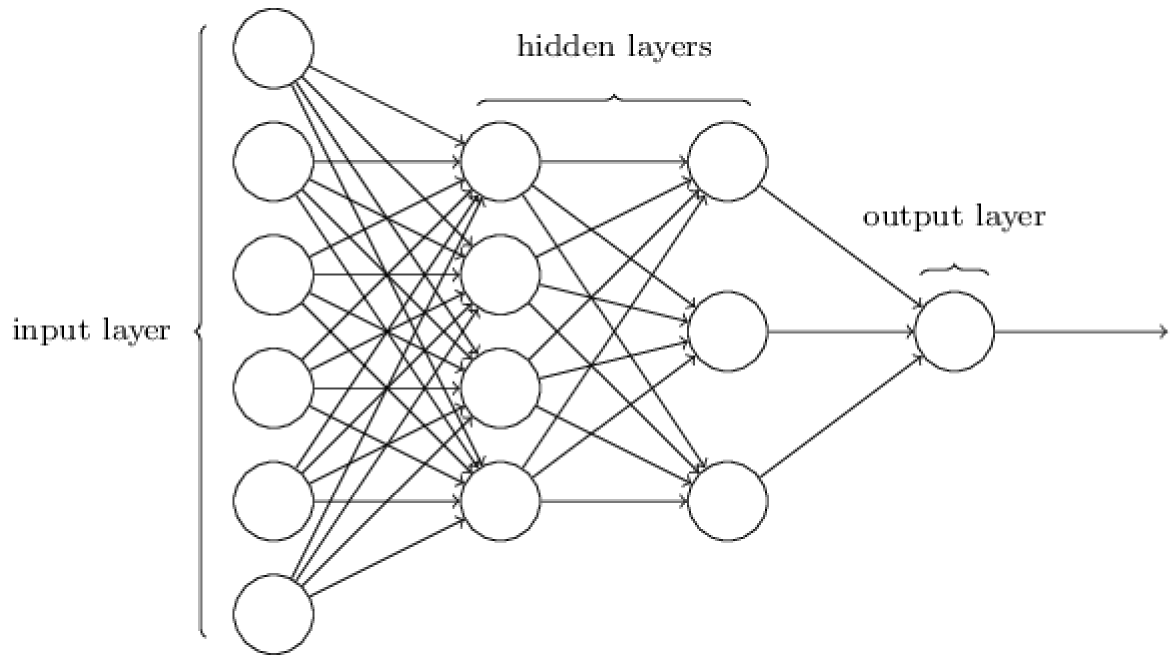
Lecture Notes

Part 1

Neural Networks and Deep Learning

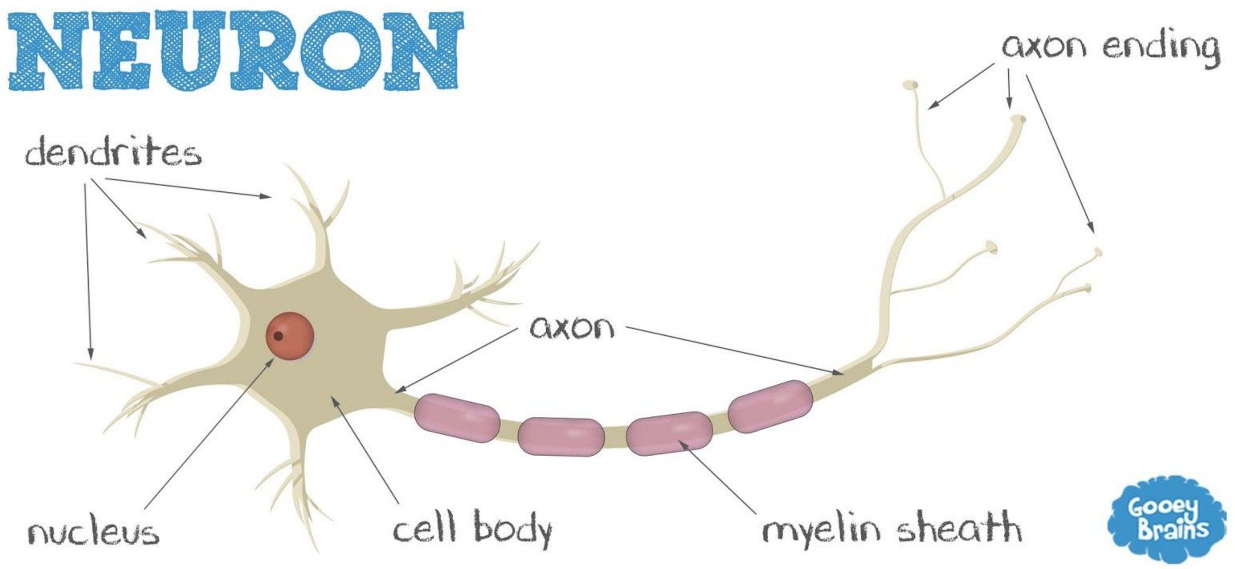


Prof. Ankita Pramanik
IIEST, Shibpur

Siladittya Manna
IIEST, Shibpur

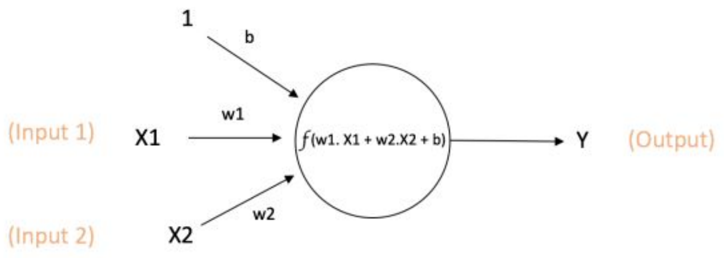
1. What is a Neuron?

The fundamental units of brain and nervous system, responsible for receiving sensory information from different parts of an animal body and transmitting motor command to the muscles.



2. What is a perceptron?

Perceptron is the mathematical model of a biological neuron.
As in the biological neuron, the dendrites receive the electrical signals from the axons of other neurons, the perceptrons receive numerical inputs. The synapses are modeled by incorporating a weight to each input, before the inputs are added. The perceptron also exhibits the firing characteristics of a biological neuron when the input is above a certain threshold, by incorporating an activation function f, which represents the frequency of spikes along the axon.



Output of neuron = Y= f(w1. X1 + w2.X2 + b)

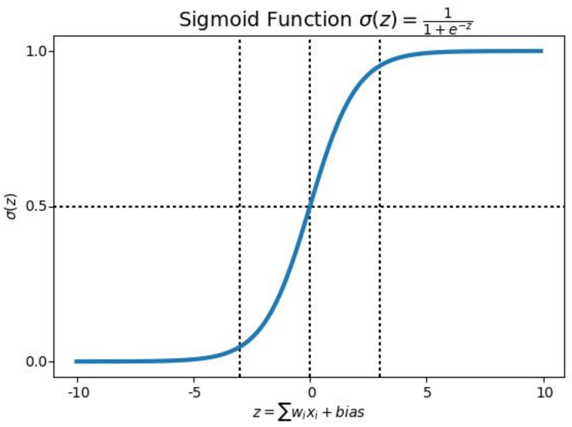
3. What is Neural Network?

Neural Network is computational system, formed by stacking several neurons/perceptrons together, used for supervised and unsupervised learning, and solving complex problems, and does not require to be hard-coded/programmed for any specific task, but works by training itself from given examples.

