Hebbian Learning Rule :- It is one of the first and easiest learning rules in ANN. It is a single layer network, i.e. has one input layer and one output layer. One input layer can have many ~~input~~ units say h. The output layer ~~can have many~~ has one unit. Hebbian rule works by updating the weights between neurons in the neural network for each training sample. It is unsupervised learning rule.

Hebbian Learning Algorithm :-

1. Set all weights to zero; $w_i = 0$ for $i = 1$ to n and bias to zero.
2. For each input vector, S(input vector): t(target output pair), repeat steps 3-5.
3. Set activations for input units with the input vector $X_i = S_i$ for $i = 1$ to n.
4. Set the corresponding output value to the output neuron, i.e. $y = t$.
5. Update the weight and bias by applying Hebb rule for all $i = 1$ to n:

$$w_i(new) = w_i(old) + x_i y$$

$$b(new) = b(old) + y$$

Implementig AND Gate:-

	x_1	x_2	b	Target Y
x_1	-1	-1	1	-1
x_2	-1	1	1	-1
x_3	1	-1	1	-1
x_4	1	1	1	1

there are 4 training samples, so there will be 4 iterations.
Also activation function used here is Bipolar sigmoidal function so the range is $[-1, 1]$;

Step 1 :- Set the weight and bias to zero, $w = [0\ 0\ 0]^T$, and $b = 0$

Step 2: Set input vector $x_i = s_i$ for $i = 1$ to 4

$x_1 = [-1\ -1\ 1]^T$

$x_2 = [-1\ 1\ 1]^T$

$x_3 = [1\ -1\ 1]^T$

$x_4 = [1\ 1\ 1]^T$

Step 3:= output value is set to $y = t$.

Step 4: modifying weights using Hebbian Rule.

$w(new) = w(old) + x_i y_i = [0\ 0\ 0]^T + [-1\ -1\ 1]^T[-1]$

$= [1\ 1\ -1]^T$

Second Iteration :-

$w(new) = [1\ 1\ -1]^T + [-1\ 1\ 1]^T[-1]$

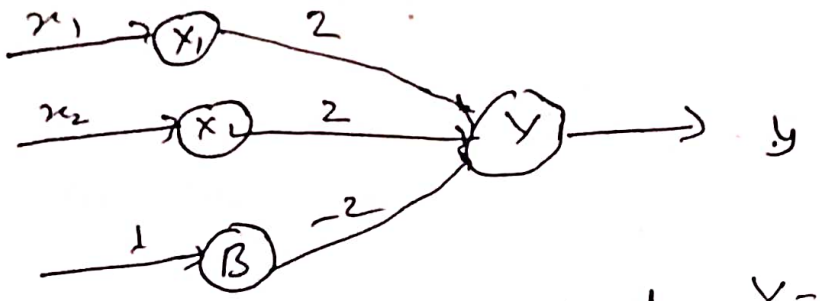
$= [2\ 0\ -2]^T$

Third Iteration $w(new) = [2\ 0\ -2]^T + [1\ -1\ 1]^T \cdot [-1]$

$= [1\ 1\ -3]^T$

Fourth iteration:- $w(new) = [1\ 1\ -3]^T + [1\ 1\ 1]^T (1)$

$= [2\ 2\ -2]^T$



for ① $x_1 = -1$, $x_2 = -2$, $b = 1$, $y = (-1)(2) + (-1)(2) + 1\cdot(-2) = -6$

for $(x_1) = -1$, $x_2 = 1$, $b = 1$, $y = (-1)2 + 1\cdot 2 + 1(-2)$

$= -2$

for $(x_1) = 1$, $x_2 = -1$, $b = 1$, $y = 1\cdot 2 + (-1)2 + 1(-2)$

$= -2$

for ~~(4)~~ $x_1 = 1$, $x_2 = 1$, $b = 1$, $y = 1\cdot 2 + 1\cdot 2 + 1(-2)$

$= 2$

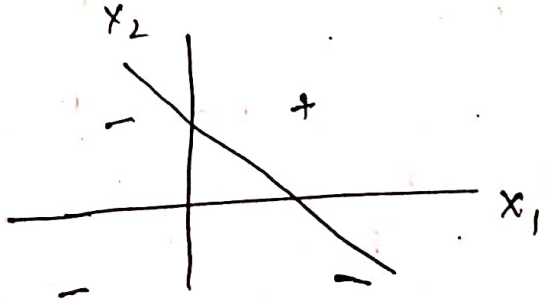
Decision boundary:

$$2x_1 + 2x_2 - 2b = y$$

Replacing y with 0, $2x_1 + 2x_2 - 2b = 0$

Since bias; $b = 1$, so $2x_1 + 2x_2 - 2\cdot(1) = 0$

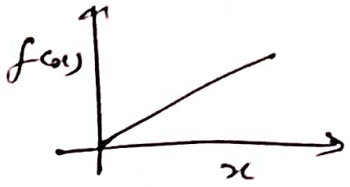
$x_2 = 1 - x_1$



Activation functions :- The activation function is used to calculate the output response of a neuron. The sum of signal is applied to with an activation to obtain the response. For neurons in same layer, same activation functions are used. There may be linear as well as nonlinear activation functions. The non-linear activation functions are used in a multilayer net.

