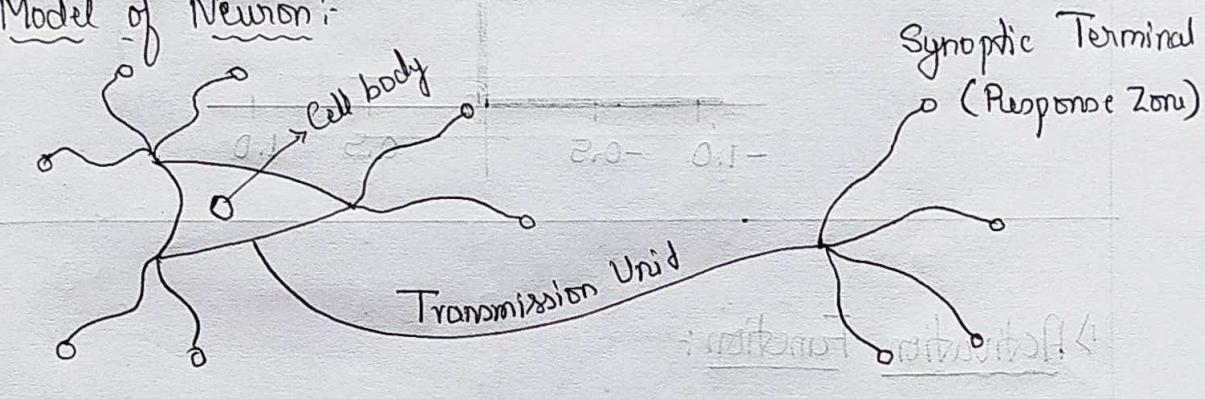
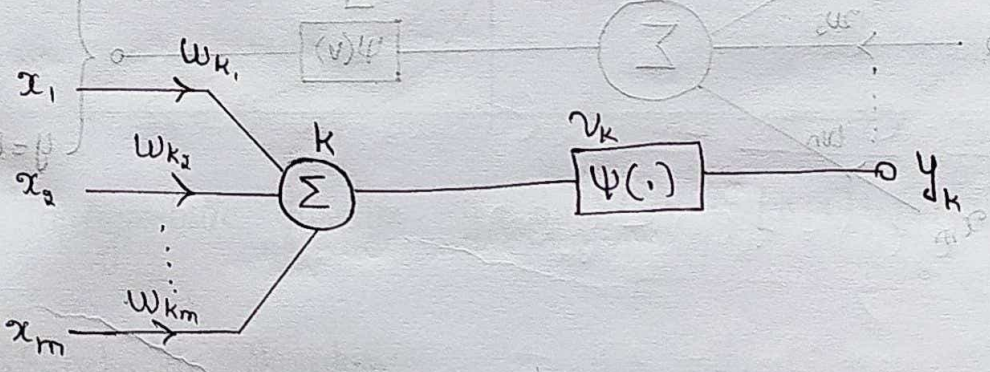
Artificial Neural Network:-

Benifits:-

* Exploits non-linearity
* Input-output Mapping
* Adaptability
* Evidential Response
* Ability for fault tallerance
* Uniformity in origin
* VLSI implementability

Model of Neuron:-





$$U_k = \sum_{j=1}^{m} x_i w_{ki}$$

$$V_k = U_k + b_2$$

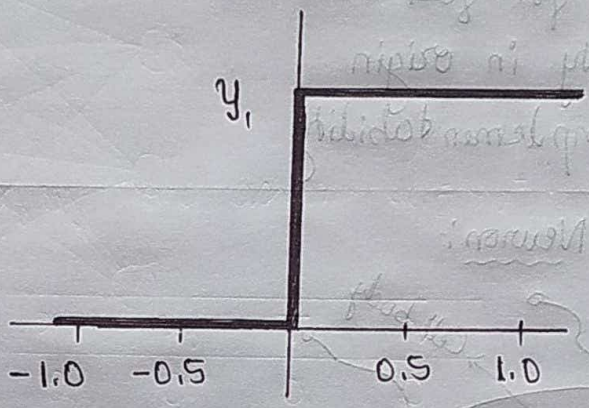
$$y_k = \psi(|A|)$$

Threshold Function (Maccallach & Pits Model)

