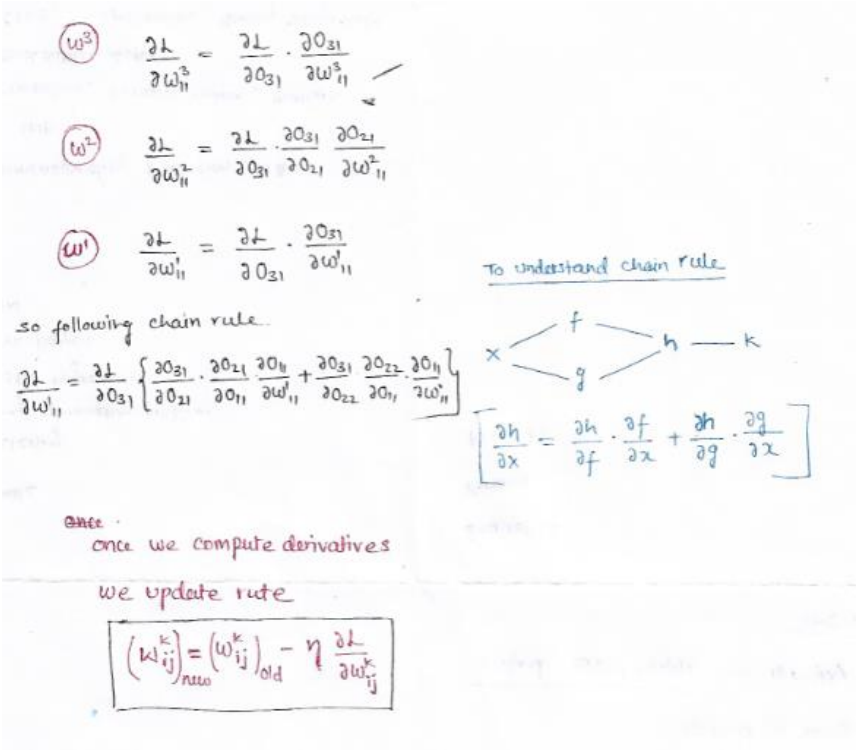
small batch of points ← batch.

In short we are doing something like following



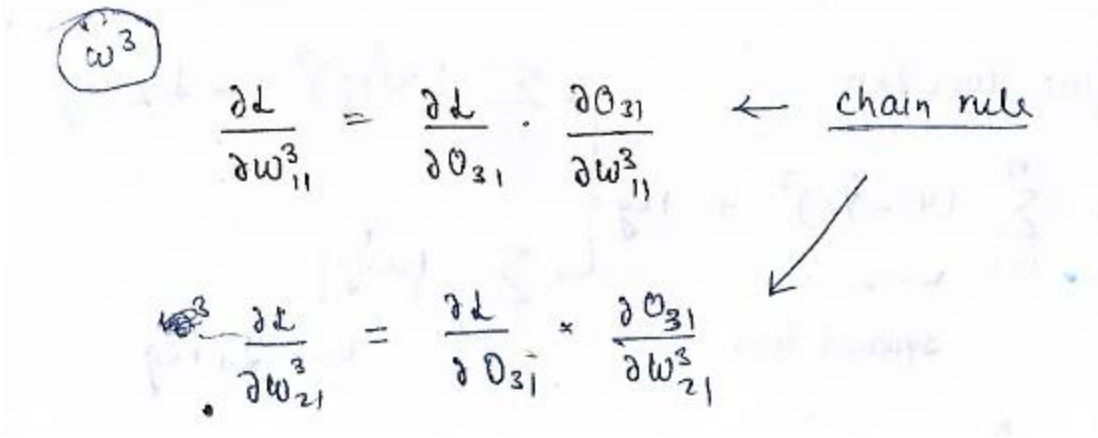
Now let's discuss about chain rule

TRAINING NN: CHAIN RULE



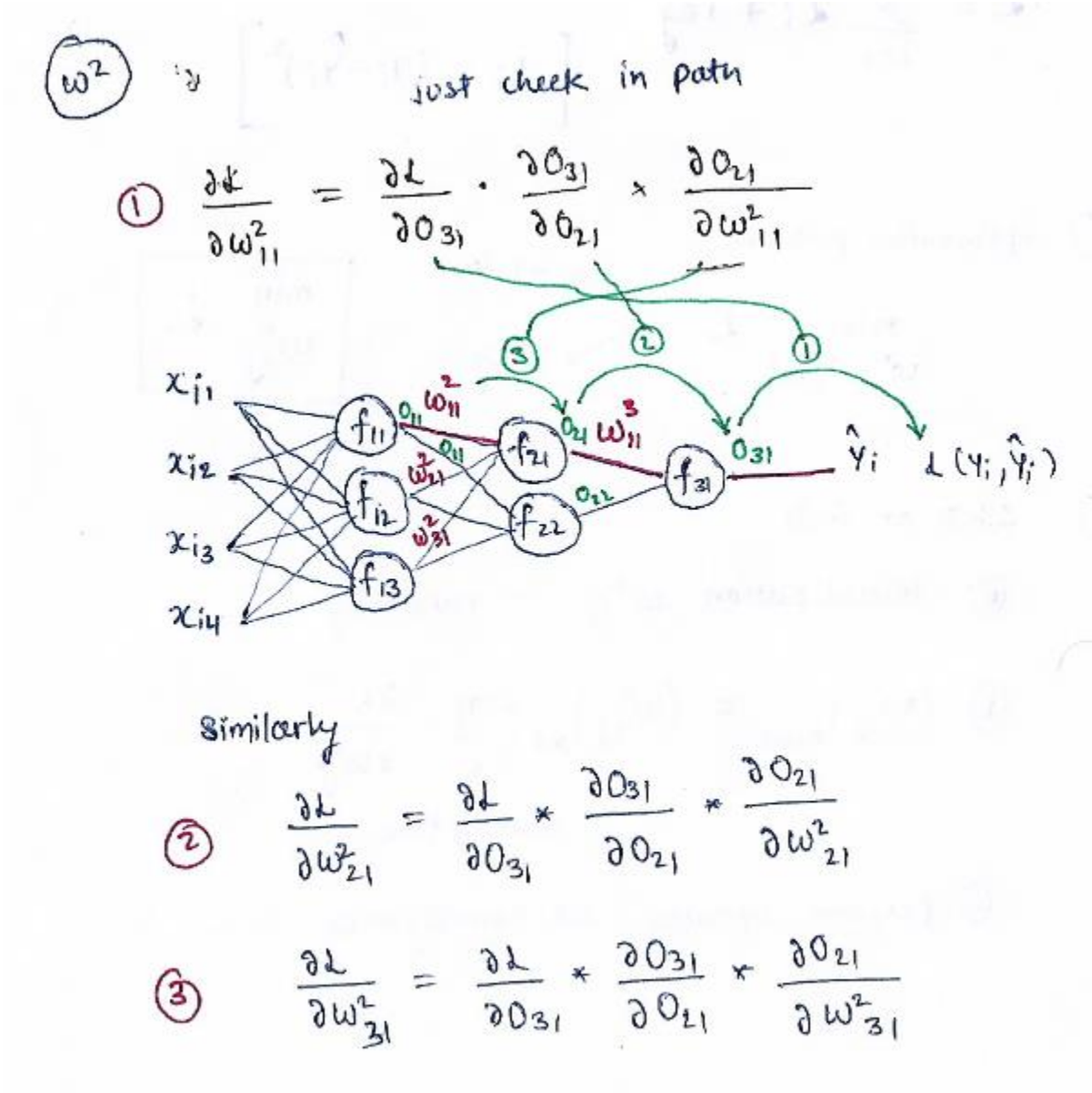
If from above point it is not clear than let's consider each one of them one by one