(I) Identity function

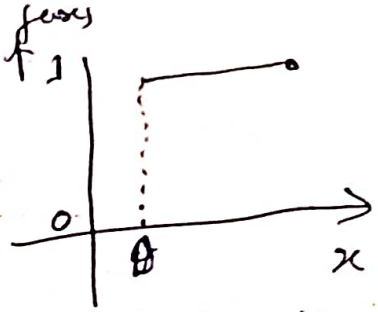
$$f(x) = x \text{ ; for all } x.$$



(II) Binary steps function

$$f(x) = \begin{cases} 1 & \text{; if } f(x) \geq \theta \\ 0 & \text{; if } f(x) < \theta \end{cases}$$

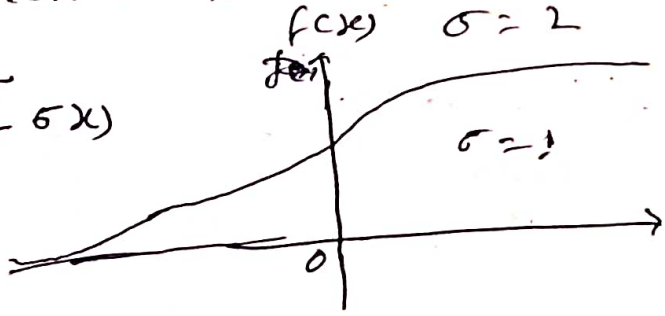
where θ is some threshold value.



BAI10329

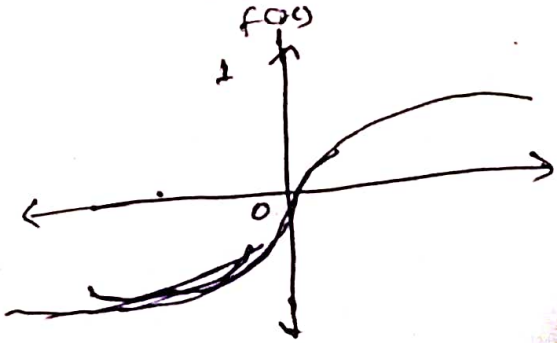
(III)(A) Binary sigmoidal function :- This is also called logistic function. It ranges between 0 to 1.

$$f(x) = \frac{1}{1 + \exp(-\sigma x)}$$



(B) Bipolar Sigmoidal function :- The desired range here is between +1 and -1. This function is hyperbolic tangent function.

$$b(x) = 2f(x) - 1 = \frac{1 - \exp(-\sigma x)}{1 + \exp(-\sigma x)}$$



Perceptron Learning Rule :-

For the perceptron learning rule, the learning signal is the difference between the desired and actual neuron's response. This types of learning is supervised.

Training Algorithm

Step 1: Initialize weights and bias (initially it can be zero). Set learning rate α (0 to 1).

Step 2: While stopping condition is false do steps 3-7.

Step 3: for each training pair (s, t) do step 4-6.

Step 4: Set activations of input units:

$$x_i = s_i \quad \text{for } i = 1 \text{ to } n.$$

Step 5: Compute the output unit response

$$y_{in} = b + \sum_i x_i w_i$$

the activation function used is

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \le y_{in} \le \theta \\ -1 & \text{if } y_{in} \le -\theta \end{cases}$$

Step 6: The weights and bias are updated if the target is not equal to the output response.

if $t \neq y$ and the value of x_i is not zero

$$w_{i(new)} = w_{i(old)} + \alpha t x_i$$

$$b(new) = b(old) + \alpha t$$

else $w_{i(new)} = w_{i(old)}$

$$b(new) = b(old)$$

Step 7: Test for stopping condition.

For and AND Gate function :-

x_1	x_2	b	t
-1	-1	1	-1
-1	1	1	-1
1	-1	1	-1
1	1	1	1

Step 1 : Initialize weights $w_1 = w_2 = 0$ and $b = 0, \alpha = 1, \theta = 0$

Step 2: for input pair (-1,-1) do step 3-5

Step 3: Set activation of inputs

$$x_i = (-1, -1)$$

Step 4: $y_{in} = b + \sum x_i w_i = 0 + -1 \times 0 + -1 \times 0 = 0$

$$y = f(y_{in}) = 0$$

Step 5: Since $t \neq y$

$$w_{i(new)} = w_i(old) + \alpha t x_i$$

$$= 0 + 1 \times (-1) [-1 \ -1]$$

$$= [1 \ 1]$$

$$b(new) = b(old) + \alpha t = 0 + 1 \cdot 1 = 1$$

The new weight and bias are [1 1 1]

$x_2 = (-1, 1)$, $y_{in} = 1 + -1 \times 1 + 1 \times 1 = 1$, $y = f(y_{in}) = 1$