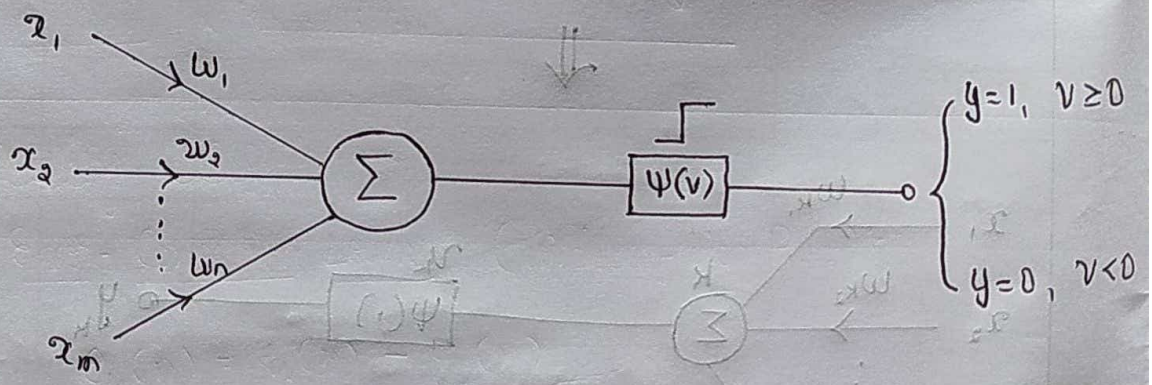
$\psi(\cdot) =$ 

if $v_k \geq 0$, $y_k = 1$

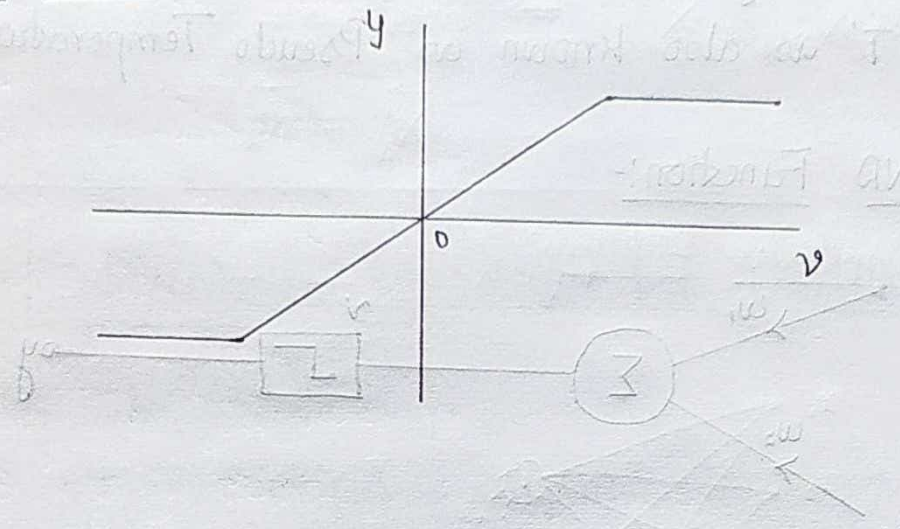
$v_k < 0$, $y_k = 0$



1) Activation Function:

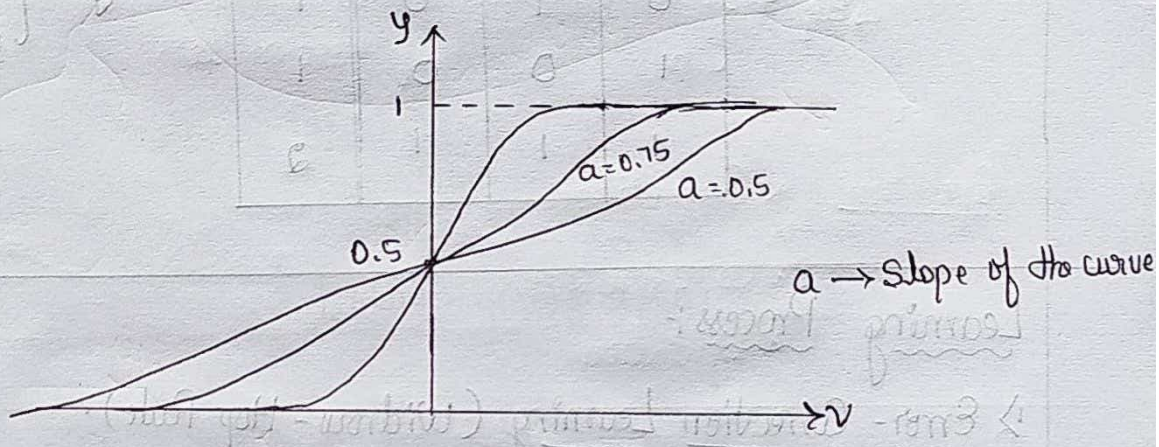


2) Linear Model:-



3) Sigmoid Function (S-Shaped):-

$$\psi(v) = \frac{1}{1 + \exp(-av)}$$



∴ Tanhyperbolic function, $y = \tanh(v) \Rightarrow [0, 1], [-1,]$

4) Stochastic function:-

* Uncertinity is introduced.

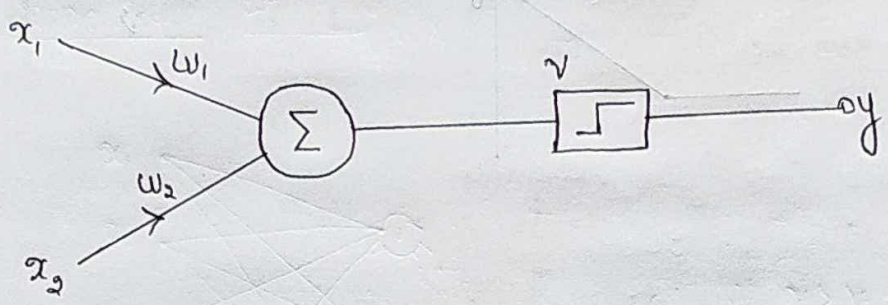
$$\therefore y = \psi(v) = \begin{cases} +1, & \text{with prob. } P(v) \\ -1, & \text{with prob. } 1 - P(v) \end{cases}$$

$$\therefore P(v) = 1 + \exp(-v/T), \quad T \rightarrow \text{Control probability}$$

if $T = 0$, $p(v) = 1$

*By increasing T, the noise is increases.
*So 'T' is also known as Pseudo Temperature

5) AND Function:-



$V = x_1 w_1 + x_2 w_2$ and assume $w_1 = w_2 = 1$

x_1	x_2	T	v
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	2

$$y = \begin{cases} 1, & v \geq 2 \\ 0, & v < 2 \end{cases}$$

Learning Process:-

1) Error-Correction Learning (Widrow-Hop Rule):-

$$e_k(n) = d_k(n) - y_k(n)$$

where, n: iteration steps
k: k^{th} neuron
t: target value
y: Actual value

Minimum Entropy Function (Cost Function)

$$E(n) = \frac{1}{2} e_k^2(n)$$

$$\therefore \Delta w(n) = \eta\, e_k(n) \cdot x_i(n)$$; $\eta \rightarrow$ Learning Rate

Delta Rule:

$$w_j(n+1) = w_j(n) + \Delta w_j(n)$$

$$\therefore E = \sum_P E^P(n) = \frac{1}{2} \sum_P e^2(n)$$

$$\therefore E^P(n) = \frac{1}{2} \sum [t(n) - y(n)]^2$$

Gradient,

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial w}$$

$$= [t(n) - y(n)]\, x_i(n)$$

$$= -e(n) \cdot x_i(n)$$

Weight updation : $\Delta w_i = -\eta \frac{\partial E_i}{\partial w_i}$

$$= \eta e(n) \cdot x_i(n)$$

↳ known as Gradient Descent.