W3 weights are straight forward



In W2

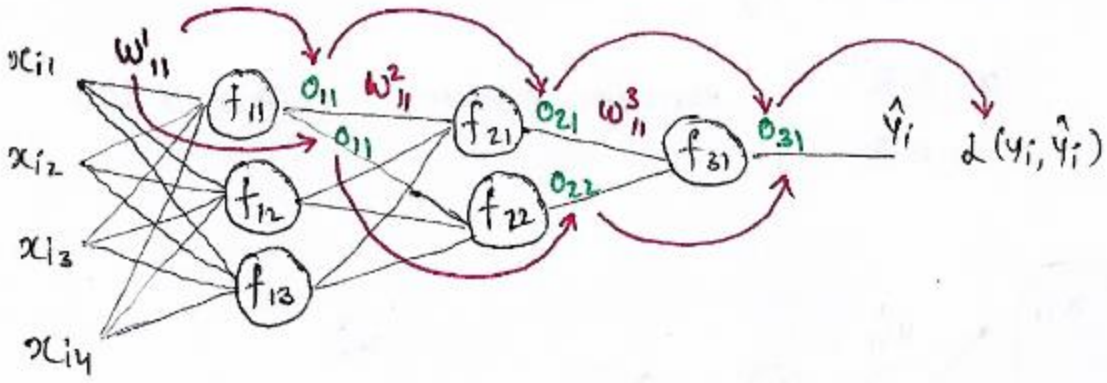
Since we have only single path available so we calculate gradient according to it



Now in below scenario multiple paths are available so we need to calculate derivative accordingly as mentioned below

(w') Here we find 2 paths.

$$\frac{\partial L}{\partial w'_{11}} =$$



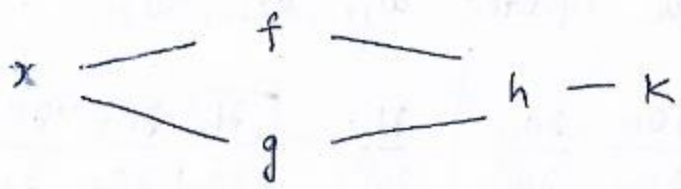
To understand it consider this example

$$\frac{\partial h}{\partial x} = \underbrace{\frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial x}}_{\text{one path}} + \underbrace{\frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial x}}_{2^{nd} \text{ path}}$$

$$\frac{\partial L}{\partial w'_{11}} = \frac{\partial L}{\partial o_{31}} \cdot \frac{\partial o_{31}}{\partial w'_{11}}$$

To understand How to compute this lets consider another example

In continuation



$$\frac{\partial K}{\partial x} = \frac{\partial K}{\partial h} \cdot \frac{\partial h}{\partial x}$$

as we already discussed.

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial x}$$

$$\frac{\partial K}{\partial x} = \frac{\partial K}{\partial h} * \left\{ \frac{\partial h}{\partial f} \cdot \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} \cdot \frac{\partial g}{\partial x} \right\}$$

Now consider we need to compute

$$\frac{\partial L}{\partial w'_{11}} = \frac{\partial L}{\partial O_{31}} \frac{\partial O_{31}}{\partial w'_{11}}$$

$$\frac{\partial O_{31}}{\partial w'_{11}} = \underbrace{\frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial O_{11}} \frac{\partial O_{11}}{\partial w'_{11}}}_{\text{Path 1}} + \underbrace{\frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial w'_{11}}}_{\text{Path 2}}$$

(w')

$$\partial w'_{11} = \frac{\partial L}{\partial O_{31}} \cdot \left\{ \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial w'_{11}} + \frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial w'_{11}} \right\}$$

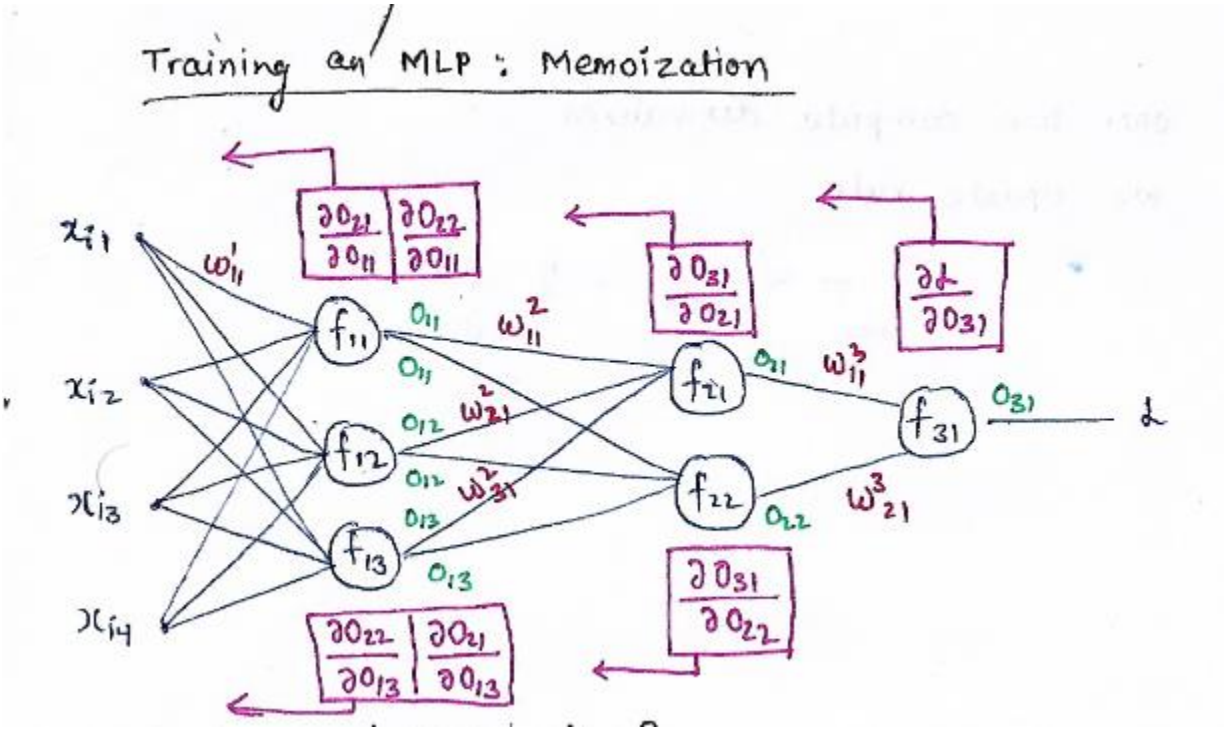
Once we have compute derivatives, we will update rule

we update rule

$$w_{ij}^{k}{}_{(new)} = w_{ij}^{k}{}_{(old)} - \eta \frac{\partial L}{\partial w_{ij}^{k}}$$