5. Logistic Regression

Hence, given input $\mathbf{x}$, we want $\hat{y} = P(\mathbf{y}=1|\mathbf{x})$
Input: $\mathbf{x} \in \mathbf{R}^{nx}$ Parameters: $W \in \mathbf{R}^{nx}$, $b \in \mathbf{R}$
Output: $\hat{y} = \sigma(W^T\mathbf{x} + b)$, where $\sigma(z) = 1/(1+e^{-z})$

Hence, $0 \le \hat{y} \le 1$



So the job is to learn the parameters W and b, such that, $\hat{y}$ is close to 1 when z is large and close to 0 when z is very small(negative).

6. Cost Function

To train the Logistic Regression, a cost Function is required.

Given $\{(x^{(i)},y^{(i)})\}$ we want $\hat{y}^{(i)} \approx y^{(i)}$, where $x^{(i)}$ is the ith training example.

Loss Function

Let us assume, the squared error function as the Loss function

$$L(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2$$

But, this loss function is not a good choice for logistic regression, because, the error surface is not always a convex one, and hence, may get stuck on multiple local minimas.

Hence, a more suitable choice for the loss function is

$$L(\hat{y}^{(i)}, y^{(i)}) = -(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})$$

- If $y^{(i)} = 1$: $L(\hat{y}^{(i)}, y^{(i)}) = -\log(\hat{y}^{(i)})$ where $\log(\hat{y}^{(i)})$ and $\hat{y}^{(i)}$ should be close to 1
- If $y^{(i)} = 0$: $L(\hat{y}^{(i)}, y^{(i)}) = -\log(1 - \hat{y}^{(i)})$ where $\log(1 - \hat{y}^{(i)})$ and $\hat{y}^{(i)}$ should be close to 0

The above two points explains the intuitive logic for which the above function is considered as a good choice for the Loss function.

Cost Function

The cost function is the average of the loss function of the entire training set.
The objective is to find the parameters W and b that minimizes the overall cost function

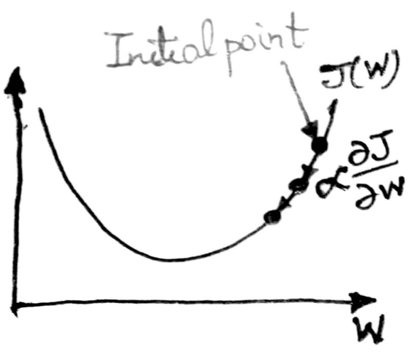
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[(y^{(i)} \log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})]$$

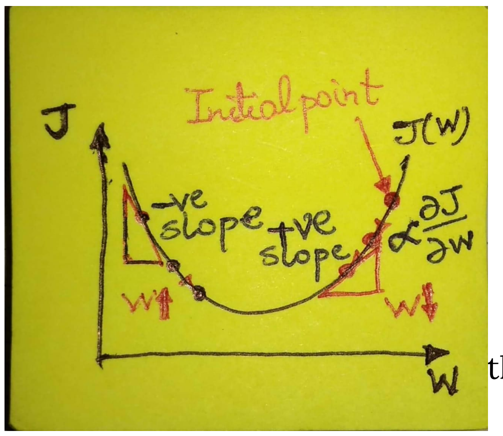
An important to note here is that the loss function computes the error for a single training example; the cost function is the average of the loss functions of the entire training set.

7. **Gradient Descent**

However, to train the neural network, the weights or parameters need to be updated. Otherwise stated, we want to find the values of W and b for which J(W,b) is minimized.

First, W and b needs to be initialized to some values. Usually, in logistic regression, W and b are initialized to 0, however, random initialization is also done.

As 

shown in figure, the 

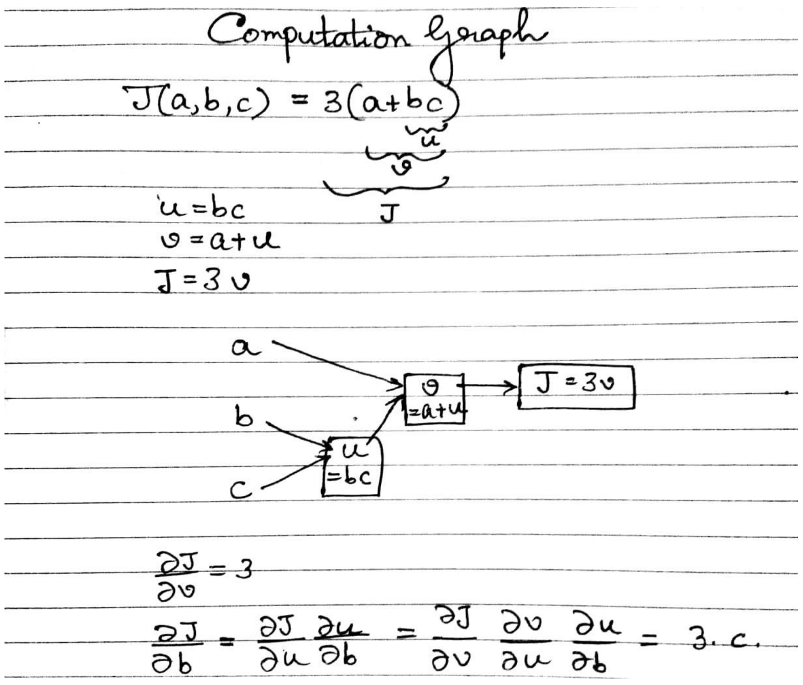
the slope ∂J(W,b)/∂W is negative on the left of the minima and positive on the right of the minima. So, the value of W increases when the point is on the left and increases when on the right.

Update Rule:

W = W − α .∂J(W,b)/∂W   b = b − α .∂J(W,b)/∂b ,where α is the learning rate.

∂J(W,b)/∂W is the derivative of the cost function J(W,b) with respect to W, that is the change of the cost function, in the direction of W.

8. **Computation Graph**



Computation Graph
$J(a,b,c) = 3(\underbrace{a+\underbrace{bc}_{u}}_{v})$ with J under the whole
$u = bc$
$v = a + u$
$J = 3v$
a
b
c
$u = bc$
$v = a + u$
$J = 3v$
$\frac{\partial J}{\partial v} = 3$
$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial u}\frac{\partial u}{\partial b} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial b} = 3.c.$

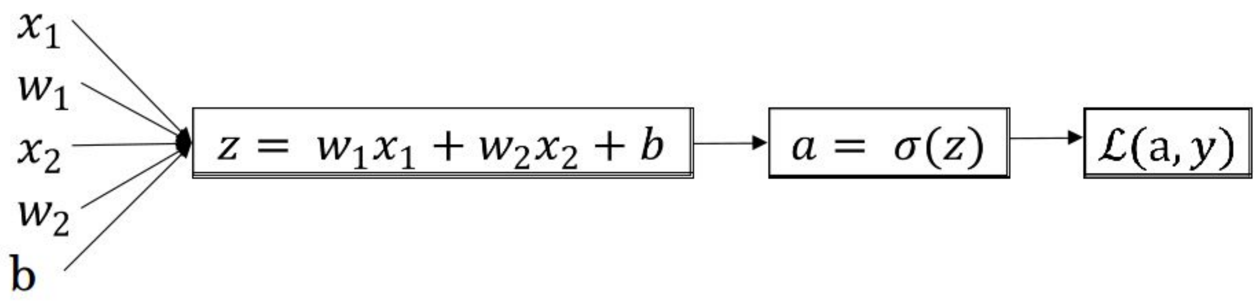
You can see that computing derivatives using computation graph is basically same as **Chain Rule**.