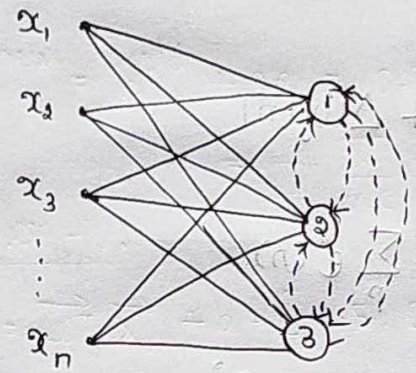
2) Memory Based Learning:-

$$\{\vec{X}_i\}_{i=1}^N , \{d_i\}_{i=1}^N$$

$d(\vec{X}_{Test}, \vec{X}_i) \Rightarrow$ Gives minimum distance.

$$\vec{X}'_N = \vec{d}_N$$

3) Competative Learning:-



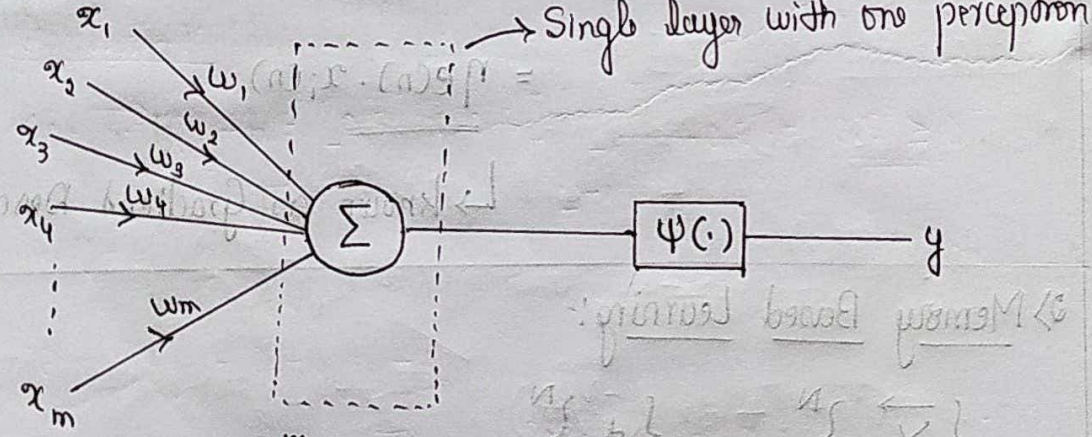
$$y = \begin{cases} 1, & \text{if } v > v_j \text{ for all } j \\ 0, & \text{otherwise} \end{cases}$$

$$\therefore \Delta w_j = \begin{cases} \eta(x_j - w_j), & \text{if neuron wins} \\ 0, & \text{if neuron loss} \end{cases}$$

Operation:- * Competition
* Co-operation
* Updation

Perceptron:-

Single Layer Perceptron:-



$$\therefore v(n) = \sum_{i=1}^{m} w_i(n) \cdot x_i(n) = \vec{W}^T \vec{X} \text{ (dot product)}$$

* Works only when linearly seperable losses.
* Also called "Perceptron Convergence Algorithm"

Training data,

$H_1 \in C_1$, $H_2 \in C_2$, $H = H_1 \cup H_2$

For correct classification,

$\vec{w}^T \vec{x} > 0$, for every $\vec{x} \in C_1$

$\vec{w}^T \vec{x} \leq 0$, for every $\vec{x} \in C_2$

For wrong classification,

if $\vec{w}^T \vec{x} > 0$ and $\vec{x} \in C_2$, so update the weight

i.e. $\vec{w}(n+1) = \vec{w}(n) - \eta \vec{x}(n)$ ⟶ decrease

if $\vec{w}^T \vec{x} \leq 0$, and $\vec{x} \in C_1$, then

$\vec{w}(n+1) = \vec{w}(n) + \eta \vec{x}(n)$ ⟶ Increases

Let, $\vec{x}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$, $\vec{x}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$, initially, $\vec{w} = \begin{bmatrix} 0.5 \\ -1 \\ -0.5 \end{bmatrix}$

$d_1 = 0$ $d_2 = 1$ $b = 0.5$

and $y = \begin{cases} 1, & v \geq 0 \\ 0, & \text{otherwise} \end{cases}$ and assume $\eta = 1$

Soln:-

Epoch-1:-

Iteration-1:-

$$v = \vec{w}^T \vec{x}_1 + b = [0.5 \quad -1 \quad -0.5] \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0.5 = 2.5 \geq 0$$

$\therefore y = 1$

But $d_1 = 0$ is mismatch with 'y' so calculate the error and update the weight

$$e = d - y = 0 - 1 = -1$$

$$w^{new} = w^{old} + \eta e \vec{x} = \begin{bmatrix} 0.5 \\ -1 \\ -0.5 \end{bmatrix} + (1)(-1)\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 0 \\ 0.5 \end{bmatrix}$$

$$b^{new} = b^{old} + \eta e = 0.5 + (1)(-1) = -0.5$$

Iteration - 2:-

$$v = \vec{w}^T \vec{x}_2 + b = [-0.5 \quad 0 \quad 0.5]\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + (-0.5) = -1.5 < 0$$

$\therefore y = 0.$

But $d_2 = 1$ is mismatch so updat weight.

$$e = d - y = 1 - 0 = 1$$

$$w^{new} = w^{old} + \eta e \vec{x} = \begin{bmatrix} -0.5 \\ 0 \\ 0.5 \end{bmatrix} + (1)(1)\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1 \\ -0.5 \end{bmatrix}$$

$$b^{new} = b^{old} + \eta e = -0.5 + (1)(1) = 0.5$$

Epoch - 2:-

Iteration - 1:-

$$v = \vec{w}^T \vec{x}_1 + b = [0.5 \quad 1 \quad -0.5]\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0.5 = 0.5 \geq 0$$