Another important concept is Memorization

It is memorizing repeated derivative for faster calculation



When we update w_{11}^3 & w_{21}^3 we compute $\frac{\partial L}{\partial w_{11}^3}$ & $\frac{\partial L}{\partial w_{21}^3}$

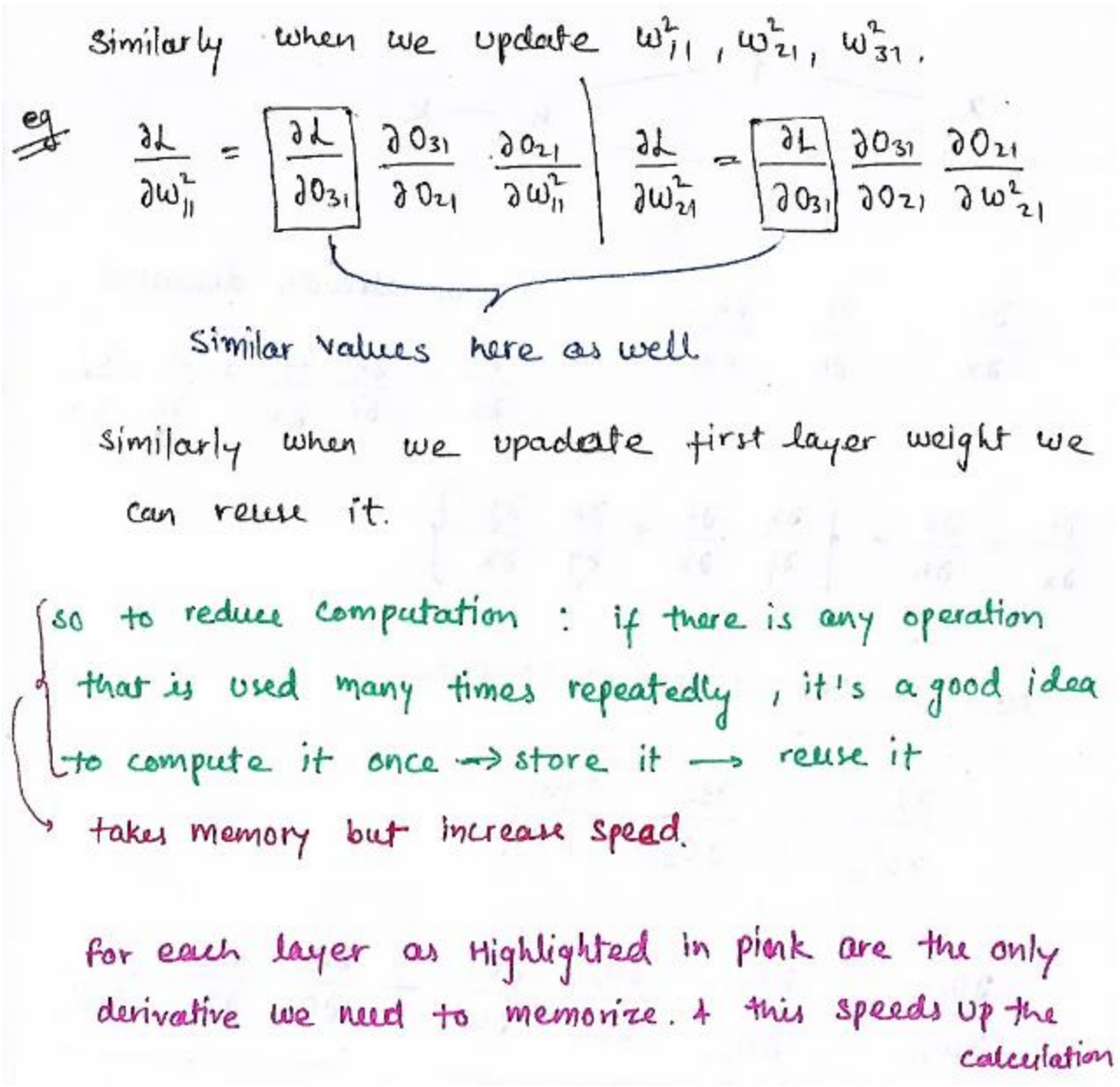
Using chain rule

$$\frac{\partial L}{\partial w_{11}^3} = \boxed{\frac{\partial L}{\partial O_{31}}} \frac{\partial O_{31}}{\partial w_{11}^3}$$

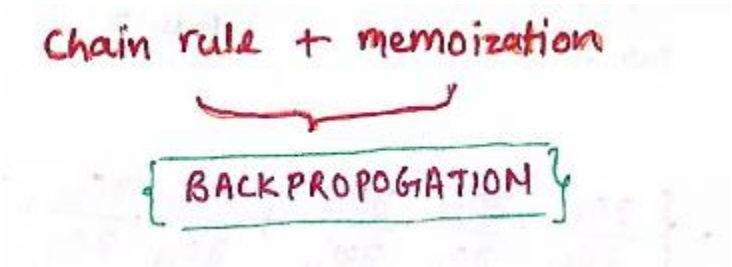
$$\frac{\partial L}{\partial w_{21}^3} = \boxed{\frac{\partial L}{\partial O_{31}}} \frac{\partial O_{31}}{\partial w_{21}^3}$$