9. **Logistic Regression Gradient Descent**

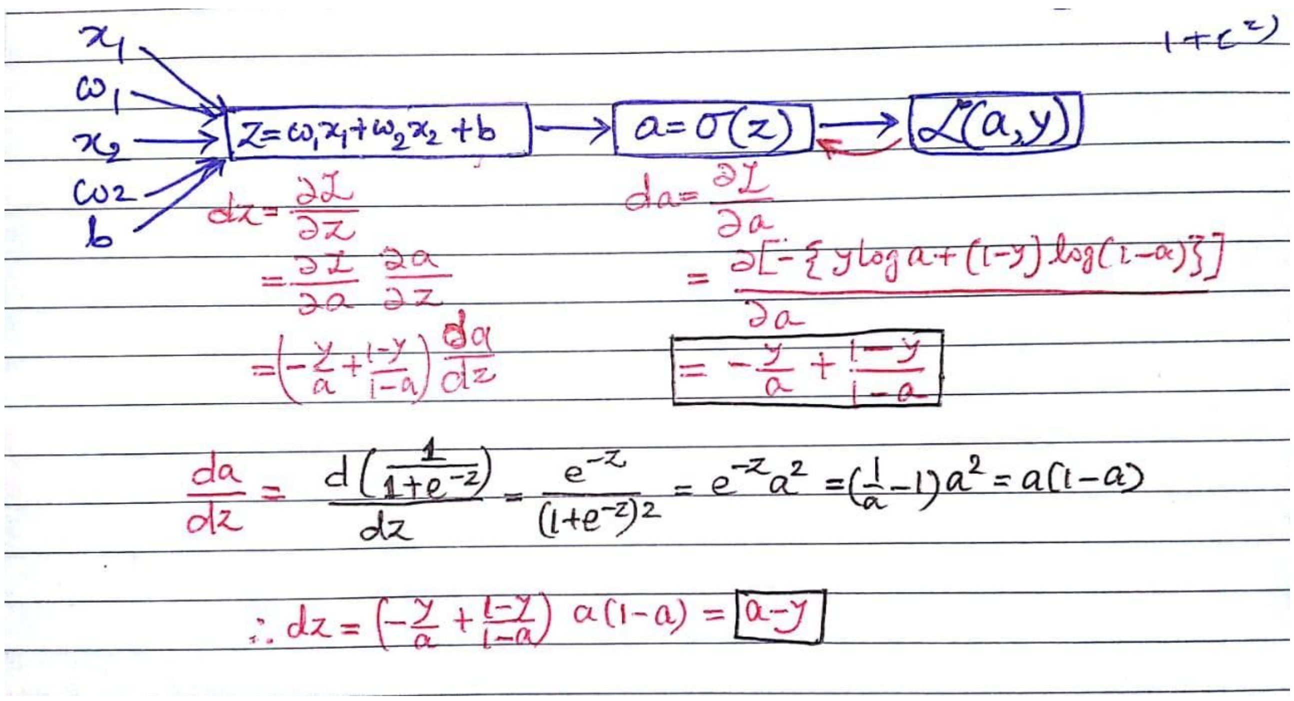
Let, the feature vector **x** contain only two values, i.e. **x** = $[x_1\ x_2]^T$

Now, if we write the operation of the neural network as a computation graph then, in addition to the feature vector, we also need to feed the parameters as input.

The computation graph will look like this:



This shows the result of forward propagation. Now, by applying backward propagation, we can get the change in the values of the parameters



$1+e^{z})$

$$\begin{tikzcd}
x_1 \arrow[rd] & & & \\
w_1 \arrow[r] & \boxed{z = w_1x_1 + w_2x_2 + b} \arrow[r] & \boxed{a = \sigma(z)} \arrow[r] & \boxed{\mathcal{L}(a,y)} \arrow[l, bend left] \\
x_2 \arrow[ru] & & & \\
w_2 \arrow[ruu] & & & \\
b \arrow[ruuu] & & &
\end{tikzcd}$$

$$dz = \frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)\frac{da}{dz}$$

$$da = \frac{\partial \mathcal{L}}{\partial a} = \frac{\partial\left[-\{y\log a + (1-y)\log(1-a)\}\right]}{\partial a}$$

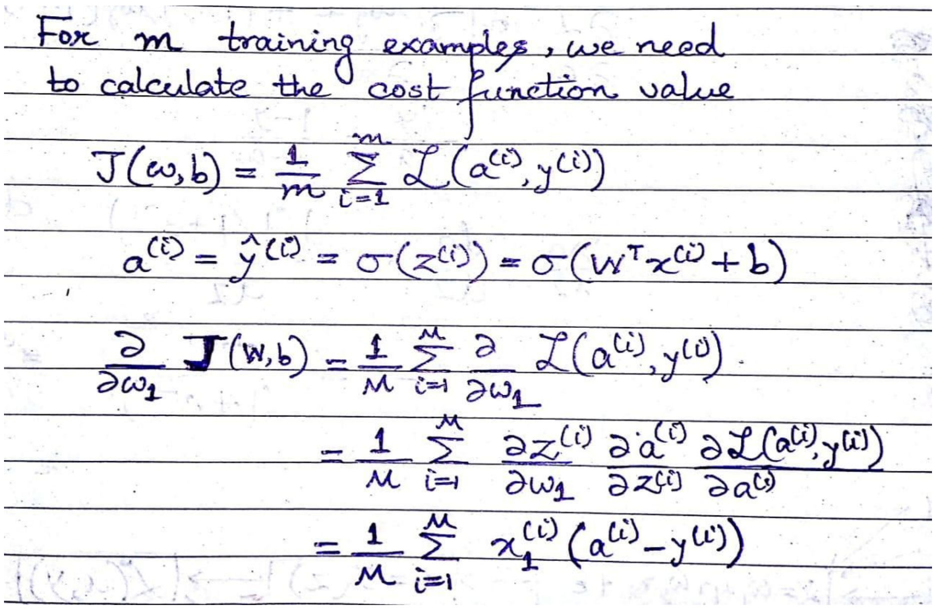
$$\boxed{= -\frac{y}{a} + \frac{1-y}{1-a}}$$

$$\frac{da}{dz} = \frac{d\left(\frac{1}{1+e^{-z}}\right)}{dz} = \frac{e^{-z}}{(1+e^{-z})^2} = e^{-z}a^2 = \left(\frac{1}{a}-1\right)a^2 = a(1-a)$$

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

Here, a is the output of the activation function **sigmoid** and y is the true label of the data.

10. **Logistic Regression on m examples**



For m training examples, we need to calculate the cost function value

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$$

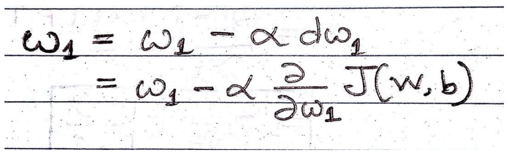
$$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(W^T x^{(i)} + b)$$

$$\frac{\partial}{\partial w_1} J(W,b) = \frac{1}{M}\sum_{i=1}^{M} \frac{\partial}{\partial w_1}\mathcal{L}(a^{(i)}, y^{(i)})$$

$$= \frac{1}{M}\sum_{i=1}^{M} \frac{\partial z^{(i)}}{\partial w_1}\frac{\partial a^{(i)}}{\partial z^{(i)}}\frac{\partial \mathcal{L}(a^{(i)}, y^{(i)})}{\partial a^{(i)}}$$

$$= \frac{1}{M}\sum_{i=1}^{M} x_1^{(i)}(a^{(i)} - y^{(i)})$$

Update Rule:



$$w_1 = w_1 - \alpha\, dw_1$$
$$= w_1 - \alpha \frac{\partial}{\partial w_1} J(w,b)$$

Now, if we implement this using a for loop in any programming language, the code will not be efficient, because as the dataset increases, the neural network will take more iterations to train. Hence, computational time will be huge.