$\therefore y = 1, \quad d_1 = 0.$

$$e = d - y = 0 - 1 = -1$$

$$w^{new} = w^{old} + \eta e \vec{x} = \begin{bmatrix} 0.5 \\ 1 \\ -0.5 \end{bmatrix} + (1)(-1)\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 2 \\ 0.5 \end{bmatrix}$$

$b^{new} = 0.5 + (1)(-1) = -0.5$

Iteration-2:

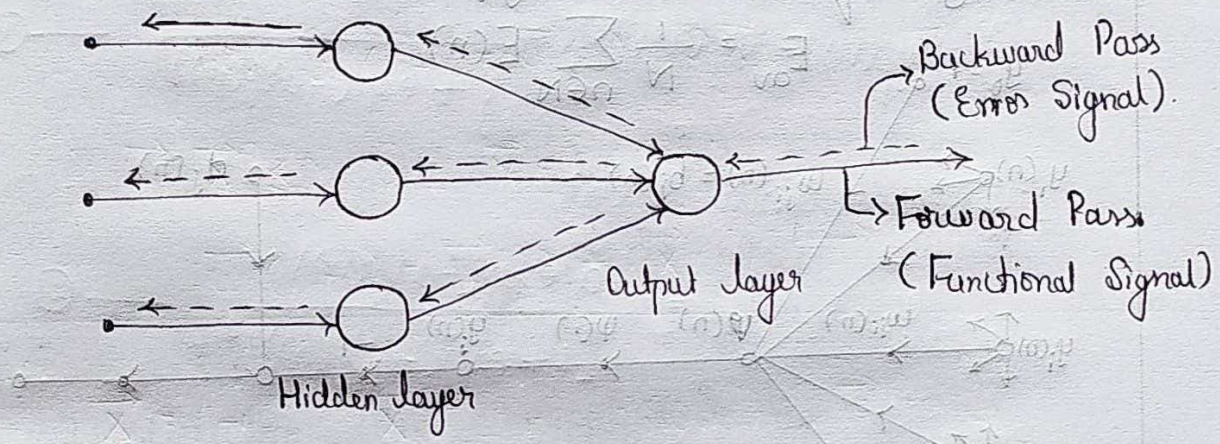
$$v = \vec{w}^T \vec{x} + b = [-0.5 \quad 2 \quad 0.5] \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + (-0.5) = 0.5 \geq 0$$

$\therefore y = 1, \; d_2 = 1$

No change.

Multilayer Perceptron:

* Multilayer feedforward networks
  → Input layer (source node)
  → One or more hidden layer (computation node)
  → Output layer (



Let,

* Indices i, j, k refers to different neuron in network.
* n is the iterations
* $\Sigma(n)$ refers to the instantaneous sum of error square at iteration n.
* $e_j(n)$ error signal at the output of neuron j at iteration 'n'
* $d_j(n)$ desired response for neuron j
* $y_j(n)$ function signal at the output neuron 'j'
* $w_j(n)$ synaptic weight connecting output of neuron i to j
* $v_j(n)$ activation potential neuron j.
* $\varphi_j(\cdot)$ activation function of neuron j

Back-Propagation Algorithm:

Error signal at the output of neuron 'j' at iteration 'n' is,

$$e_j(n) = d_j(n) - y_j(n) \quad \text{——— (1)}$$

Instantaneous value of the error energy for neuron 'j'

$$\frac{1}{2} e_j^2(n)$$

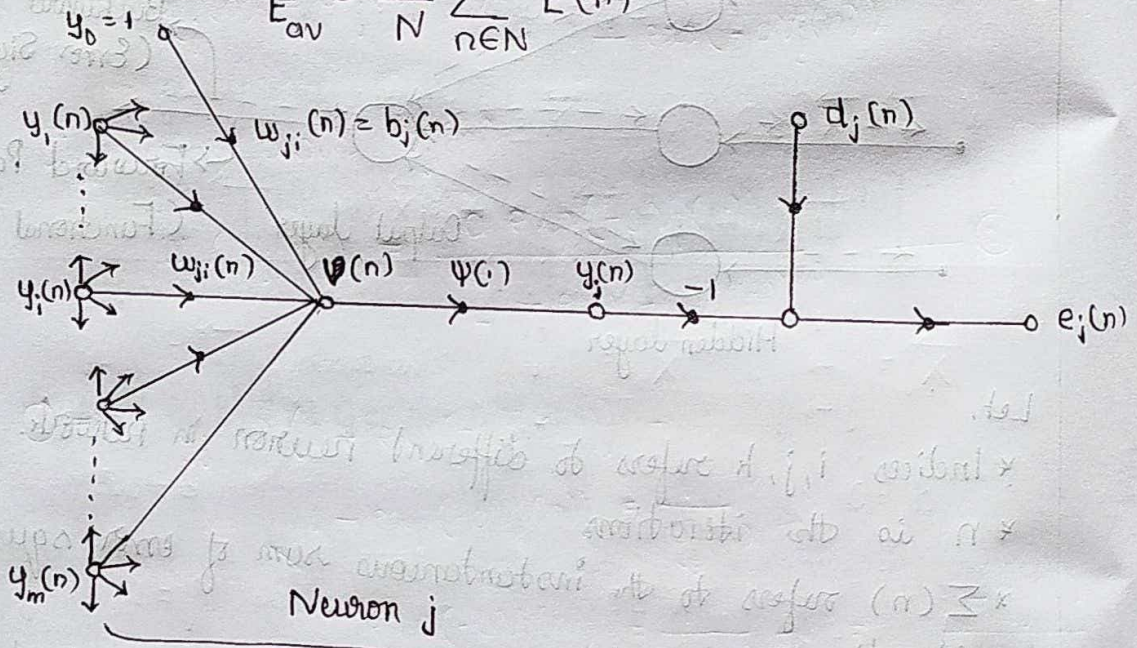
For total instantaneous error energy, sum over all output neurons

$$E(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n) \quad \text{——— (2)}$$

where 'C' is the all output neuron.