Same thing caculate twice

So core idea is to compute repeative value once and reuse them again

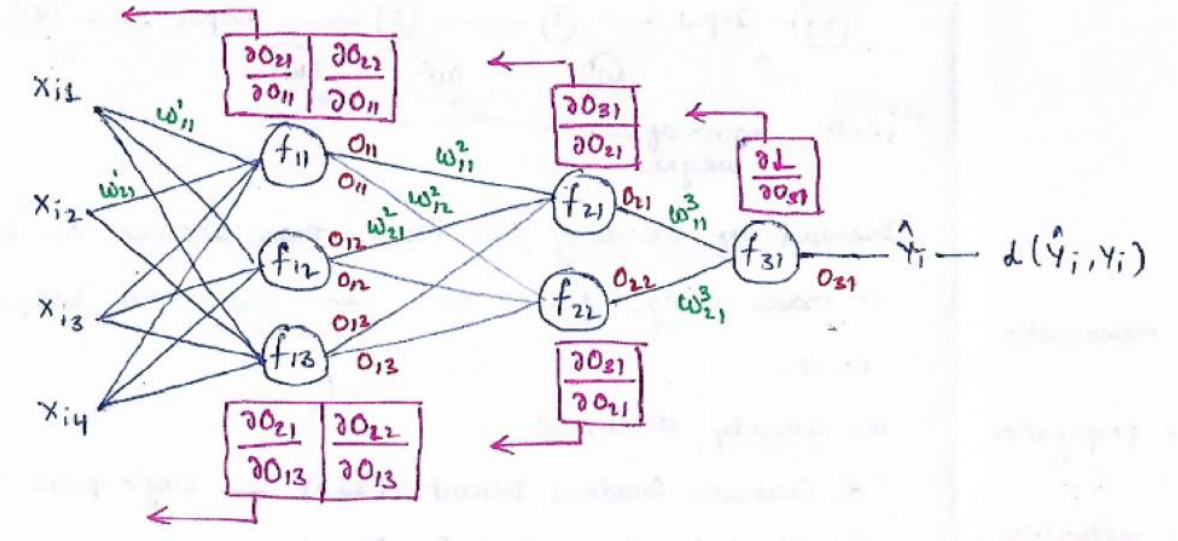


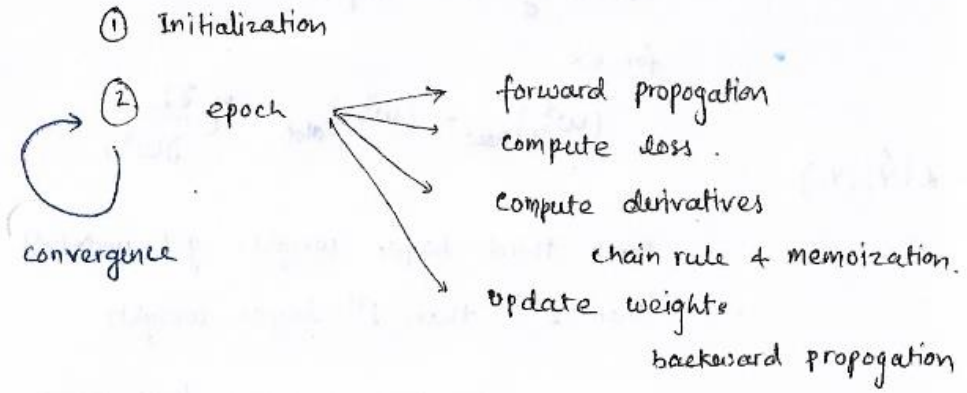
Now defining BACKPROPAGATION using chainm rule and memorization



BACKPROPOGATION

Lets discuss backpropagation in details



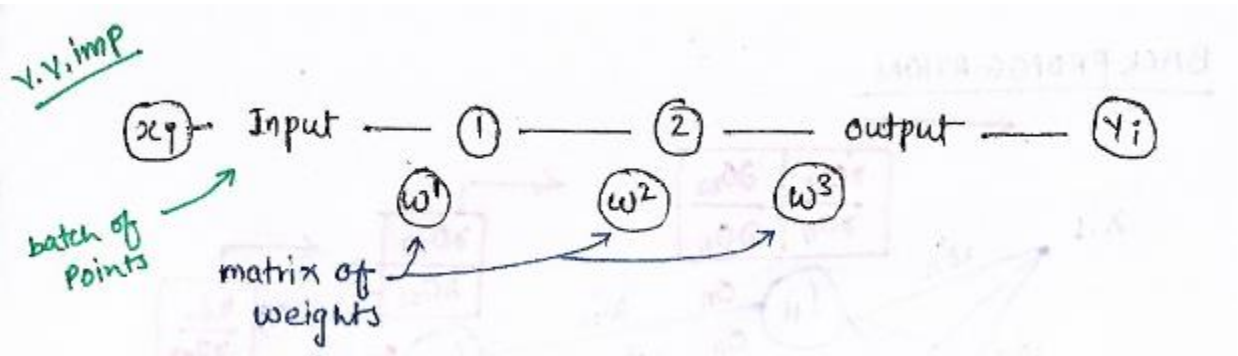


Backpropogation : It is multi epoch training methodology where we leverage chain rule & memoization to update weights .

BACKPROPAGATION works only when activation functions are differentiable

If these functions not differentiable then we can't use chain rule so we cannot update weights so we can say whole concept will not work

Also, if we can differentiate activation function faster we can speed up the training of the neural network



==========

GRADIENT DESCENT(GD), STOCHASTIC GD(SGD), MINI BATCH SGD

==========

- **BATCH GRADIENT DESCENT (GD)** –> Each epoch --> Consider all n points --> Mean calculation --> Parameter updated after each epoch
  ==> Each Epoch ==> All points ==> 1 step of Gradient Descent
- **STOCHASTIC GRADIENT DESCENT (SGD)** –> Each epoch --> One single point --> Parameter updated each epoch
  ==> Each Epoch = Single point
- **MINI BATCH STOCHASTIC GRADIENT DESCENT** –> Each epoch --> Consider all batch points --> Parameter updated after each epoch
  ==> Each Epoch ==> All batch points

TO update weights

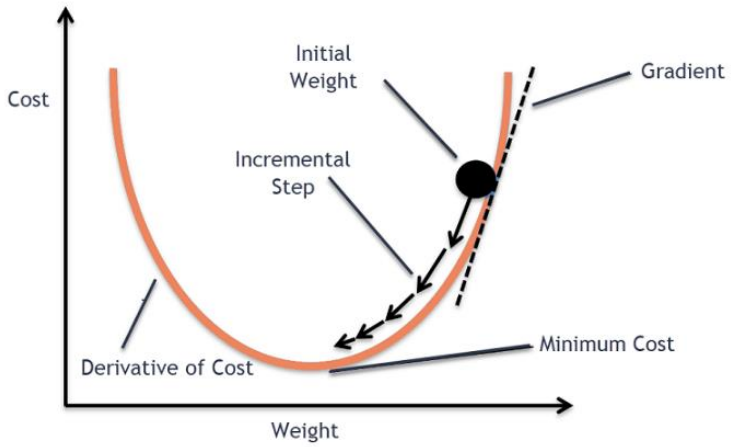
Mostly used MINI BATCH STOCHASTIC GRADIENT DESCENT
For example, 100k point in Dataset D → mini batch 100
1000 times → 1000 epoch → for each epoch 100 mini batch size
For each epoch → FORWARD PROPOGATION → LOSS → DERIVATIVES → UPDATE
→ BACKPROPAGATION

Since I have tried to explain BACKWARD PROPOGATION above but I will try to add separate article individually on BACKWARD PROPOGATION.

GRADIENT DESCENT

The goal of the gradient descent is to minimise a given function which, in our case, is the loss function of the neural network. To achieve this goal, it performs two steps iteratively.

1. Compute the slope (gradient) that is the first-order derivative of the function at the current point
2. Move-in the opposite direction of the slope increase from the current point by the computed amount



So, the idea is to pass the training set through the hidden layers of the neural network and then update the parameters of the layers by computing the gradients using the training samples from the training dataset.

BATCH GRADIENT DESCENT

In Batch Gradient Descent, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. **So that's just one step of gradient descent in one epoch**.

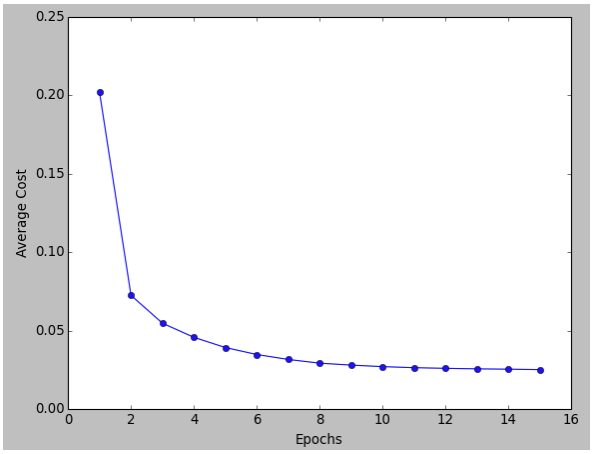
Here, the parameters are updated once that is after all the training examples have been passed through the network. For example, if the training dataset contains 100 training examples then the parameters of the neural network are updated once.