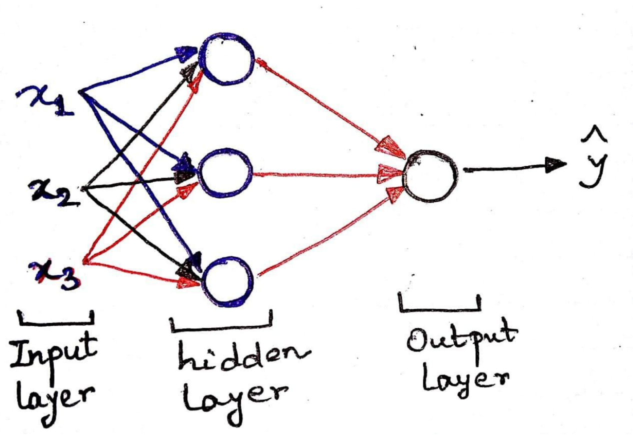
Hence, here comes the concept of vectorization.

11. **Vectorization**

Vectorization will be discussed, during **Part 3.a.**

12. **Neural Network Representation**

Neural Network with a single hidden layer



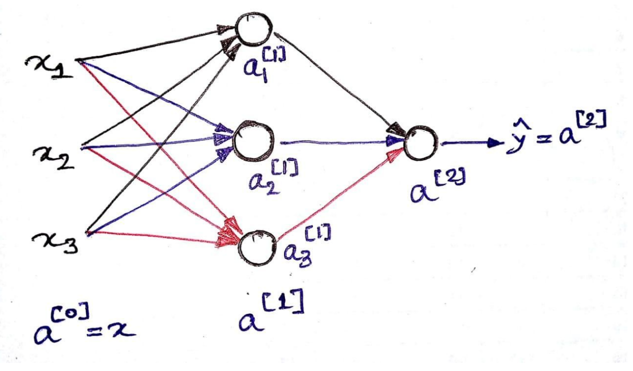
This is a 2-layer neural network (because the input layer is not counted)

The term **"Hidden Layer"** refers to the fact that in the training set, the true values of the nodes in these layers are not observed in the training set, hence they are caled hidden layers. The function of the Hidden Layer is to take the input and apply a linear transformation to the input, followed by a squashing non-linearity. However, we have values for the input layers and the output label in the training set.

The letter "**a**" refers to the output of a layer which is passed on to the next layer as input.

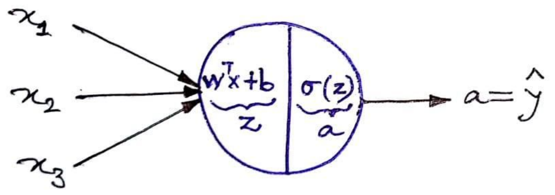
The input layer passes on the input to the first hidden layer, hence the input layer is termed as the 0th layer **$a^{[0]}$**. **The output of the first hidden layer is denoted by $a^{[1]}$**. For a hidden layer with ***n*** nodes, the activation generated is

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ \vdots \\ a_n^{[1]} \end{bmatrix}$$

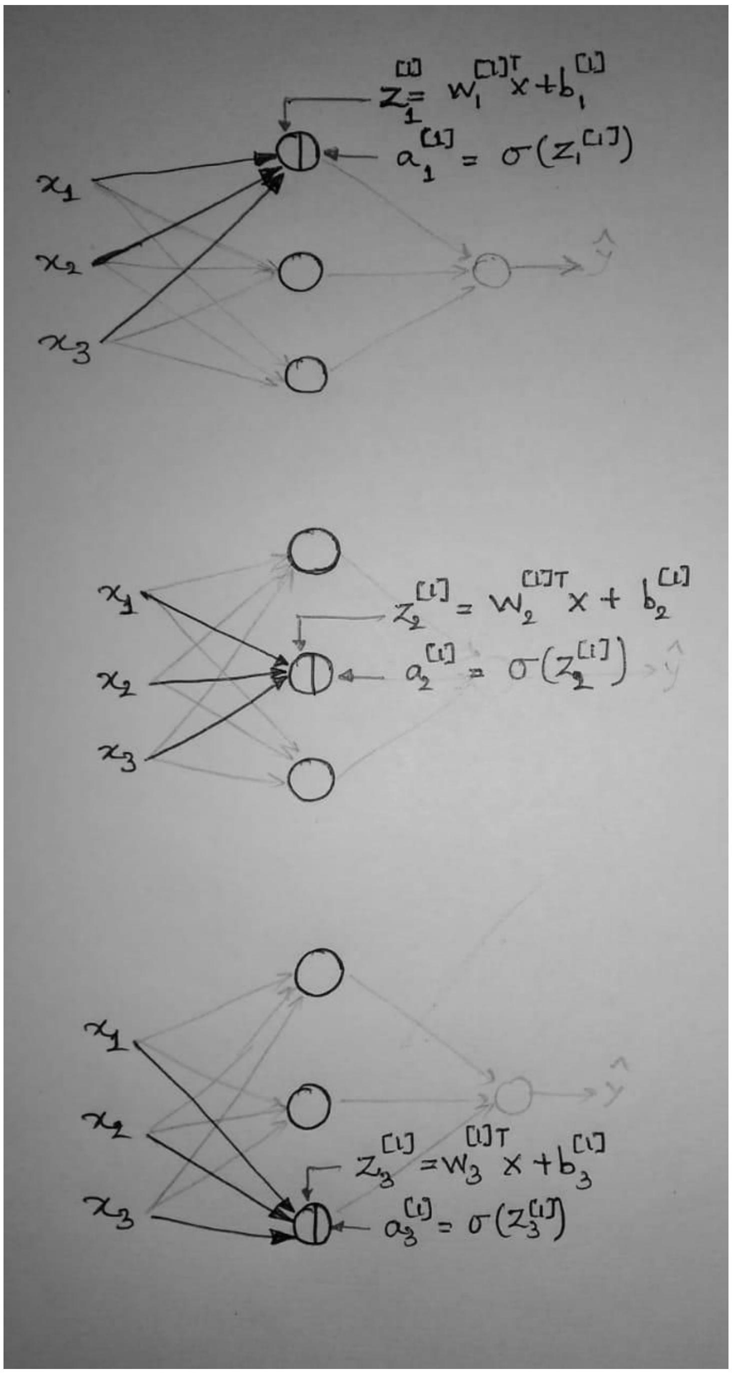


13. **Computing a Neural Network’s Output**

Two steps of Computation in a perceptron

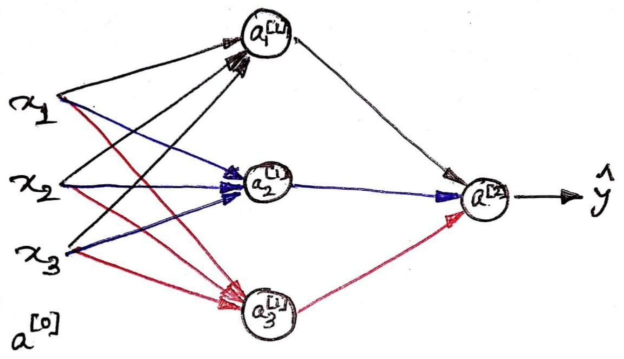


$$z = w^T x + b$$
$$a = \sigma(z)$$



$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$
$a_1^{[1]} = \sigma(z_1^{[1]})$
x_1
x_2
x_3
$\hat{y}$
$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$
$a_2^{[1]} = \sigma(z_2^{[1]})$
x_1
x_2
x_3
$\hat{y}$
x_1
x_2
x_3
$\hat{y}$
$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}$
$a_3^{[1]} = \sigma(z_3^{[1]})$