Let 'N' be the no. of examples in the training set. Then average squared error energy is,

$$E_{av} = \frac{1}{N} \sum_{n \in N} E(n)$$



$$\therefore v_j(n) = \sum_{i=0}^{m} w_{ji}(n)\, y_i(n) \quad \text{——— (3)}$$

$$y_j(n) = \varphi_j(v_j(n)) \quad \text{——— (4)}$$

Using chain rule calculus,

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \cdot \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

Differentiate eqn ① to ④ and put them in above eqn.

* Diff^n ② w.r.to $e_j(n)$

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n)$$

* Diff^n ① w.r.to $y_j(n)$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1$$

* Diff^n ④ w.r.to $v_j(n)$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \psi_j'(v_j(n))$$

Diff^n ③ w.r.to $w_{ji}(n)$

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_j(n)$$

Substitute these into the chain rule,

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = -e_j(n)\,\psi_j'(v_j(n))\,y_i(n)$$

∴ The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is,

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)}$$

$$= \eta\,\delta_j(n)\,y_i(n)$$

where $\delta_j(n) = e_j(n)\,\psi_j'(v_j(n))$ is local gradient.

* There are 2 possibilities

Case-1:- Neuron j is the output neuron → direct solution.

Case-2:- Neuron j is the hidden neuron

* When y_j is the hidden neuron we cannot get the desired response. We can redefine $\delta_j(n)$ for hidden neuron.

BP Networks:-

$$\Delta w_{ji}(n) = \eta \, \delta_j(n) \cdot y_i(n)$$

$y_i(n)$ → Input signal of neuron j

$\delta_j(n)$ → Local Gradient

- Output Neuron: $e_j(n) \, \varphi'(v_j(n))$
- Hidden Neuron: $\varphi'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$

1) Logistic Activation Function:-

$$y_j(n) = \varphi_j(v_j(n)) = \frac{1}{1 + \exp(-a v_j(n))} \quad \text{---} (1)$$

differentiate w.r.to $v_j(n)$

$$\varphi_j'(v_j(n)) = \frac{a \exp(-a v_j(n))}{[1 + \exp(-a v_j(n))]^2} \quad (2)$$

(1) $\Rightarrow$ $y_j(n)\,(1 + \exp(-a v_j(n))) = 1$

$$1 + \exp(-a v_j(n)) = \frac{1}{y_j(n)}$$

$$\exp(-a v_j(n)) = \frac{1}{y_j(n)} - 1 = \frac{1 - y_j(n)}{y_j(n)}$$

$\therefore$ (2) becomes,

$$\varphi_j'(v_j(n)) = a \, \frac{1 - y_j(n)}{y_j(n)} \cdot y_j(n)^2$$

$$= a \, y_j(n)[1 - y_j(n)] \quad \text{---} (3)$$

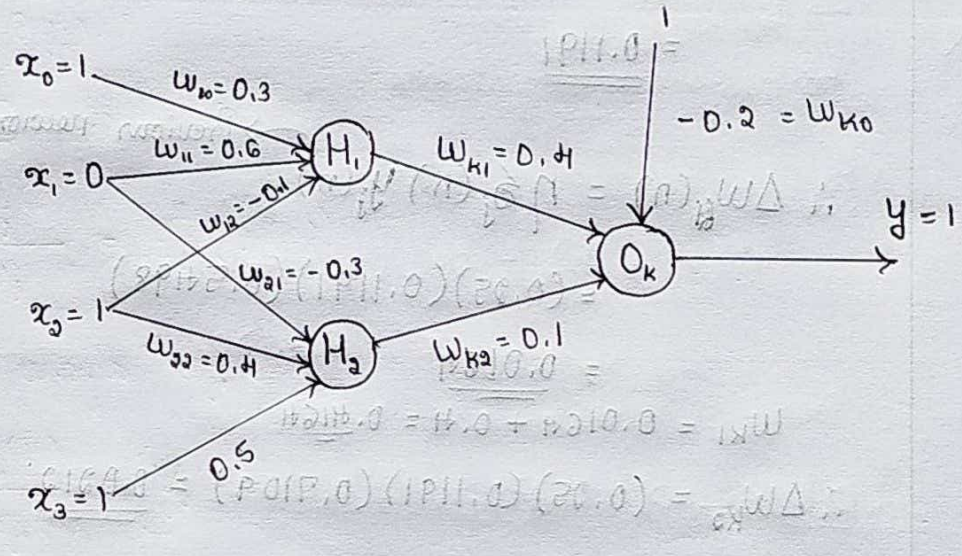
∴ $\delta_j(n)$ for Output neuron is,

$$\delta_j(n) = e_j(n) \cdot a\, y_j(n)[1 - y_j(n)]$$

∴ $\delta_j(n)$ for hidden neuron is,

$$\delta_j(n) = a\, y_j(n)[1 - y_j(n)] \sum_k \delta_k(n)\, w_{kj}(n)$$

Given,



and input pattern [0, 1], target o/p = 1, $\eta = 0.25$, $a = 1$.

Soln:-

For Hidden Layer

H_1, $v_1 = (1)(0.3) + (0)(0.6) + (1)(-0.1) = 0.2$

$$\psi_1(n) = \frac{1}{1 + e(-v)} = \frac{1}{1 + e(-0.2)} = 0.5498$$

H_2, $v_2 = (0)(-0.3) + (1)(0.4) + (1)(0.5) = 0.9$

$$\psi(n) = \frac{1}{1 + e(-0.9)} = 0.7109$$

Output k,

$$v_k = (0.5498)(0.4) + (0.7109)(0.1) - 0.2 = 0.09101$$

$$\therefore y_k = \frac{1}{1 + e(-0.09101)} = 0.5227$$

Here we need o/p, $y=1$, but we got $y_k = 0.5227$, so we need to update the weight at output neuron. So error,

$$e(n) = y - y_k = 1 - 0.5227$$

$$\therefore \delta_j(n) = e_j(n) \, a \, y_j(n)[1 - y_j(n)]$$

$$= [1 - 0.5227](1)(0.5227)[1 - 0.5227]$$

$$= 0.1191$$

$$\therefore \Delta w_{kj}(n) = \eta \delta_j(n) y_j(n)$$

$$= (0.25)(0.1191)(0.5498)$$

$$= 0.0164$$

$$w_{k1} = 0.0164 + 0.10 = 0.1164$$

$$\therefore \Delta w_{k2} = (0.25)(0.1191)(0.7109) = 0.0212$$

$$w_{k2} = 0.10 + 0.0212 = 0.1212$$

$$\therefore \Delta w_{k0} = (0.25)(0.1191)(1) = 0.0298$$

$$\therefore w_{k0} = 0.0298 + (-0.3) = -0.2702$$

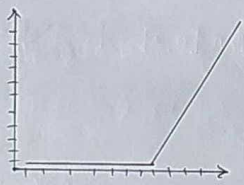
Computation Graphs:

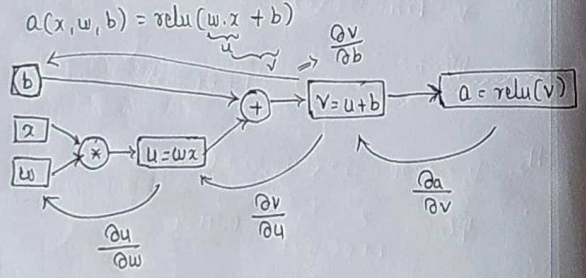
- A type of graph can be used to represent mathematical expression
- In context of deep learning (PyTorch) we can think of neural networks as computational graph
- Two types of calculation,
  - ↳ Forward computation
  - ↳ Backward computation
- Suppose we have a activation fn,

$$a(x, w, b) = \text{relu}(w \cdot x + b),$$

$$\text{relu}(z) = \begin{cases} z, & \text{if } z > 0 \\ 0, & \text{otherwise} \end{cases}$$

- ReLU → Rectified Linear Unit



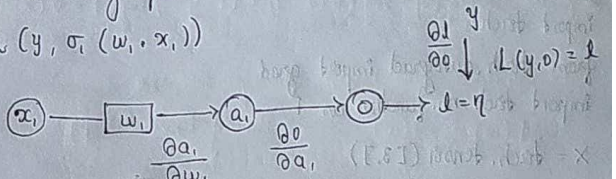


$$\therefore \frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \cdot \frac{\partial a}{\partial v}$$

$$\therefore \frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \cdot \frac{\partial a}{\partial u} = \frac{\partial u}{\partial w} \cdot \frac{\partial v}{\partial u} \cdot \frac{\partial a}{\partial v}$$

Graph with single path,

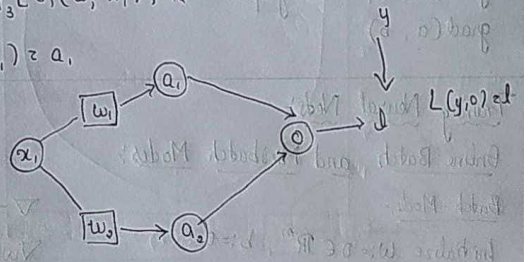
$$L(y, \sigma_1(w_1 \cdot x_1))$$



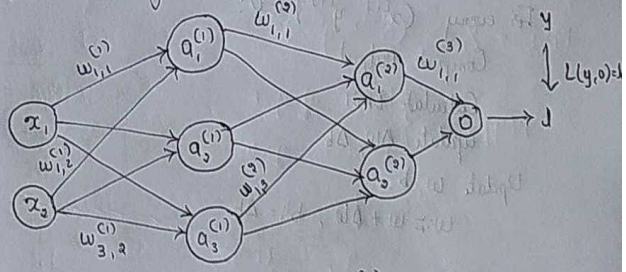
Graph with weight sharing

$$L(y, a_3[\sigma_1(w_1 x_1), \sigma_2(w_2 \cdot x_2)])$$

$$\therefore \sigma_1(x_1) = a_1$$



Graph with Fully connected layers



$$\therefore \frac{\partial J}{\partial w_{1,1}^{(1)}} = \frac{\partial J}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

$$+ \frac{\partial J}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

```
import torch
from torch.autograd import grad
import torch.nn.function as F
x = torch.tensor([3.])
w = torch.tensor([2., requires_grad = True])
b = torch.tensor([1., requires_grad = True])
a = F.relu(x * w + b)
grad(a, w, retain_graph = True)   → for saving
grad(a, b)
```

Training Neural Nets:-

Online, Batch, and Minibatch Modes:-

Batch-Mode:-

$\nabla \rightarrow$ Gradient

$\nabla_w L = \frac{\partial L}{\partial w}$

Initialize $w := 0 \in \mathbb{R}^m$, $b := 0$

For every training epoch:

  Initialize $\Delta w := 0$, $\Delta b := 0$

  For every $(x^{[i]}, y^{[i]}) \in D$:

    Compute Output

    Calculate error

    update Δw, Δb

  Update w, b:

    $w := w + \Delta w$, $b := \Delta b$

Minibatch:-

Initialize $w := 0 \in \mathbb{R}^m$, $b := 0$

For every training epoch:

  For every minibatch of size k:

    For every $\{\{x^{[i]}, y^{[i]}\}, \ldots \{x^{[i+k]}, y^{[i+k]}\}\} \subset D$

      Compute Output

      Compute error

      Update Δw, Δb

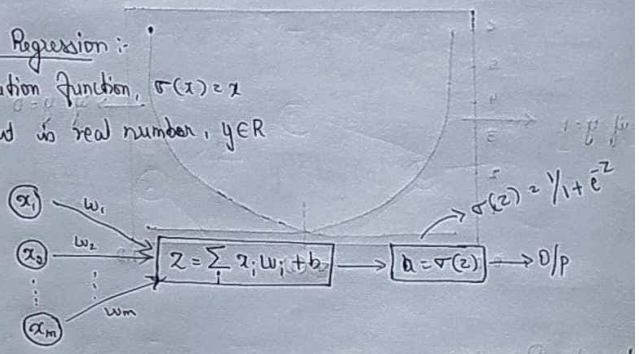
    Update w, b:

      $w := \Delta w + w$; $b := + \Delta b$.