The following equation is iterated over only once.

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Here alpha is learning rate, theta j corresponds to parameters

Batch Gradient Descent is great for convex or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution.



The graph of cost vs epochs is also quite smooth because we are averaging over all the gradients of training data for a single step. The cost keeps on decreasing over the epochs.

Advantages of Batch Gradient Descent

1. **Less oscillations and noisy steps taken towards the global minima** of the loss function due to updating the parameters by computing the average of all the training samples rather than the value of a single sample
2. It **can benefit from the vectorization** which increases the speed of processing all training samples together
3. It produces a **more stable gradient descent convergence** and stable error gradient than stochastic gradient descent
4. It is **computationally efficient** as all computer resources are not being used to process a single sample rather are being used for all training samples

Disadvantages of Batch Gradient Descent

1. Sometimes a stable error gradient can lead to a local minima and **unlike stochastic gradient descent no noisy steps are there to help get out of the local minima**
2. The **entire training set can be too large to process in the memory** due to which additional memory might be needed
3. Depending on computer resources **it can take too long for processing** all the training samples as a batch

---

STOCHASTIC GRADIENT DESCENT

In Batch Gradient Descent we were considering all the examples for every step of Gradient Descent. But what **if our dataset is very huge**. Deep learning models crave for data. The more the data the more chances of a model to be good. Suppose our dataset has 5 million examples, then just to take one step the model will have to calculate the gradients of all the 5 million examples. This does not seem an efficient way. To tackle this problem we have Stochastic Gradient Descent.

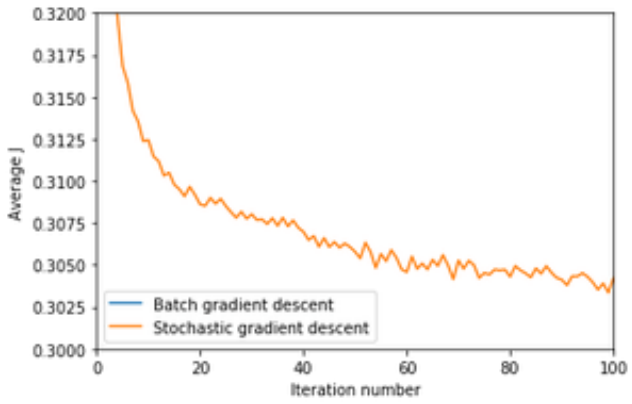
In this method **one training sample (example)** is passed through the neural network at a time and the parameters (weights) of each layer are updated with the computed gradient. So, at a time a single training sample is passed through the network and its corresponding loss is computed. T**he parameters of all the layers of the network are updated after every training sample. For example, if the training set contains 100 samples then the parameters are updated 100 times that is one time after every individual** example is passed through the network. Following is the gradient descent equation and for stochastic gradient descent it is iterated over 'n' times for 'n' training samples in the training set.

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

In Stochastic Gradient Descent (SGD), **we consider just one example at a time to take a single step**. We do the following steps in one epoch for SGD:

1. Take an example
2. Feed it to Neural Network
3. Calculate it's gradient
4. Use the gradient we calculated in step 3 to update the weights
5. **Repeat steps 1–4 for all the examples in training dataset**

Since we are considering just one example at a time the cost will fluctuate over the training examples and it will not necessarily decrease. But in the long run, you will see the cost decreasing with fluctuations.



Also because the cost is so fluctuating, it will never reach the minima but it will keep dancing around it.
SGD can be used for larger datasets. It converges faster when the dataset is large as it causes updates to the parameters more frequently.

Advantages of Stochastic Gradient Descent

1. It is **easier to fit** into memory due to a single training sample being processed by the network
2. It is **computationally fast** as only one sample is processed at a time
3. **For larger datasets it can converge faster** as it causes updates to the parameters more frequently
4. Due to frequent updates the steps taken towards the minima of the loss function have **oscillations which can help getting out of local minimums of the loss function** (in case the computed position turns out to be the local minimum)

Disadvantages of Stochastic Gradient Descent

1. Due to frequent updates the **steps taken towards the minima are very noisy**. This can often lead the gradient descent into other directions.
2. Also, **due to noisy steps it may take longer to achieve convergence to the minima of the loss function**
3. Frequent updates are computationally expensive due to using all resources for processing one training sample at a time
4. It **loses the advantage of vectorized operations** as it deals with only a single example at a time

---

MINI BATCH GRADIENT DESCENT

We have seen the Batch Gradient Descent. We have also seen the Stochastic Gradient Descent. Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large. Batch Gradient Descent converges directly to minima. SGD converges faster for larger datasets. But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations.
To tackle this problem, a **mixture of Batch Gradient Descent and SGD** is used.

This is a mixture of both stochastic and batch gradient descent. The training set is divided into multiple groups called batches. Each batch has a number of training samples in it. At a time a single batch is passed through the network which computes the loss of every sample in the batch and uses their average to update the parameters of the neural network. For example, say the training set has 100 training examples which is divided into 5 batches with each batch containing 20 training examples. This means that following equation will be **iterated over 5 times** (number of batches).

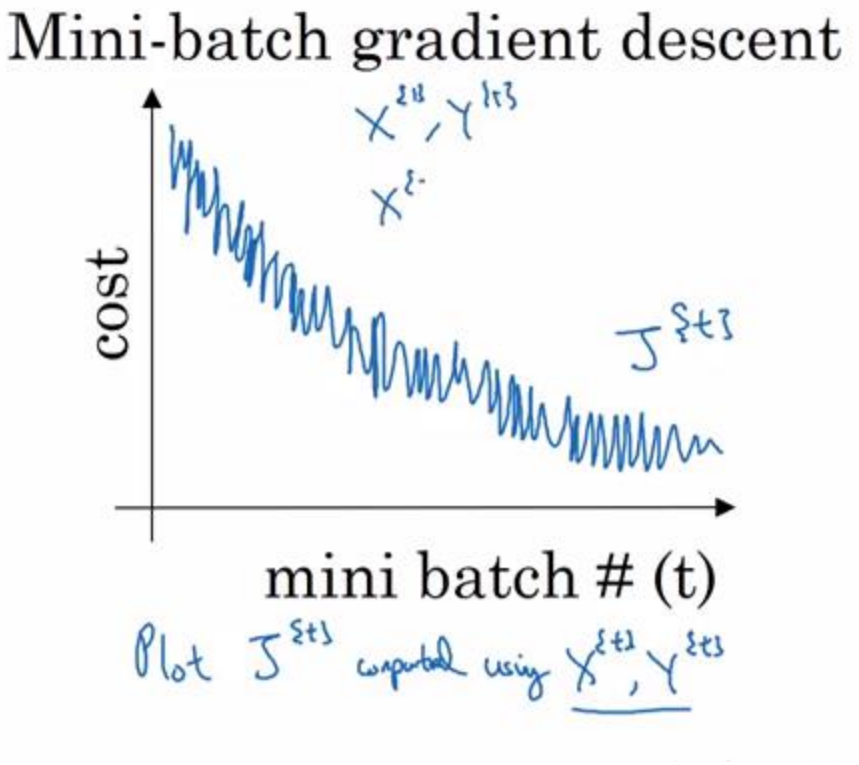
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Here alpha is learning rate, theta j corresponds to parameters

Neither we use all the dataset all at once nor we use the single example at a time. **We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch**. Doing this helps us achieve the advantages of both the former variants we saw. So, after creating the mini-batches of fixed size, we do the following steps in **one epoch**:

1. Pick a mini-batch
2. Feed it to Neural Network
3. Calculate the mean gradient of the mini-batch
4. Use the mean gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for the mini-batches we created

Just like SGD, the average cost over the epochs in mini-batch gradient descent fluctuates because we are averaging a small number of examples at a time.