so $y_{in} = 1$, since $t = y_{in}$

$$w_{new} = [1, 1] + 1 \cdot (-1) [-1 \ 1]$$

$$= [1 \ 1] + [1 \ -1] = [2 \ 0]$$

$$b_{new} = 1 + 1 \cdot (-1) = 0$$

$x_3 = (1, -1)$, $y_{in} = 0 + [1 \ -1] \begin{bmatrix} 2 \\ 0 \end{bmatrix}$

$$= 0 + 1 \times 2 + -1 \times 0 = 2$$

$y = f(y_{in}) = f(2) = 1$, $b(new) = 0 + 1 \cdot (-1) = -1$

~~$x_4 = (1, 1)$, $y_{in} = -1$ t~~

Since ~~$y = 1 \neq t$~~,

~~$w_{new} = [2 \quad 0]$~~

$y \neq t$, since $y = 1$, $t = -1$,

$$w_{new} = [2 \quad 0] + 1 \cdot (-1)[1 \quad 1]$$

$$= [1 \quad 1]$$

~~$b_{new} = 1 + 1 = (-1)$~~

for $x_4 = (1, 1)$, $y_{in} = -1 + 1 \times 1 + 1 \times 1 = 1$

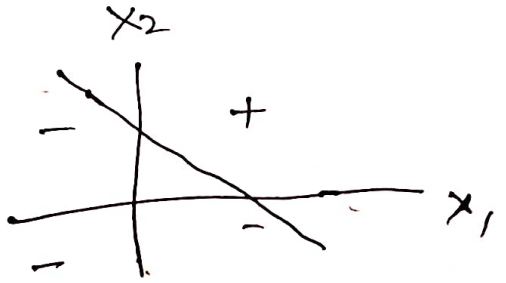
$$y = f(y_{in}) = f(1) = 1$$

Now $y = 1 = t$, so updated weight $= [1 \quad 1]$

and bias $= -1$

$$-1 + x_1 + x_2 = 0$$

$$x_2 = 1 - x_1$$



Associative Memory Network :-

Associative neural nets are single layer nets in which the weights are determined to store an assed of pattern associations.

Hebb Rule for Pattern Association :-

Step 1: Initialize all weight ($i=1,\dots,n$, $j=1,2,\dots,m$)

$$w_{ij}=0$$

Step 2: For each training input-target output vector, $s:t$, do

Steps 3-5

Step 3: Set activation for input units to present training input ($i=1,2,\dots,n$)

$$x_i = s_i$$

Step 4: Set activations for output units to current target output ($j=1,\dots,m$)

$$y_j = t_j$$

Step 5: Adjust the weights

$$w_{ij}(new) = w_{ij}(old) + x_i y_j$$

Delta Rule for Pattern Association :-

$$\Delta w_i = \alpha (t - y_{in}) x_i$$

Delta Rule for Several output units :-

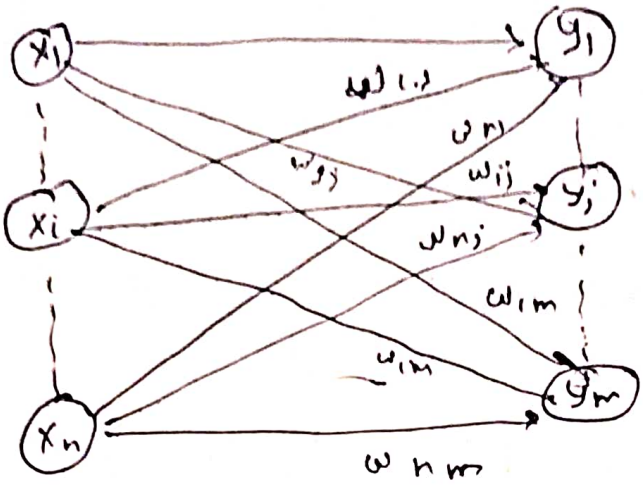
$$\Delta w_{ij} = \alpha (t_j - y_{inj}) x_i$$

Extended Delta Rule

$$\Delta w_{ij} = \alpha (t_j - y_j) x_i f'(y_{in})$$

Hetero Associative Memory Neural Network :-

Associative memory neural networks are networks in which the weights are determined in such a way that the net can store a set of p pattern associations. Hetero associative networks are static networks. Nonlinear or delay operation can be done using hetero associative networks. The weights may be found using the Hebb rule or delta rule.



Hetero Associative Neural Net.

Application Algorithm :

Step 1: Weights are initialized using Hebb or delta rule.

Step 2: For each input vector do steps 3 to 5

Step 3: Set the activations for input layer units equal to the current vector x_i

Step 4: Compute the net input to the output units

$$y_{inj} = \sum_{i=1}^{n} x_i w_{ij}$$

Step 5: Determine the activation of the output units

$$y_j = \begin{cases} 1, & \text{if } y_{inj} > 0 \\ 0, & \text{if } y_{inj} = 0 \\ -1, & \text{if } y_{inj} < 0 \end{cases}$$

if the target

(b) $y_{in} = \sum x_i v_{ij}$

$$= [1 \quad 1 \quad -1 \quad -1] \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & +1 \\ -1 & -1 & 1 & 1 \end{bmatrix}$$

$$y_{in} = [4 \quad 4 \quad -4 \quad -4]$$

$$y = f(y_{in}) = f[4 \quad 4 \quad -4 \quad -4]$$

$$= [1 \quad 1 \quad -1 \quad -1]$$

The response vector is same as the input vector, we can say that the input vector vector is recognized as a know vector.

Bidirectional Associative memory :

In this network net iterates by sending a signal back and forth between the two layers until each neurons remains constant for several steps. Three form of BAM are

(I) Binary (II) Bipolar (III) Continuous

Activation function for binary associative memory is,

$$y_j = \begin{cases} 1, & \text{if } y_{inj} > 0 \\ y_j, & \text{if } y_{inj} = 0 \\ 0 & \text{if } y_{inj} < 0 \end{cases}$$

$$x_i = \begin{cases} 1; & \text{if } x_{in_i} > 0 \\ x_i & \text{if } x_{in_i} = 0 \\ 0 & \text{if } x_{in_i} < 0 \end{cases}$$

For bipolar vector, the activation function for Y-layer is

$$y_j = \begin{cases} 1, & \text{if } y_{inj} > V_j \\ y_j & \text{if } y_{inj} = V_j \\ 0 & \text{if } y_{inj} < V_j \end{cases}$$

$$x_i = \begin{cases} 1 & \text{if } x_{in\,i} > V_i \\ o_j & \text{if } x_{ini} = V_i \\ 0 & \text{if } x_{ini} < V_i \end{cases}$$

For Continuous BAM.