Linear Regression:-

Activation function, $\sigma(z) = z$

Output is real number, $y \in R$



Likelihood loss:-

$$L(w) = P(y|x; w)$$

$$= \prod_{i=1}^{n} (y^{[i]} | x^{[i]}; w)$$

$$= \prod_{i=1}^{n} \left(\sigma(z^{[i]})\right)^{y^{[i]}} \left(1 - \sigma(x^{[i]})\right)^{1-y^{[i]}}$$

* In practice it is easier to maximize the log of this equation is called log-likelihood loss.

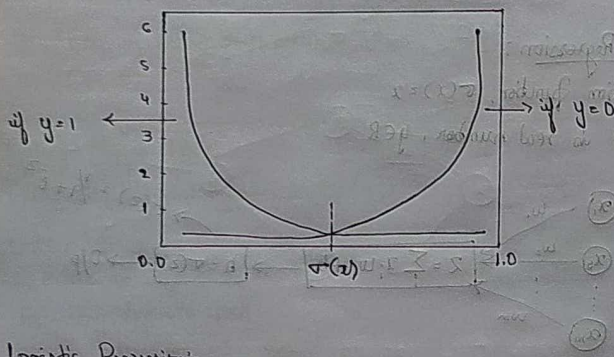
  i.e. $l(w) = \log(L(w))$

$$= \sum_{i=1}^{n} \left[y^{[i]} \log\left(\sigma(x^{[i]})\right) + (1 - y^{[i]}) \log\left(1 - \sigma(x^{[i]})\right) \right]$$

* To minimize this regard the equation, i.e

$$\therefore L(w) = -l(w) \longrightarrow \text{Binary Cross Entropy}$$

$$= -\sum_{i=1}^{n} \left[y^{[i]} \log\left(\sigma(x^{[i]})\right) + (1 - y^{[i]}) \log\left(1 - \sigma(x^{[i]})\right) \right]$$



Logistic Regression:-

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (a^{[i]} - y^{[i]})^2$$

$$P(y|x) = \begin{cases} h(x), & \text{if } y=1 \\ 1 - h(x), & y=0 \end{cases}$$

↓ Rewrite

$$P(y|x) = a^{y} (1-a)^{(1-y)}$$

Learning Rate:

$$\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial a} \frac{da}{dz} \frac{\partial z}{\partial w_j}$$

$$\therefore \frac{\partial L}{\partial a} = \frac{a-y}{a-a^2}, \quad \frac{da}{dz} = \frac{e^{-z}}{(1+e^{-z})^2} = a(1-a), \quad \frac{\partial z}{\partial w_j} = x_j$$

∴ Multi-Category Cross entropy for k-different class labels

$$H(y) = \sum_{i=1}^{n} \sum_{k=1}^{p} -y_k^{(i)} \log\left(a_i^{[i]}\right)$$

Softmax / Multi-class classification

$$P(y=t \mid z_t^{[i]}) = \sigma_{softmax}(z_t^{[i]})$$

∴ Loss function

$$L(w) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{h} -y^{[i]} \log\left(a_j^{[i]}\right)$$

Better ways to train Deep Neural Networks:-

- ↳ Learning rate
- ↳ Activation function
- ↳ Weight updation
- ↳ Normalization
- ↳ Regularization

Optimization:-

1) Batch Normalization:-

Step-1: Normalize the net input

$$\mu_j = \frac{1}{n}\sum_i z_j^{[i]}$$

$$\sigma_j^2 = \frac{1}{n}\sum_i (z_j^{[i]} - \mu_j)^2$$

$$z_j^{'[i]} = \frac{z_j^{[i]} - \mu_j}{\sigma_j} \quad \text{or} \quad z_j^{'[i]} = \frac{z_j^{[i]} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Step-2: Pre-Activation scaling

$$z_j^{'[i]} = \frac{z_j^{[i]} - \mu_j}{\sigma_j}$$

$$\sigma_j^{[i]} = \gamma_j z_j^{'[i]} + \beta_j$$

β_j → Controls the mean

γ_j → Controls the spread & scale.

2) Weight Initialization:- (Xavier Initialization)

Step-1:- Initialize weight from Gaussian or uniform distribution

Step-2:- Scale the weight proportional to the number of inputs to the layer.

Rationale → Variance of the samples linearly increases as the sample size increases.

$$w^{(l)} := w^{(l)} \cdot \sqrt{\frac{1}{m^{(l-1)}}}$$

where 'm' is the no. of input units to the current layer.

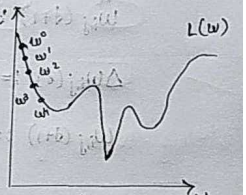
Optimizers:-

Challenges:-

Gradient Descent:

$$w^0 = w^{init}$$

$$w^{i+1} = w^i - \eta \nabla_w L(w^i)$$



- Neural network's loss (L(w)) is not convex w.r.to the network parameters w.
- There exist multiple local minima.
- Choosing the learning rate too low leads to very slow progress.
- Steep cliffs can pose great challenges to optimization.
- Gradient clipping is a method
- At a Saddle point we have $\nabla_w L(w) = 0$, but we are not at a minimum.
- A flat region is called plateau where $\nabla_w L(w) \approx 0 \Rightarrow$ Slow Progress

Training with "Momentum":-

$$\Delta w_{i,j}(t) := \alpha \Delta w_{i,j}^{(t-1)} + \eta \frac{\partial L}{\partial w_{ij}}(t)$$

∴ Weight update using the velocity vector

$$w_{ij}(t+1) := w_{ij}(t) - \Delta w_{ij}(t)$$

Nesterov Momentum:-

* Look ahead and calculate the gradient w.r.to predicted parameters

$\hat{w}_{i,j}(t+1)$

$$\hat{w}_{i,j}(t+1) := w_{i,j}(t) - \alpha \Delta w_{i,j}(t-1)$$

$$\Delta w_{i,j}(t) := \alpha \Delta w_{i,j}(t-1) + \eta \frac{\partial L}{\partial \hat{w}_{i,j}}(t+1)$$

$$w_{i,j}(t+1) := w_{i,j}(t) - \Delta w_{i,j}(t)$$

Adaptive Learning:-

1) Define a local gain(g) for each weight (Initialize with $g=1$)

$$\Delta w_{i,j}(t) := \eta \cdot g_{i,j} \cdot \frac{\partial L}{\partial w_{i,j}}$$