Writing all the equations together,



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

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

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

The equatins can be written as

$$\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix} = \underbrace{\begin{bmatrix} \text{---} w_1^{[1]T} \text{---} \\ \text{---} w_2^{[1]T} \text{---} \\ \text{---} w_3^{[1]T} \text{---} \end{bmatrix}}_{W^{[1]} \, (n_{a^{[1]}}, n_{a^{[0]}})} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix}}_{b^{[1]} \, (n_{a^{[1]}}, 1)}$$

$$Z^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix} ; \; a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix} = \sigma(Z^{[1]})$$

The size of the resulting weight matrix W is thus ($n_a^{[0]}$, $n_a^{[1]}$) and that of b is ($n_a^{[0]}$,1)

Thus given input ***x***

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

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

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

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

For n-layer neural net

$z^{[n]} = W^{[n]}a^{[n-1]}+b^{[n]}$

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

13. Vectorizing across multiple examples

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

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

For m examples,

$X = [x^{[1]}\ x^{[2]} \dots x^{[m]}]$

Size of **X** is ($\boldsymbol{n_x}$ X **m**)

Hence, $\mathbf{Z^{[1]}}$ becomes of size **($\mathbf{n_{a[1]}}$ X m)**, and so does $\mathbf{A^{[1]}}$

Hence at the input of the second layer the input is of the **s**ize $\mathbf{n_{a[1]}}$ X **m**

For the second layer

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

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

Size of $W^{[2]}$ is **($\mathbf{n_{a[2]}}$ X $\mathbf{n_{a[1]}}$)**

Hence, size of $Z^{[2]}$ is **($\mathbf{n_{a[2]}}$ X m)** and so is for $A^{[2]}$.

13. **Activation Functions**

Activation Functions are a very important part of neural nets. It is responsible for the non-linear mapping between the input and the output. Without the activation function, the Neural Network would simply be a linear function of degree 1. Hence, the neural network would be unable to learn complex functional relations between the input and the target. Therefore, we need to apply the non-linear activation functions, so as to empower the neural network to learn the non-linear complex functional mappings that exists between the inputs and outputs, and represent it too.

Furthermore, the activation functions should be differentiable, otherwise, back-propagation cannot be performed.

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

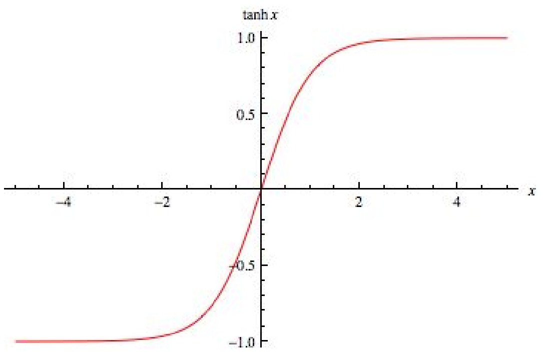
$a^{[1]} = g(\mathbf{z^{[1]}})$

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

$a^{[2]} = g(z^{[2]})$

In more general case, we consider g to be the activation function

The activation function which almost always works better than the sigmoid function is the tanh function.



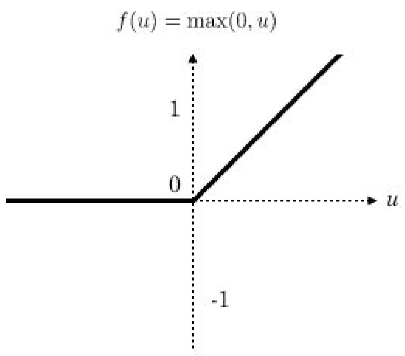
tanh(z) = $(e^{z}-e^{-z})/(e^{z}+e^{-z})$

This function has the effect of centering the data nearer to 0 mean, and it makes learning a little bit easier, than in sigmoid, in which the data gets centered at 0.5.

However, for the output layer, the activation function is preferably sigmoid, because we want the output label to be in [0,1].

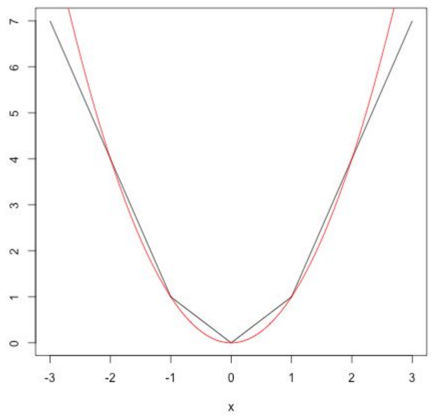
Hence, the activation function can be different for different layers depending on the function.

Rectified Linear Unit(ReLU): It is the most commonly used activation function.



ReLU captures Interactions and Non-Linearities.As more layers and nodes are added, the complexity of the interactions increases.

However, it may seem that, the ReLu function is non-linear only at **x** = 0. Since, there are many nodes in a layer, and also different bias for different nodes, the slope changes at many places along the **x** axis giving rise to non-linearity.



There are some disadvantages of these activation functions.

Firstly, for **tanh** function, the non-linear part of the tanh function lies roughly between -2.0 to 2.0 and the rest of the function is relatively flat. So, the change in the output will not be well reflected in the input. The derivative of the **tanh** function beyond this narrow range, is almost zero, which makes update of weights very difficult. This problem gets worse, of the number of layers increases. This problem is termed as **vanishing gradient problem**.

ReLU however rectifies vanishing gradient problem.

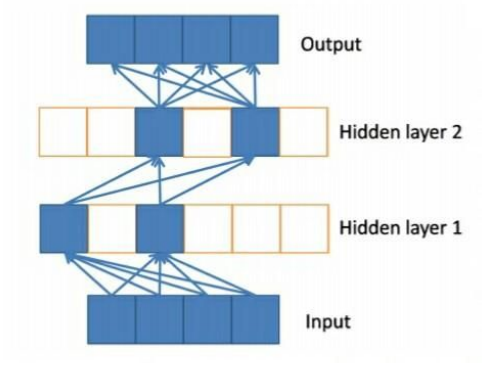
In a multilayer network, during back propagation, the derivatives are gradually multiplied by more and more terms arising from the derivatives of that layer. If tanh or sigmoid functions are used then the updates becomes close to zero, hence we get a vanishing gradient.

However, if we use ReLU, then the derivative of ReLU is always 1, if $\mathbf{x} > 0$, i.e. the activation is positive. That is, the error signal is fully propagated to the input, and the gradient value does not decreases as we move up the network layers. Thus the vanishing gradient problem is resolved. ReLU is actually more biologically plausible and enables the network to obtain sparse representations.

Other advantages of ReLU:

Network easily obtains sparse representation. Refer to this link for the advantages of sparsity introduced by ReLU. The only, non-linearity that comes due to ReLU, is due to the path selection associated with the individual active neurons.