Y-layer $f(y_{inj}) = \dfrac{1}{1+\exp(-y_{inj})}$

if bias is calculate $y_{ini} = b_j + \sum x_i w_{ij}$

X-layer $f(x_{inj}) = \dfrac{1}{1+\exp(-x_{inj})}$

if bias is calculated

$$x_{ini} = b_i + \sum y_j w_{ij}$$

At the memory capacity of BAM is $\min(n, m)$ where,

n is the number of units in x-layers

m is the number of units in y-layers.

Training Algorithm for Madaline :

Step 1: Initialize weights and bias, set learning rate α.

$V_1 = V_2 = 0.5$ and $b_2 = 0.5$, other weights may be small random values).

Step 2: When stopping condition is false do steps 3-9.

Step 3: For each bipolar training pair $s:t$, do steps 4-9.

Step 4: Set activations of input units:

$x_i = s_i$ for $i = 1$ to n

Step 5: Calculate the net input of hidden adaline units

$$z_{in1} = b_1 + x_1 w_{11} + x_2 w_{21}$$

$$z_{in2} = b_2 + x_1 w_{12} + x_2 w_{22}$$

Step 6: Find the output of hidden adaline units using activation function for z_1, z_2 and y is given by

$$z_1 = f(z_{in1})$$

$$z_2 = f(z_{in2})$$

where $f(p) = \begin{cases} 1, & \text{if } p \geq 0 \\ 0, & \text{if } p < 0 \end{cases}$

Step 7: Calculate net input to output

$$y_{in} = b_3 + z_1 v_1 + z_2 v_2$$

Apply activation to get the output of net.

$$y = f(y_{in})$$

Step 8: Find the error and do weight updation

if $t = y$, no weight updation

if $t \neq y$, then,

if $t = 1$, then update weight on z_j unit whose net input is closest to 0.

$w_{ij}(new) = w_{ij}(old) + \alpha (1 - z_{in_j}) x_i$

$b_j(new) = b_j(old) + \alpha (1 - z_{in_j})$

if $t = -1$, then update weights on all unit z_k which have positive net input

$w_{ik}(new) = w_{ik}(old) + \alpha (-1 - z_{in_k}) x_i$

$b_k(new) = b_j(old) + \alpha (1 - z_{in_k})$

Step 9: Test for the stopping condition.

Delta Learning Rule (Widrow-Hoff Rule or Least mean square Rule)

" The mean square error for a particular training pattern is

$$E = \sum_j (t_j - y_{in_j})^2$$

The gradient of E is a vector consisting of the partial derivation of E with respect to each of the weight.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_j (t_j - y_{in_j})^2$$

$$= -2 (t_j - y_{in_j}) \frac{\partial y_{in_j}}{\partial w_{ij}}$$

$$= -2 (t_j - y_{in_j}) x_i$$

Hence error will be rapidly reduced depending upon the given learning by adjusting the weight according to the delta rule.

$$\Delta w_{ij} = \alpha (t_j - y_{in_j}) x_i$$

Training Algorithm :-

Step 1 : Initialize weights (not zero but small random values are used). Set learning rate α.

Step 2 : While stopping condition is false do step 3-7.

Step 3 : For each bipolar training pair (s, t) perform 4-6.

Step 4 : Set activation of input unit $x_i = s_i$ for $i = 1$ to n.

Step 5 : Compute $y_{in} = b + \sum x_i w_i$

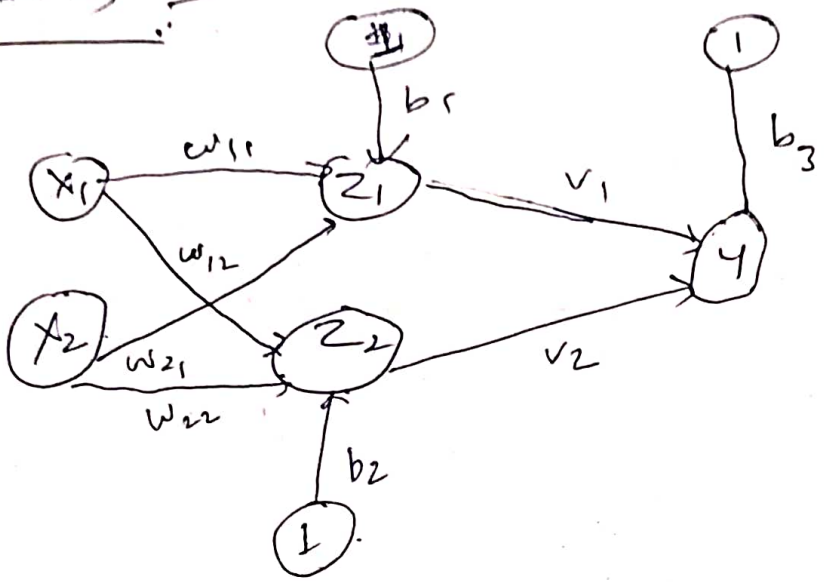
Step 6 : Update bias and weights, $i = 1$ to n

$$w_i(new) = w_i(old) + \alpha (t - y_{in}) x_i$$

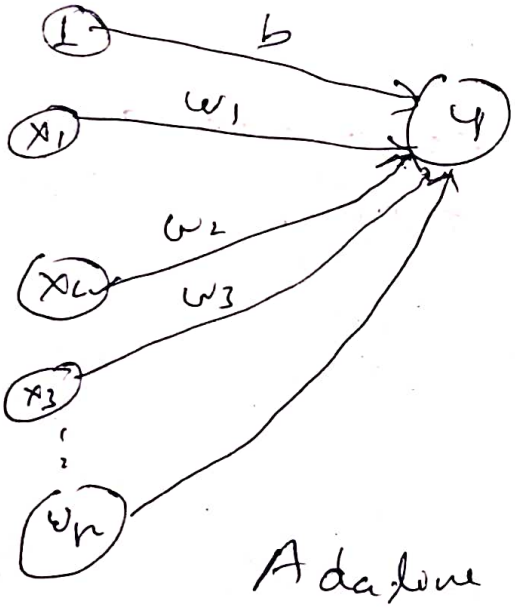
$$b(new) = b(old) + \alpha (t - y_{in})$$

Step 7 : Test for stopping condition.