So, when we are using the mini-batch gradient descent we are updating our parameters frequently as well as we can use vectorized implementation for faster computations.

advantages of both stochastic and batch gradient descent are used due to which Mini Batch Gradient Descent is most commonly used in practice.

1. Easily fits in the memory
2. It is computationally efficient
3. Benefit from vectorization
4. If stuck in local minimums, some noisy steps can lead the way out of them
5. Average of the training samples produces stable error gradients and convergence

Conclusion

Just like every other thing in this world, all the three variants we saw have their advantages as well as disadvantages. It's not like the one variant is used frequently over all the others. Every variant is used uniformly depending on the situation and the context of the problem.

UNDERSTANDING VECTORIZATION on FORWARD & BACKWARD PROPOGATION

Below part is inspired from Andrew Ng deep learning where VECTORIZATION concept I have tried to explain

LOGISTIC REGRESSION

Algorithm is used for classification algorithm of 2 classes.
Equations:

- Simple equation: $y = wx + b$
- If x is a vector: $y = w(transpose)x + b$
- If we need y to be in between 0 and 1 (probability): $y = sigmoid(w(transpose)x + b)$
- In some notations this might be used: $y = sigmoid(w(transpose)x)$
    - While b is $w0$ of w and we add $x0 = 1$. but we won't use this notation in the course (Andrew said that the first notation is better).

In binary classification Y has to be between 0 and 1.
In the last equation w is a vector of Nx and b is a real number

LOGISTIC REGRESSION COST FUNCTION

This is the function that we will use: **`L(y',y) = - (y*log(y') + (1-y)*log(1-y'))`**

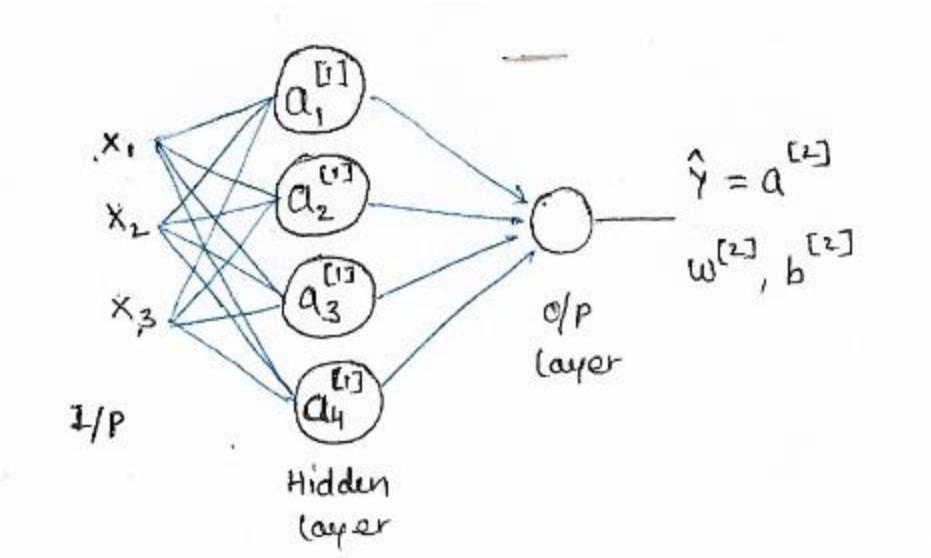
GRADIENT DESCENT

- We want to predict w and b that minimize the cost function.
- Our cost function is convex.
- First, we initialize w and b to 0,0 or initialize them to a random value in the convex function and then try to improve the values the reach minimum value.
- In Logistic regression people always use 0,0 instead of random.
- The gradient decent algorithm repeats: w = w - alpha * dw where alpha is the learning rate and dw is the derivative of w (Change to w) The derivative is also the slope of w
- Looks like greedy algorithms. the derivative give us the direction to improve our parameters.
- The actual equations we will implement:
    - w = w - alpha * d(J(w,b) / dw) (how much the function slopes in the w direction)
    - b = b - alpha * d(J(w,b) / db) (how much the function slopes in the d direction)

VECTORIZATION

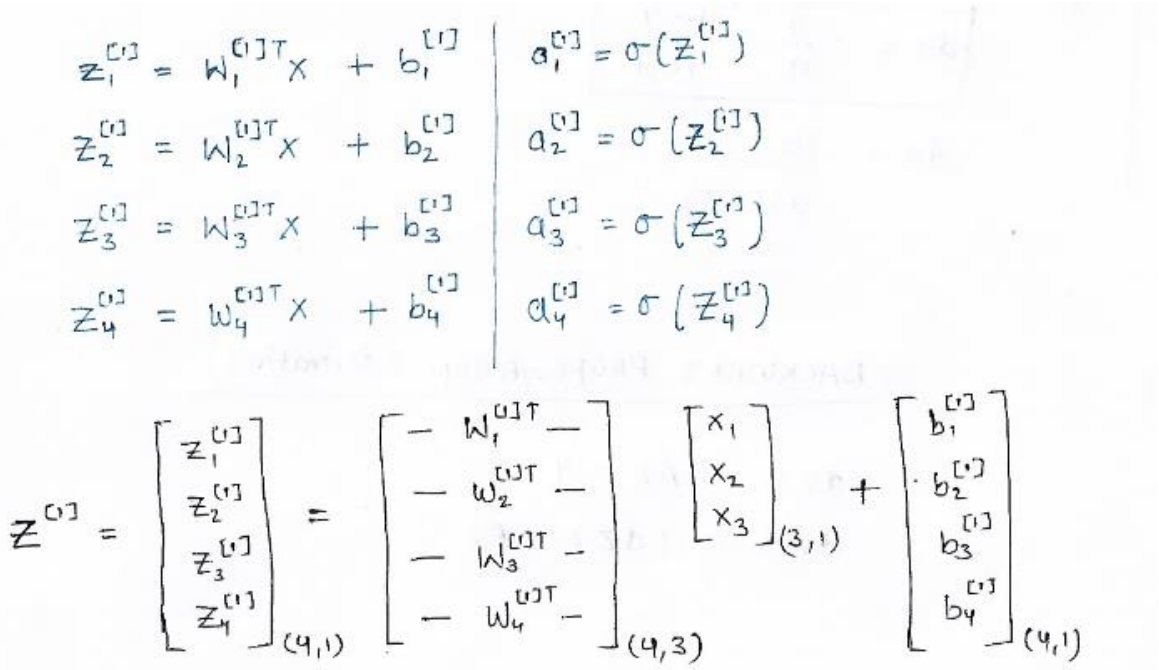
- Vectorization is so important on deep learning to reduce loops. We can make the whole loop in one step using vectorization!
- Deep learning shines when the dataset are big. However, for loops will make you wait a lot for a result. That's why we need vectorization to get rid of some of our for loops.
- 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.