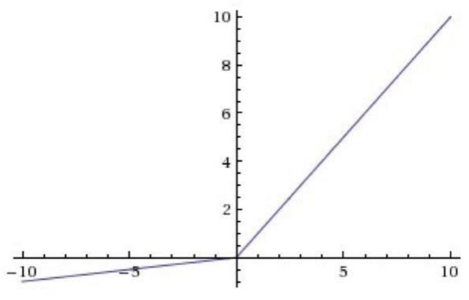
The function obtained by using ReLU as an activation function in the hidden layers, is linear by parts, due to which the gradient flows well on the active paths of the neurons.



ReLU however has a **disadvantage** too. Since ReLU gives zero output for inputs less than zero, the gradient can go to zero, resulting in the weights not being updated. Hence, neurons stop responding to changes in error/input. Thus, it can make a neuron never to activate again, and thus making a D**ead Neuron.** To fix this problem **Leaky ReLU** is used.

Leaky* ReLU**: ***f(***x***) = ***x*** , ***x*** > 0

= 0.01***x***, ***x*** < 0



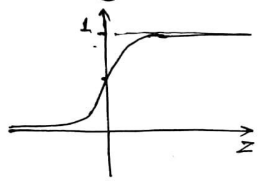
***Parametric* ReLU: *f(x)* = *x*; *x* > 0**

= *a.x*; *x* < 0

The parameter a is trainable in PReLU.

Derivatives of activation Functions

Sigmoid

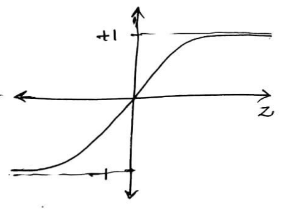


$$g(z) = \frac{1}{1+e^{-z}}$$

$$\frac{dg}{dz} = \frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right)$$

$$g'(z) = g(z)\,(1-g(z)).$$

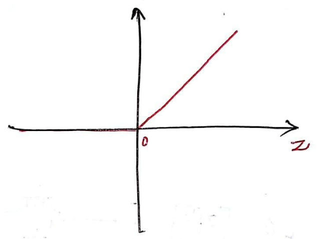
Tanh



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

$$g'(z) = 1-(\tanh(z))^2$$

ReLU



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

$$g'(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

Gradient Descent for Neural Networks

Forward Propagation

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

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

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

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

Back Propagation

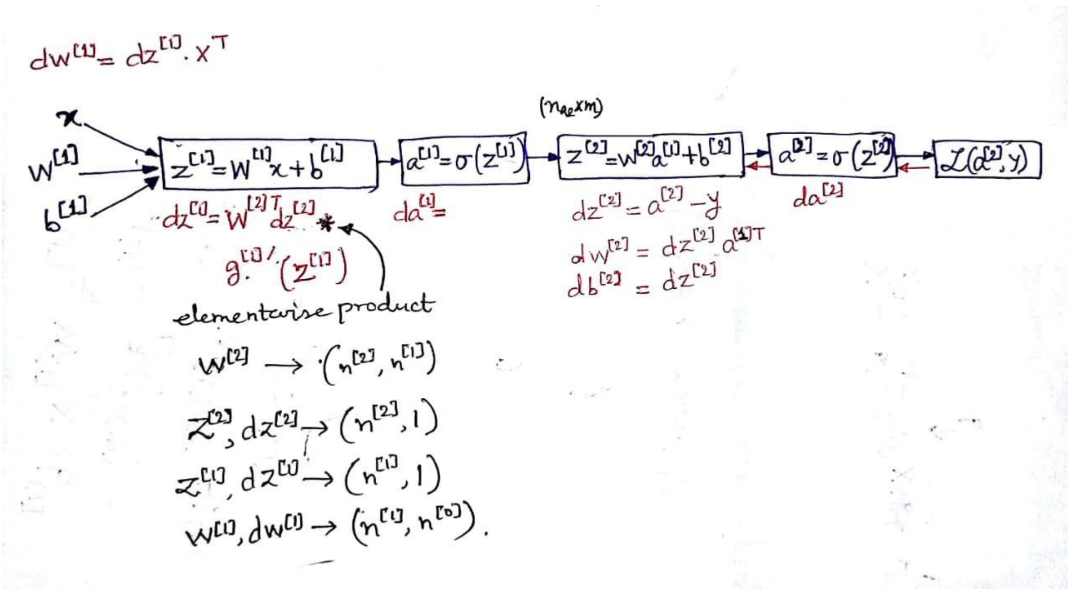
$$dz^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{M}\, dz^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{M}\left[\begin{array}{l}\text{summation of } dZ^{[2]} \\ \text{over axis} = 1\end{array}\right].$$

$$dZ^{[1]} = \frac{\partial \mathcal{L}}{\partial z^{[1]}} = W^{[2]T} dz^{[2]} g'(z^{[1]})$$

Backpropagation Explanation using Computation Graph



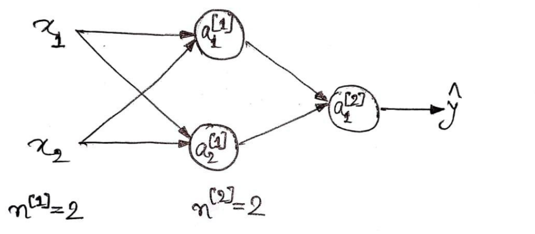
$dw^{[1]} = dz^{[1]} . x^T$
x
$W^{[1]}$
$b^{[1]}$
$z^{[1]} = W^{[1]}x + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$(n_{a_2} \times m)$
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$
$\mathcal{L}(a^{[2]}, y)$
$dz^{[1]} = W^{[2]T} dz^{[2]} *$
$g^{[1]\prime}(z^{[1]})$
elementwise product
$da^{[1]} =$
$dz^{[2]} = a^{[2]} - y$
$dw^{[2]} = dz^{[2]} a^{[1]T}$
$db^{[2]} = dz^{[2]}$
$da^{[2]}$
$W^{[2]} \rightarrow (n^{[2]}, n^{[1]})$
$z^{[2]}, dz^{[2]} \rightarrow (n^{[2]}, 1)$
$z^{[1]}, dz^{[1]} \rightarrow (n^{[1]}, 1)$
$W^{[1]}, dW^{[1]} \rightarrow (n^{[1]}, n^{[0]})$.

$$da^{[1]} = \frac{\partial L}{\partial a^{[1]}} = \frac{\partial L}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} = (a^{[2]} - y)W^{[2]}$$

To match the dimension of dW$^{[2]}$ with the RHS, a$^{[1]T}$ is taken.

Weight Initialization

1. **Zero Initialization**



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

The problem with zero initialization is that, a$_1^{[1]}$ and a$_2^{[1]}$ will be equal. Because both of the hidden units will be computing the same function. Also, during backpropagation, dz$_1^{[1]}$ and dz$_2^{[1]}$ will also be same, due to symmetry. That is both the hidden units are computing the same function. So, even if weight is updated,

the error signal propagating back through the network will also be same for both nodes.

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

Hence, the updated weights: W$^{[1]}$ = W$^{[1]}$ – α dW

Hence, the updated weights will be same for the hidden units. Hence, even after several iterations, it will be computing the same function.

This happens if the weights are initialized to the same value, other than zero.

No matter what was the input - if all weights are the same, all units in hidden layer will be the same too. In other words, the neural network, fails to break symmetry.

The solution to this is to initialize the weights randomly.

2. **Xavier Initialization**

Now for random initialization, if the weights are too small then the variance of the input signal propagating through the network, starts decreasing and if the weights are too large then the variance increases rapidly. Eventually it becomes very large and becomes useless. This is mainly observed, if sigmoid function is used.