Adaptive Linear Neuron (Adaline) and Multilayered Adaline (Madaline) :-



Madaline



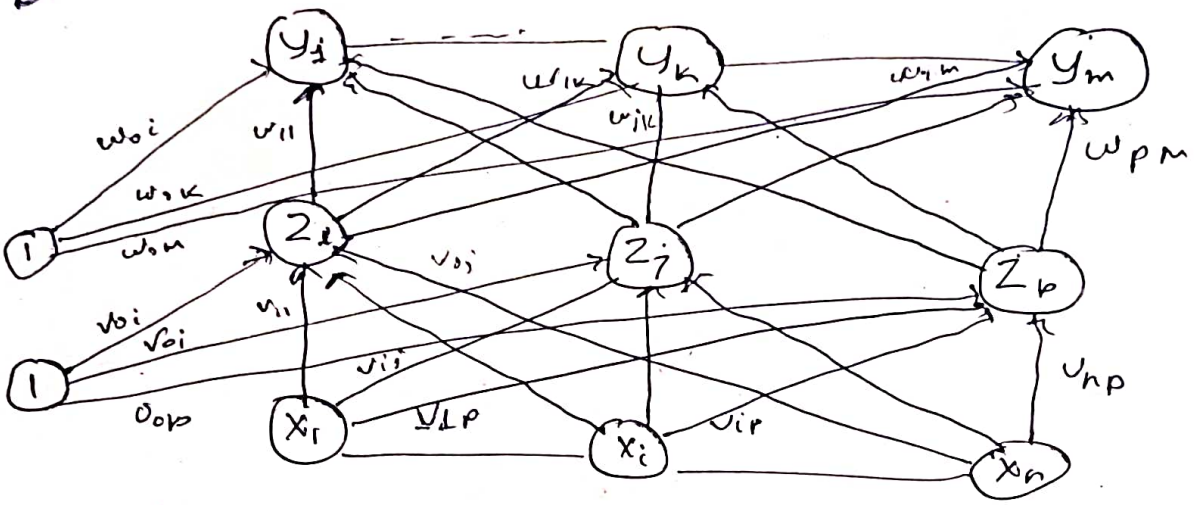
Adaline

Adaline and madaline network uses the Least mean Square error (LMS error) rule. It uses delta rule to update the weight

Back Propagation Network (BPN)

It is a multilayer forward network using extend gradient-descent based delta-learning rule, commonly known as back propagation (of errors) rule. The network is trained by supervised learning method.

D.



Derivation

Consider an arbitrary activation function f_{cs}. ~~The derivative of activation function is~~

Let

$$y_{in_k} = \sum_i z_i w_{jk}$$

$$z_{in_j} = \sum_i v_{ij} x_i$$

$$y_k = f(y_{in_k})$$

The error to be minimized is $E = 0.5 \sum_k [t_k - y_k]^2$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \left[0.5 \sum_k [t_k - y_k]^2 \right]$$

$$= -[t_k - y_k] \frac{\partial}{\partial w_{jk}} f(y_{in_k})$$

$$= -(t_k - y_k) f'(y_{in_k}) z_j$$

Let us define $\delta_k = -[t_k - y_k] f'(y_{ink})$

Weights on connections to the hidden unit z_j

$$\frac{\partial E}{\partial V_{ij}} = -\sum_k (t_k - y_k) \frac{\partial}{\partial V_{ij}} y_k$$

$$= -\sum (t_k - y_k) f(y_{ink}) \frac{\partial}{\partial V_{ij}} y_{ink}$$

$$= -\sum \delta_k \frac{\partial}{\partial V_{ij}} y_{ink}$$

Now

$$= -\sum_k \delta_k \frac{\partial}{\partial V_{ij}} \left(\sum z_j \cdot w_{jk} \right)$$

$$= -\cancel{\sum_k \delta_k \frac{\partial}{\partial V_{ij}}} = -\sum_k \delta_k w_{jk} \frac{\partial}{\partial V_{ij}} z_j$$

$$= -\sum_k \delta_k w_{jk} \frac{\partial}{\partial V_{ij}} f(z_{inj})$$

$$= -\sum_k \delta_k w_{jk} f'(z_{inj}) (x_i)$$

$$\delta_j = -\sum_k \delta_k w_{jk} f'(z_{inj})$$

The weight updation for output unit is given by

$$\Delta w_{jk} = -\alpha \frac{\partial E}{\partial w_{jk}}$$

$$= -\alpha \delta_k z_j$$

The weight updation for hidden unit is given by

$$\Delta V_{ij} = -\alpha \frac{\partial E}{\partial V_{ij}}$$

$$= -\alpha \delta_j x_i$$

This is generalized delta rule used for training purpose.

Back Propagation Algorithm:

(I) Initialize weights to small random values

(II) while stopping condition is false do step 3 - 10

(III) for each training pair do steps - 4 - 9

(IV) Each input units receives the input signal x_i and transmits this signal to all hidden units.