Consider a simple neural network

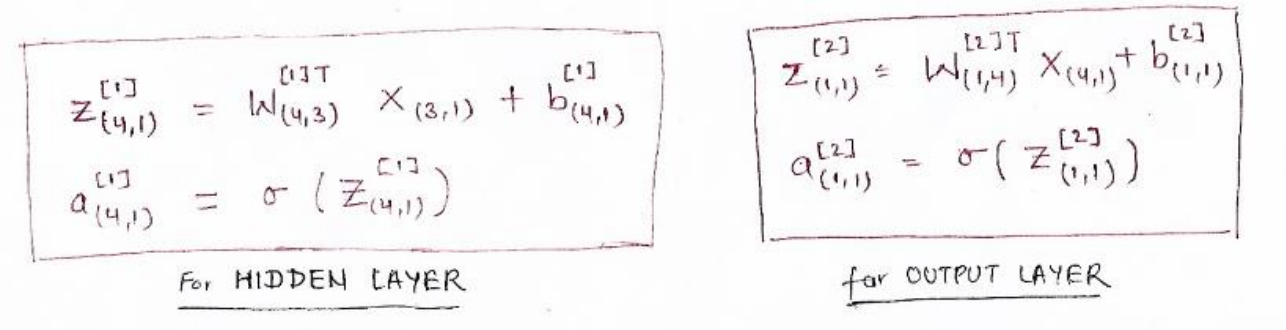


- Here are some informations about the last image:
  - `noOfHiddenNeurons = 4`
  - `Nx = 3`
  - Shapes of the variables:
    - `W1` is the matrix of the first hidden layer, it has a shape of `(noOfHiddenNeurons,nx)`
    - `b1` is the matrix of the first hidden layer, it has a shape of `(noOfHiddenNeurons,1)`
    - `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)`

Now if we first create equation then create matrix



So from above explanation we will try to create generalized equation for hidden and output layers



I hope things are clear till now, so now we will generalize the equation for m multiple

Vectorizing across multiple examples

- Pseudo code for forward propagation for the 2 layers NN:

```
for i = 1 to m
  z[1, i] = W1*x[i] + b1      # shape of z[1, i] is (noOfHiddenNeurons,1)
  a[1, i] = sigmoid(z[1, i])  # shape of a[1, i] is (noOfHiddenNeurons,1)
  z[2, i] = W2*a[1, i] + b2   # shape of z[2, i] is (1,1)
  a[2, i] = sigmoid(z[2, i])  # shape of a[2, i] is (1,1)
```

- Lets say we have `X` on shape `(Nx,m)`. So the new pseudo code:

```
Z1 = W1X + b1     # shape of Z1 (noOfHiddenNeurons,m)
A1 = sigmoid(Z1)  # shape of A1 (noOfHiddenNeurons,m)
Z2 = W2A1 + b2    # shape of Z2 is (1,m)
A2 = sigmoid(Z2)  # shape of A2 is (1,m)
```

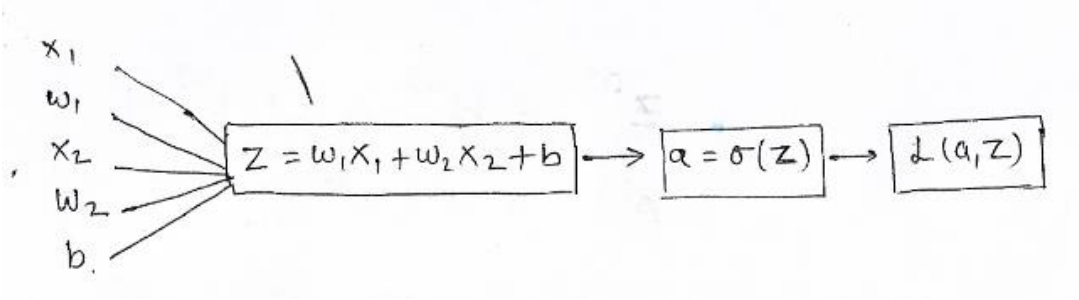
- If you notice always m is the number of columns.
- In the last example we can call `X` = `A0`. So the previous step can be rewritten as:

```
Z1 = W1A0 + b1    # shape of Z1 (noOfHiddenNeurons,m)
A1 = sigmoid(Z1)  # shape of A1 (noOfHiddenNeurons,m)
Z2 = W2A1 + b2    # shape of Z2 is (1,m)
A2 = sigmoid(Z2)  # shape of A2 is (1,m)
```

Now in order to understand BACKPROPAGATION & GRADIENT DESCENT
Let's consider below equation

$$z = w^T x + b$$
$$\hat{y} = a = \sigma(z)$$
$$\mathcal{L}(a, y) = -(y \log(a) + (1 - y) \log(1 - a))$$

For simple neuron example



For above example if we calculate derivative

$$\frac{\partial \mathcal{L}}{\partial w_1} = dw_1 = x_1 dz \qquad dw_2 = x_2 dz \qquad db = dz$$

$$\frac{\partial \mathcal{L}}{\partial z} = dz = \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz}$$

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

$$\boxed{dz = (a - y)}$$

$$\frac{\partial \mathcal{L}}{\partial a} = da$$

$$\boxed{da = -\frac{y}{a} + \frac{1-y}{1-a}}$$

$$da = \frac{(a-y)}{a(1-a)}$$

to get this eq.