Now, from the computation graph we have seen that

$dz^{[i]} = W^{[i+1]} dz^{[i+1]} * g'(z^{[i]})$

$dW^{[i]} = dz^{[i]} a^{[i]T}$

If $\boldsymbol{n_i}$ be the size of the layer $\boldsymbol{i}$ and $\boldsymbol{x}$ be the layer input, then

$g'(z^{[i]}) \approx 1$

$Var[z^{[i]}] = Var[x] \prod_{0}^{i-1} n_{i'} Var[W^{i'}]$

$Var[W^{i'}]$ = shared scalar variance of all weights at layer i'

Then for a network with $\boldsymbol{d}$ layers

$Var[dz^{[i]}] = Var[dz^{[d]}] \prod_{i}^{d} n_{i'+1} Var[W^{i'}]$

$Var[dW^{[i]}] = \prod_{i'=0}^{i-1} n_{i'} Var[W^{i'}] \prod_{i'=i}^{d-1} n_{i'+1} Var[W^{i'}] \times Var[x] Var[dz^{[d]}]$

From forward propagation point of view, to keep information flowing,

$\forall (i,i')\ Var[z^{[i]}] = Var[z^{[i']}]$

From Back propagation point of view,

$\forall (i,i')\ Var[dz^{[i]}] = Var[dz^{[i']}]$

These two conditions transform to

$\forall \boldsymbol{i},\ n_i \text{Var}[W^{[i]}] = 1$ and $\boldsymbol{n_{i+1}} \text{Var}[W^i] = 1$

Combining these two constraints, we get

$$\forall \boldsymbol{i},\ \text{Var}[W^{[i]}] = \frac{2}{n_i + n_{i+1}}$$

However, the variance of the back-propagated gradient may still vanish or explode in deep networks.

Thus, the normalization factor is important in deep networks because of the multiplicative effect of the layers.

In their paper, Xavier Glorot et al. Suggests that the weights be initialized(**Normalized initialization**) with weights

such that

$$W \sim U\left[\frac{-\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

Ref.: Understanding the difficulty of training deep feedforward neural networks

3. **He Initialization**

Xavier Initialization was mainly based on the assumption that the activation functions are linear and works good for sigmoid activation function.

Later on, ReLU surpassed Sigmoid, and solved the problem of vanishing and exploding gradient. Hence, came a new initialization procedure proposed by Kaiming He et al and is referred to as He Initialization. This initialization method allows very deep models to converge while Xavier initialization does not.

The derivation follows the procedure as given in the paper by Xavier et al.

Forward Propagation case

For a layer, $\mathbf{z}^{[i]} = \mathbf{W}^{[i]}\ \mathbf{x} + \mathbf{b}^{[i]}$

$\mathbf{x} = f(\mathbf{z}^{[i-1]})$

The initialized elements in **W** are assumed to be mutually independent and share the same distribution. Furthermore, it is assumed that the elements in **x** are also mutually independent and share the same distribution and **x** and **W** are independent of each other.

Then we have:

$\text{Var}[\boldsymbol{z}] = \boldsymbol{n}\ \text{Var}[\boldsymbol{w}\ \boldsymbol{x}]$

$\boldsymbol{z}, \boldsymbol{w}, \boldsymbol{x}$ represents random variables in **z**,**W** and **x** respectively..

Var[$\boldsymbol{z}$] = n.Var[$\boldsymbol{w}$] E[$\boldsymbol{x}^2$]

For ReLU

E[$\boldsymbol{x}^2$] = $\int_{-\infty}^{+\infty} max(0, z^{[i-1]})^2 p(z) dz$

Since, in ReLU, the part of the ReLU for z<0 will not contribute to the integral

E[$\boldsymbol{x}^2$] = $\int_{0}^{+\infty} (z^{[i-1]})^2 p(z) dz$

The above equation can be written as half of the integral over the entire real domain (as $(z^{[i-1]})^2$ is symmetric around 0 and p(z) is assumed to be symmetric around 0)

E[$\boldsymbol{x}^2$] = $\frac{1}{2}\int_{-\infty}^{+\infty} (z^{[i-1]})^2 p(z) dz$

Since z[i-1] is assumed to have zero mean, hence,

E[$\boldsymbol{x}^2$] = $\frac{1}{2}\int_{-\infty}^{+\infty} (z^{[i-1]} - E[z^{[i-1]}])^2 p(z) dz = \frac{1}{2} Var[z^{[i-1]}]$

Therefore,

Var[$\boldsymbol{z}^{[\mathbf{i}]}$] = $\frac{1}{2} n Var[w^{[i]}] Var[z^{[i-1]}]$

With **L** layers put together,

Var[$z^{[\mathbf{L}]}$] = Var[$y^{[\mathbf{1}]}$] $(\prod_{i=2}^{L} \frac{1}{2} n^{[i]} Var[w^{[i]}])$

A proper initialization should avoid reducing or magnifying the magnitudes of input signals exponentially. So, the above product should be a scalar(e.g. 1).

A sufficient condition is

$$\frac{1}{2} n^{[i]} Var[w^{[i]}] = 1 \forall\ i$$

This results in a zero mean Gaussian distribution with standard deviation $\sqrt{\frac{2}{n^{[i]}}}$

This is the way of initialization as mentioned in the paper by He et al.. They also initialized **b** = 0. and the input layer with weights $w^{[1]}$, such that $z^{[1]}$ Var[$w^{[1]}$] = 1

But the factor (1/2) does not matter if it just exists on one layer.

Backward propagation Case.

$da^{[i]} = W^{[i]T} dz^{[i+1]}$

As in the forward propagation case, we consider, W and $dz^{[i+1]}$ to be independent of each other, then $dz^{[i]}$ has zero mean for all ***i*****,** when ***w*** is initialized by a symmetric distribution around zero**.**

We also have,

$dz^{[i]} = g'(z^{[i]})\, da^{[i]}$, where g' is the derivative of g

For ReLU, derivative is either 1 or 0, with equal probability.

Assuming $dz^{[i]}$ and $g'(z^{[i]})$ are independent of each other.

We have, $E[dz^{[i]}] = E[da^{[i]}]/2 = 0$ and also $E[(dz^{[i]})^2] = Var[dz^{[i]}] = (1/2)Var[da^{[i]}]$

Thus, $Var[da^{[i]}] = n_i Var[w^{[i]}]Var[dz^{[i]}] = (1/2)n_i Var[w^{[i]}]Var[da^{[i]}]$

Though the derivations are different, the factor (1/2) is the result of using ReLU.

Putting together ***L*** layers,

$$Var[da^{[2]}] = Var[da^{[L+1]}] \left(\prod_{i=2}^{L} \frac{1}{2} n^{[i]} Var[w^{[i]}]\right)$$

To prevent the gradient from getting exponentially large or small

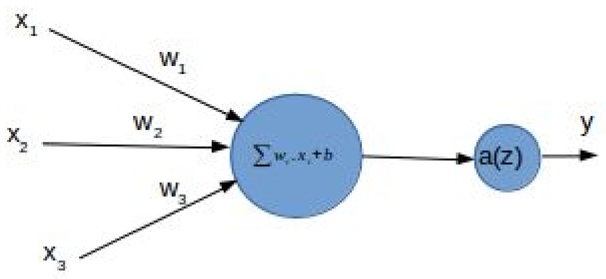
$$\frac{1}{2} n^{[i]} Var[w^{[i]}] = 1, \forall\, i$$

The above equation results in a Gaussian Distribution with zero mean and standard deviation $\sqrt{\frac{2}{n^{[i]}}}$

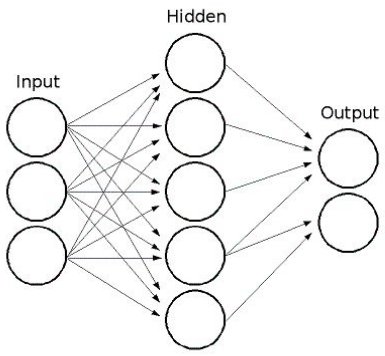
The main difference with the Xavier initialization is that, the He initialization addresses the rectifier nonlinearities. For very deep networks, the Xavier initialization will stall learning, while the He initialization will be able to make the model converge.