(V) Each hidden units (z_j, $j = 1, \dots, p$) sums its weighted input signals.

$$z_{in_j} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

Applying activation function

$$z_j = f(z_{in_j})$$

and sends this signal to all units in the layer above i.e. output units.

(VI) Each output unit (y_k, $k = 1, \dots, m$) sums its weighted input signals

$$y_{in_k} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and applies its activation function to calculate the output signals.

$$y_k = f(y_{in_k})$$

(VII) Each output units (y_k, $k = 1, \dots m$) receives a target pattern corresponding to an input pattern. error information term is calculated as

$$\delta_k = (t_k - y_k) f(y_{in_k})$$

Step 8 : Each hidden unit (z_j, $j = 1, \dots, n$) sums its delta inputs from units in the layer above

$$\delta_{inj} = \sum_{k=1}^{m} \delta_j w_{jk}$$

The error information term is calculated as

$$\delta_j = \delta_{inj} f(z_{inj})$$

Step 9 : Each output unit ($y_k, k = 1, \dots, m$) updates its bias and weight ($j = 0, \dots p$)

The weight correction term is given by

$$\Delta w_{jk} = \alpha \delta_k z_j$$

and bias correction term is given by

$$\Delta w_{0k} = \alpha \delta_k$$

Therefore $w_{jk}(new) = w_{jk}(old) + \Delta w_{jk}$

$$w_{ok}(new) = w_{ok}(old) + \Delta w_{ok}$$

Each hidden units (z_j, $j = 1, \dots, p$) updates its bias and weights ($i = 0, \dots n$)

The weight correction term

$$\Delta v_{ij} = \alpha \delta_j x_i$$

The bias correction term

$$\Delta v_{oj} = \alpha \delta_j$$

Therefore, $v_{ij}(new) = v_{ij}(old) + \Delta v_{ij}$,

$$v_{oj}(new) = \sout{v_{oj}(new)} \cdot v_{oj}(old) + \Delta v_{oj}$$

Step 10 : Test the stopping condition.

Auto Associative Memory Network :-

In auto associative net the training input and target output vectors should be identical or same the auto associative net training is often called storing the vectors, which may be binary or bipolar in nature.

Training Algorithm :-

(I) Initialize all weights, $i = 1, \dots, n$, $j = 1, \dots, n$

$$w_{ij} = 0;$$

(II) For each vector to be stored follow steps 3-4.

(III) Set activation for each input unit $i = 1, \dots, n$

$$x_i = s_i$$

(IV) set activation for each output unit $j = 1, 2, \dots, n$

$$y_j = s_j$$

(V) Adjust the weight for $i = 1$ to n and $j = 1$ to n

$$w_{ij}(new) = w_{ij}(old) + x_i y_j$$

Example :- Use the Hebb rule to store the vector $(1\ 1\ -1\ -1)$ in an auto associative neural net

(a) Find weight matrix matrix

(b) Test the input vector $x = (1\ 1\ -1\ -1)$

Sol :- (a) $$W = \begin{pmatrix} 1 \\ 1 \\ -1 \\ -1 \end{pmatrix} \begin{bmatrix} 1 & 1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix}$$

Associative memory :- Pattern association is the process of forming association between related patterns. Associative memory net can be seen as a simplified model of a human brain which can associate similar patterns. Associative neural nets are ~~sigh~~ single layer nets in which the weights are ~~updated~~ determined to store an assets of pattern associations.

Example :- A hetro associative net is trained by Hebb outer product rule for input row vectors $s = (s_1, s_2, s_3, s_4)$ to output vectors $t = (t_1, t_2)$. Find the weight matrix

$t_3 = (1\ 0)$ ~~to~~ $t_1 = (1\ 0)$ $s_3 = (0\ 0\ 1\ 1)$ $s_1 = (1\ 1\ 0\ 0)$

$t_4 = (1\ 0)$ ~~Matrix~~ $t_4 = t_2 = (0\ 1)$ $s_4 = (0\ 1\ 0\ 0)$ $s_2 = (1\ 1\ 1\ 0)$

Sol. 1. Initialize weight to 0

2. $W_1 = s_1^T t = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} [1\ 0] = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$

$W_2 = s_2^T t_2 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} [0\ 1] = \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$

$W_3 = s_3^T t_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} [1\ 0] = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$

$W_4 = s_4^T t_4 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} [1\ 0] = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$

$W = W_1 + W_2 + W_3 + W_4 = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}$

Logistic Regression : Logistic regression is an example of supervised learning. It is used to calculate or predict the probability of a binary (yes/no) event occuring. In linear regression, the outcome is continuous and can be any possible value. However in the case of logistic regression, the predicted outcome is discrete and restricted to a limited number of values.

Hence, Linear regression is an example of a regression model and logistic regression is an example of a classification model.

Logistic regressions are three types

(I) Binary logistic regression (II) multinomial logistic regression.

(III) Ordinal logistic regression.

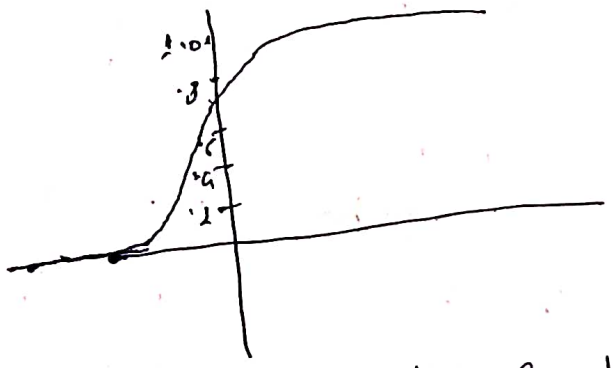
The logistic function is a simple S-shaped curve used to convert data into a value between 0 and 1.