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

$$= \frac{(a-y)}{a(1-a)} \cdot a(1-a)$$

$$\boxed{dz = (a - y)}$$

Now from above equation for 2 layer simple neural network equation will become

BACKWARD PROPOGATION

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

$$dw^{[2]} = (dz^{[2]} A^{[1]T}) / m$$

$$db^{[2]} = \text{Sum}(dz^{[2]}) / m$$

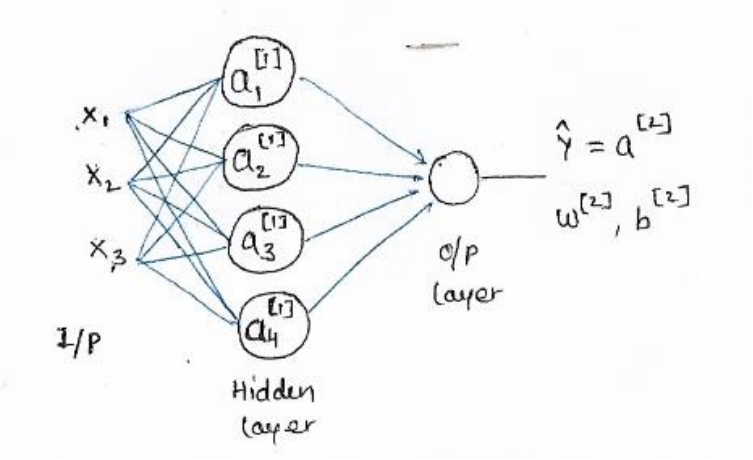
derivative of activation fxn

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

$$dw^{[1]} = (dz^{[1]} A^{[0]T}) / m$$

$$db^{[1]} = \text{sum}(dz^{[1]}) / m$$

So now in last let's try to summarize equation for simple neural network



- Gradient descent algorithm:
  - NN parameters:
    - `n[0] = Nx`
    - `n[1] = NoOfHiddenNeurons`
    - `n[2] = NoOfOutputNeurons = 1`
    - `W1` shape is `(n[1],n[0])`
    - `b1` shape is `(n[1],1)`
    - `W2` shape is `(n[2],n[1])`
    - `b2` shape is `(n[2],1)`
  - Cost function `I = I(W1, b1, W2, b2) = (1/m) * Sum(L(Y,A2))`
  - 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
```

DEEP NEURAL NETWORK

Deep L-layer neural network

- Shallow NN is a NN with one or two layers.
- Deep NN is a NN with three or more layers.
- We will use the notation `L` to denote the number of layers in a NN.
- `n[l]` is the number of neurons in a specific layer `l`.
- `n[0]` denotes the number of neurons input layer. `n[L]` denotes the number of neurons in output layer.
- `g[l]` is the activation function.
- `a[l] = g[l](z[l])`
- `w[l]` weights is used for `z[l]`
- `x = a[0]`, `a[l] = y'`
- These were the notation we will use for deep neural network.
- So we have:
  - A vector `n` of shape `(1, NoOfLayers+1)`
  - A vector `g` of shape `(1, NoOfLayers)`
  - A list of different shapes `w` based on the number of neurons on the previous and the current layer.
  - A list of different shapes `b` based on the number of neurons on the current layer.

Forward and Backward Propagation

- Pseudo code for forward propagation for layer l:

```
Input  A[l-1]
Z[l] = W[l]A[l-1] + b[l]
A[l] = g[l](Z[l])
Output A[l], cache(Z[l])
```

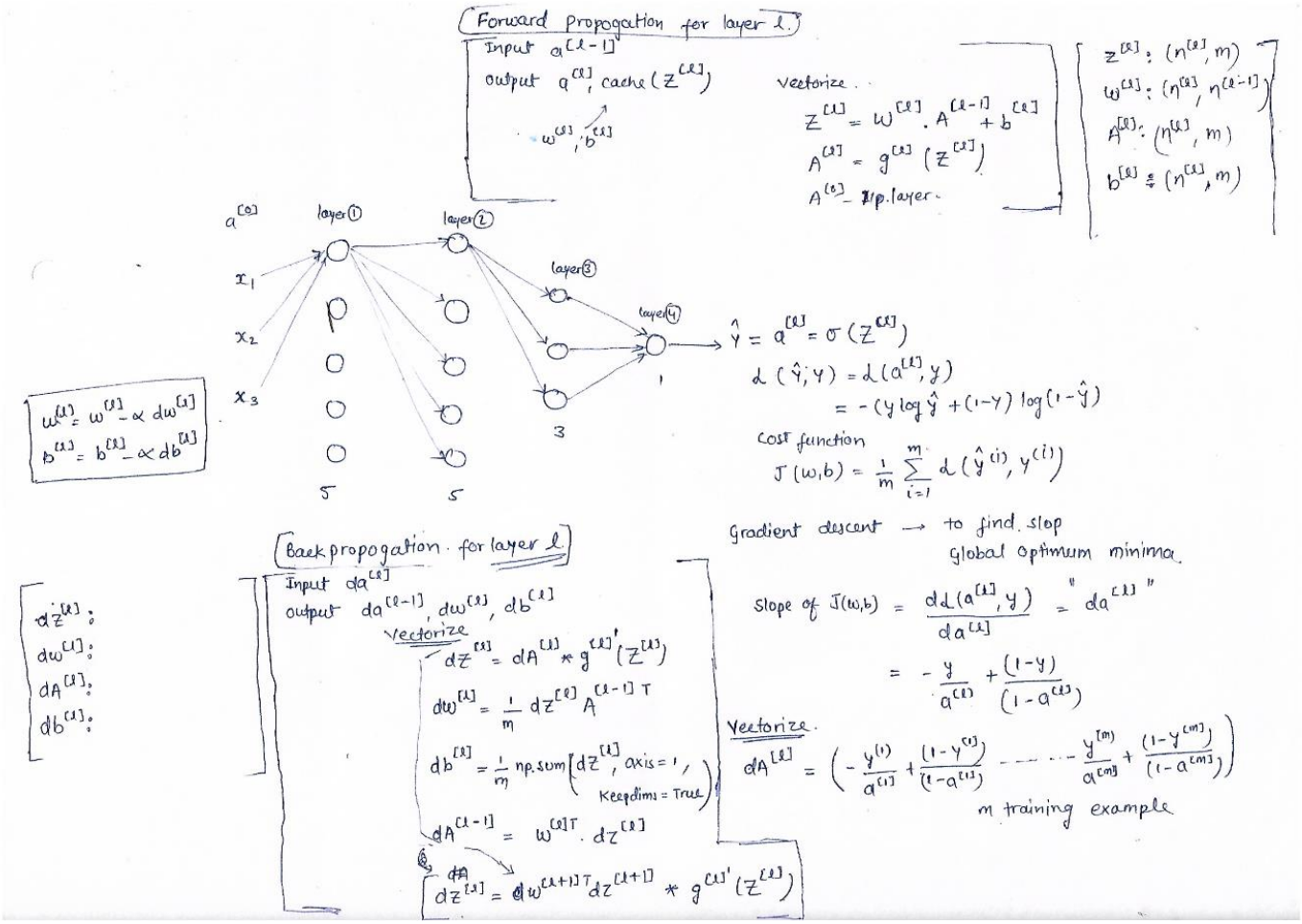
- Pseudo code for back propagation for layer l:

```
Input da[l], Caches
dZ[l] = dA[l] * g'[l](Z[l])
dW[l] = (dZ[l]A[l-1].T) / m
db[l] = sum(dZ[l])/m                # Dont forget axis=1, keepdims=True
dA[l-1] = w[l].T * dZ[l]            # The multiplication here are a dot product.
Output dA[l-1], dW[l], db[l]
```

- If we have used our loss function then:

```
dA[L] = (-(y/a) + ((1-y)/(1-a)))
```

Now for deep neural network I have try to summarize



Parameters vs Hyperparameters

- Main parameters of the NN is W and b
- Hyper parameters (parameters that control the algorithm) are like:
    - Learning rate.
    - Number of iterations.
    - Number of hidden layers L.
    - Number of hidden units n.
    - Choice of activation functions.
- You have to try values yourself of hyper parameters.
- In the earlier days of DL and ML learning rate was often called a parameter, but it really is (and now everybody call it) a hyperparameter.

Reference

https://github.com/mbadry1/DeepLearning.ai-Summary/tree/master/1-%20Neural%20Networks%20and%20Deep%20Learning
Applied AI course
Depplearning.ai
Oneforthlab
https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a
https://medium.com/@divakar_239/stochastic-vs-batch-gradient-descent-8820568eada1
https://www.oreilly.com/library/view/learn-arcore-/9781788830409/e24a657a-a5c6-4ff2-b9ea-9418a7a5d24c.xhtml
https://hackernoon.com/Niranjankumar