2) If gradient is considered

$$g_{i,j}(t) := g_{i,j}(t-1) + \beta$$

else

$$g_{i,j}(t) := g_{i,j}(t-1) \cdot (1-\beta)$$

Adaptive Learning Rate via RMSProp:-

$$\text{MeanSquare}(w_{i,j}, t) := \beta \cdot \text{MeanSquare}(w_{i,j}, t-1) + (1-\beta)\left(\frac{\partial L}{\partial w_{i,j}(t)}\right)^2$$

where $\beta = 0.9$ to 0.999

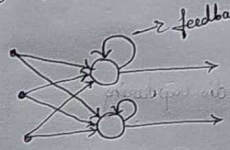
$$w_{i,j}(t) := w_{i,j}(t) - \eta \cdot \frac{\partial L}{\partial w_{i,j}} \Big/ \sqrt{\text{MeanSquare}(w_{i,j}, t) + \epsilon}$$

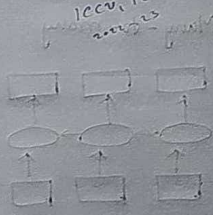
Adaptive

Common Application of CNN:-

* Image Classification * Object detection * Object Segmentation
* Image Synthesis * Human detection * Face Recognition / Identification
* Facial expression * Fatigue detection * Lip reading
* Unmanned Aerial Vehicles (Drones) * Biometrics
* Computer Aided Diagnosis

Recurrent Neural Network (RNN):-





Classical Approach:-

1) Bag-of-word model

Raw training dataset

$x^{[1]}$ = "The sun is shining"

$x^{[2]}$ = "....."

$x^{[3]}$ = "....."

$y = [0, 1, 0]$

Class label

Dictionary

Vocabulary = {

"and": 0,

"is": 1,

}

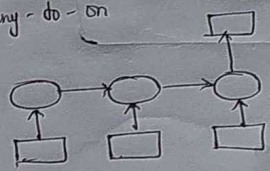
Desing matrix:

$$x = \begin{matrix} x^{[1]} \\ x^{[2]} \\ x^{[3]} \end{matrix} \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots \\ 0 & 0 & 1 & \cdots & \\ 2 & 1 & 0 & 2 & \cdots \end{bmatrix}$$

$y = [0, 1, 0]$

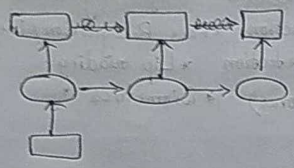
Different types of Sequence Modeling

1) Many-to-one



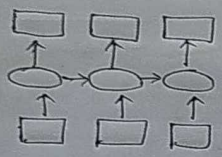
Eg:- Sentiment Analysis

2) One-to-many

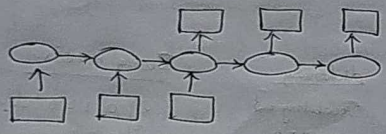


Eg:- Text extracting from image

3) Many-to-many

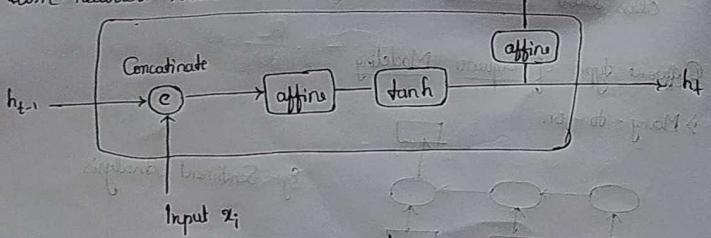


Eg:- Vedio capturing



delayed

Basic Recurrent Neural Network



if $a_h > 1$ gradient will ~~exp~~ explode

if $a_h < 1$ gradient will vanish

Backpropagation through time:-

$$L = \sum_{t=1}^{T} L^{(t)} = \frac{\partial L^{(t)}}{\partial w_{h,h}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial h^{(t)}} \cdot \left(\sum_{k=1}^{t} \frac{\partial h^{(t)}}{\partial h^{(k)}} \cdot \frac{\partial h^{(k)}}{\partial w_{hh}} \right)$$

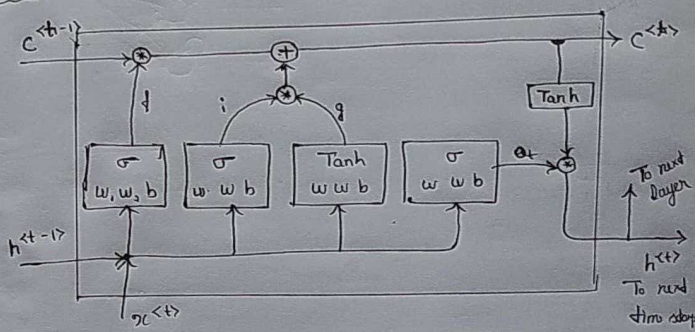
Solutions for Vanish / Exploding problem

↳ Gradient Clipping

↳ Truncated back propagation through time (TBPTT)

↳ Long short-term memory (LSTM)

LSTM:-



Forget gate, $f_t = \sigma(w_{fx} x^{(t)} + w_{fh} h^{(t-1)} + b_f)$

Input gate, $i_t = \sigma(w_{ix} x^{(t)} + w_{ih} h^{(t-1)} + b_i)$

Input node, $g_t = \tanh(w_{gx} x^{(t)} + w_{gh} h^{(t-1)} + b_g)$

$C^{(t)} = (C^{(t-1)} \odot f_t)$

Output gate, $o_t = \sigma(w_{ox} x^{(t)} + w_{oh} h^{(t-1)} + b_o)$

$\therefore h^{(t)} = o_t \odot \tanh(C^{(t)})$