Model , output = 0, 1

Hypothesis $\Rightarrow Z = WX + B$

$h_\theta(x) = \text{sigmoid}(Z)$

$\text{sigmoid}(t) = \frac{1}{1 + e^{-t}}$



Sigmoid Activation function

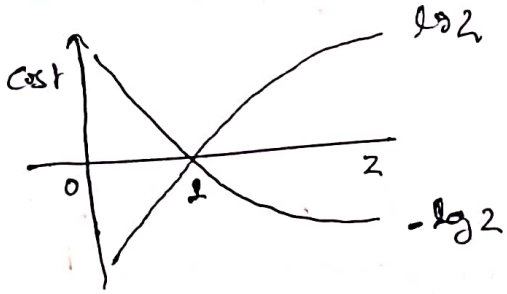
Decision Boundary:- To predict which class a data belongs, a threshold can be set. Based upon this threshold, the obtained estimated probability is classified into classes. Decision boundary can be linear or non-linear.

Say, if predicted_value ≥ 0.5, then classify email as spam else as not spam.

Cost function :

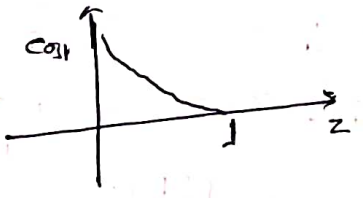
$$\text{Cost}(h_\theta(x), Y(\text{actual})) = \begin{cases} -\log(h_\theta(x)) & \text{if } y=1 \\ -\log(1-h_\theta(x)) & \text{if } y=0 \end{cases}$$

Linear regression uses mean squared error as its cost function. If this is used for logistic regression, then it will be a non-convex function of parameter θ. Gradient descent will converge into global minimum only if the function is convex.



$$, z = h_\theta(x)$$

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y=1 \\ -\log(1-h_\theta(x)) & \text{if } y=0 \end{cases}$$



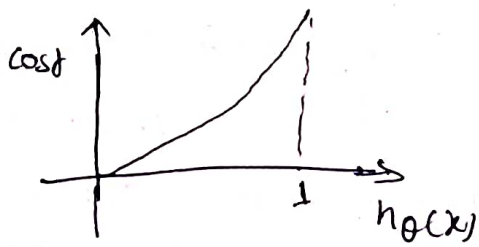
if $y=1$

$$\text{cost}(h_\theta(x), y) = -\log(h_\theta(x))$$

if cost $=0 \Rightarrow y=1, \Rightarrow h_\theta(x)=1$

cost $=\infty$ for $h_\theta(x)=0$

if $h_\theta(x)=0$, it is similar to predicting $P(y=1|x;\theta)=0$



if cost $=0, \Rightarrow h_\theta(x)=0 \Rightarrow y=0$

cost $=\infty \Rightarrow h_\theta(x)=1$

if $h_\theta(x)=1$, it is similar to predicting $P(y=0|x;\theta)=0$

Why this cost function:-

Let us consider

$$\hat{y} = P(y=1|x)$$

$\hat{y}$ is the probability that $y=1$, given x

$$1-\hat{y} = P(y=0|x)$$

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

~~$\Rightarrow \log P(y|x) = \log(\hat{y}^{y} + ($~~

$$\Rightarrow \log P(y|x) = \log(\hat{y}^{y} \cdot (1-\hat{y})^{1-y})$$

$$\Rightarrow y \log \hat{y} + (1-y)\log(1-\hat{y})$$

$$\Rightarrow -L(\hat{y}, y)$$

$$\Rightarrow \log P(y|x) = -L(\hat{y}, y)$$

This negative function is because when we train, we need to maximize the probability by minimizing loss function. Decreasing the cost will increase the maximum likelyhood assuming that samples are drawn from an identically independent distribution.

Now

$$\boxed{z = w_1 x_1 + w_2 x_2 + b} \rightarrow \boxed{\hat{y} = a = \sigma(z)} \rightarrow \boxed{L(\hat{y}, y)}$$

$$\boxed{a = \hat{y}}$$

$$w_1 \Rightarrow \frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

$$\frac{\partial L}{\partial a} = \frac{\partial}{\partial a}(-y \log - (1-y)\log(1-a))$$

$$\boxed{\frac{\partial L}{\partial a} = \frac{-y}{a} - (-1)\frac{(1-y)}{(1-a)}}$$

$$\frac{\partial a}{\partial z} = a(1-a)$$

$$\frac{\partial z}{\partial w_1} = x_1$$

$$\frac{\partial L}{\partial w_1} = \left(\left[\frac{-y}{a} + \frac{(1-y)}{(1-a)} \right] (a)(1-a) \right) \cdot x_1$$

$$= (a-y) x_1$$

update for w_1,

$$\frac{\partial L}{\partial w_1} = (a-y) \cdot x_1$$

$$\Rightarrow \boxed{w_1 = w_1 - \alpha \frac{\partial L}{\partial w_1}}$$

Similarly, for all parameters

$$\boxed{w_i = w_1 - \alpha \frac{\partial L}{\partial w_i}} \qquad i = 1, 2, \ldots m$$