Ref.: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

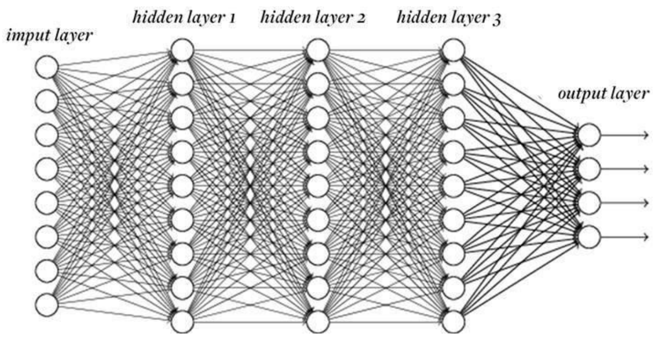
What is a Deep Neural Network?



This is a Logistic Regression However, this model cannot represent complex functions Hence, we need to add more number of neurons.



This is a neural network with 1 hidden layer. Not too deep, eh?



This one is 4 layers deep.

The depth of a neural network depends on how complex the function it is trying to learn.

If the function is too complex, then the number of neurons and layers, should be more. Otherwise, it will not be able to learn the function properly.

So, for practical purposes, you should start from logistic regression or shallow network and then gradually increase the number of neurons/layers, and observe the accuracy.

Depth of the neural network, depends on the objective.

However, increasing depth causes increase in the number of parameters, which the neural network needs to train. This causes increase in space requirement, and also time to train the network also increases.

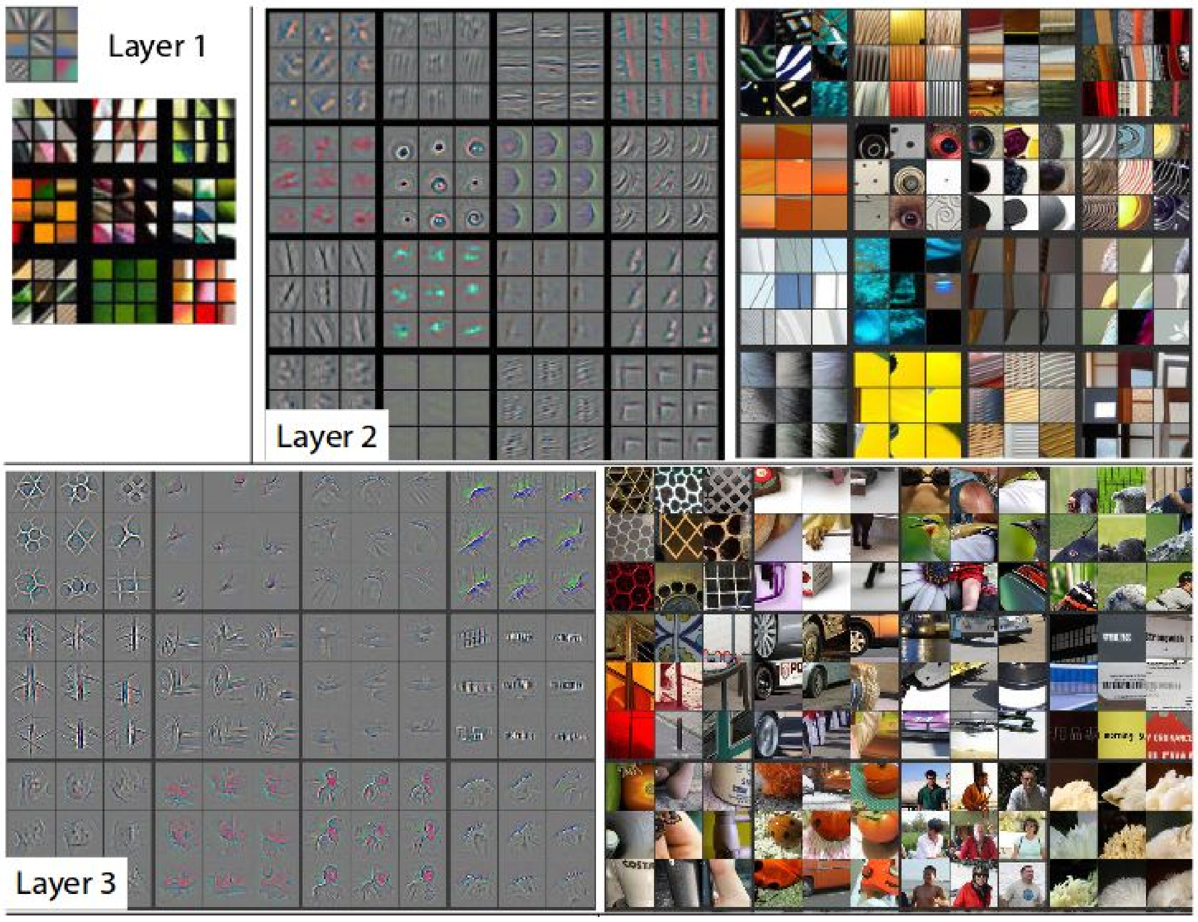
This also gives rise to the risk of the network getting over-fitted. (Will be discussed later)

Why Deep Representations?

The bottom layers of a neural network, usually, behave as local feature detectors/extractors. They detect small features, which are again used by the next layers to learn, more complex features, than the bottom layers. Again the layers towards the top of the NN, utilise these complex features to learn further complex features. This gives rise to complex hierarchical representations.

As you can see, in the picture in the next page, the filters of the Layer 1, corresponds to objects of simpler structures than the filters in Layer 2. The filters in Layer 2 have learnt more complex features than in Layer 1, as is evident in the picture.
And in Layer 3, the filters can be seen to have learnt further complex features.

In this way, the deep neural network builds up a hierarchical relationship between the learned features of the successive layers.



Parameters vs Hyper-parameters

Parameters: $W^{[1]}$, $b^{[1]}$, $W^{[2]}$, $b^{[2]}$, ...

Hyper-parameters:

1. Learning rate: $\boldsymbol{\alpha}$
2. No. of iterations
3. No. of hidden layers
4. No. of hidden units/neurons
5. Choice of activation functions
6. Dropout probability (Regularization)
7. Minibatch size (Training Minibatch GD)
8. Weight Decay (Regularization)
9. Momentum (GD with Momentum)
10. Weight Initialization

These are called Hyper-parameters because, these parameters control the real parameters **W** and **b.**

The Hyper-parameters are often manually selected. The performance of the neural network model depends very much on the value of the hyper-parameters, and needs to be set carefully to obtain optimum performance. There are several methods for finding out the optimal hyper-parameter setting.

The values of the Hyper-parameters determine the network structure and also determines how the network is trained.

---THE END---