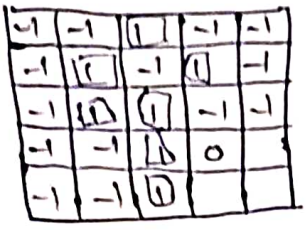
m = no. of parameters

$$\boxed{b = b - \alpha \frac{\partial L}{\partial b}}$$

where, $\frac{\partial L}{\partial b} = (a-y)$

Convolutional Neural Network :-

① Used in image recognition



Disadvantages of ANN using for Image Classification

(I) Too much computation

(II) Treats local pixels same as pixels far apart

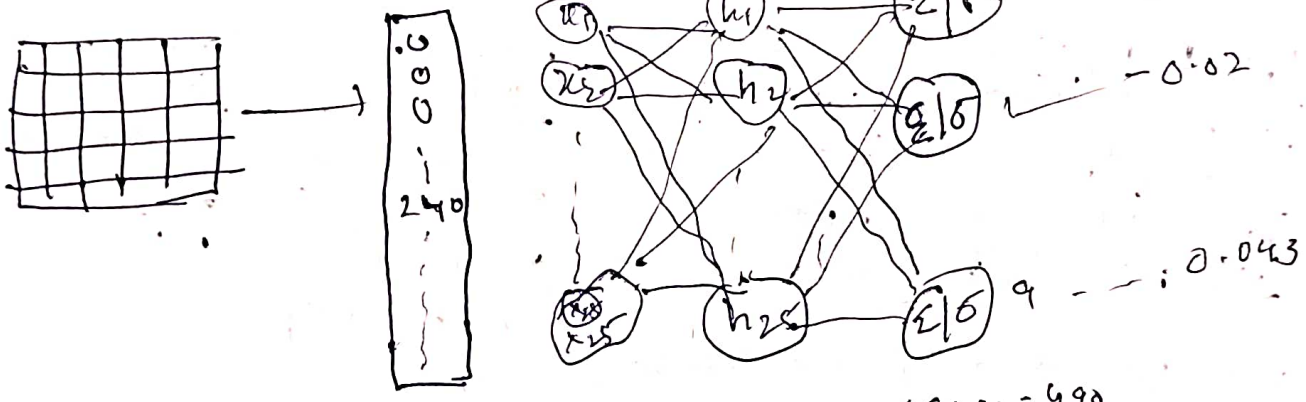
(III) Sensitive to location of an object in an image.

Suppose Computer want to recognize on a 5 × 5 grid

(a) CNN can recognize vericty but ANN cannot.

eg: 9 9 9

matrix is flatten to on dimensional vector



(b) CNN is invarient of position.

Suppose image size = 1920 × 1080 × 3 (Cow image)

First layer neurons = 1920 × 1080 × 3 = 6 million

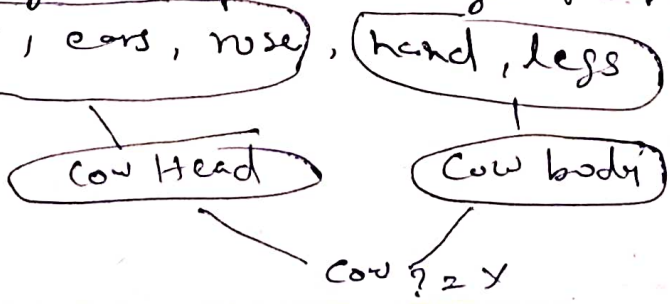
Hidden layer neuron = Let say = 4 million.

weights between Input layer and Hidden layer = 6 × 4 = 24 million.

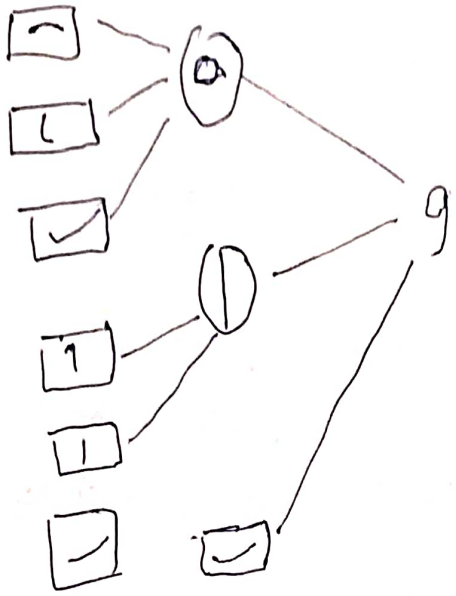
CNN is basically used for Feature extraction.

For recognition of cow image features are.

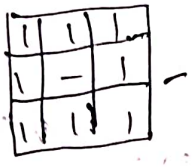
(a) eyes, ears, nose, hand, legs



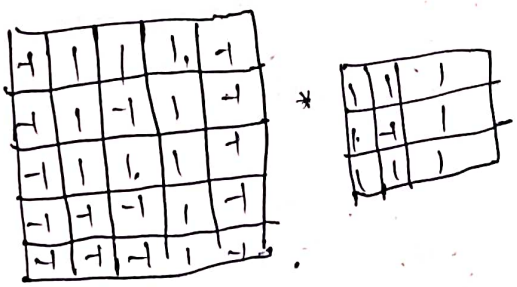
9



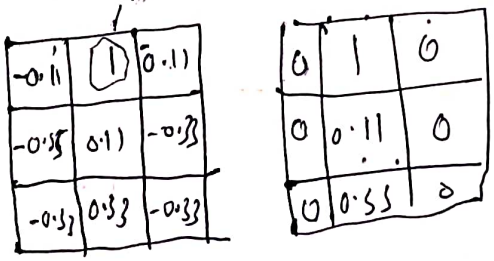
9) we apply filter for loopy image



- filter



feature set detected for loopy.

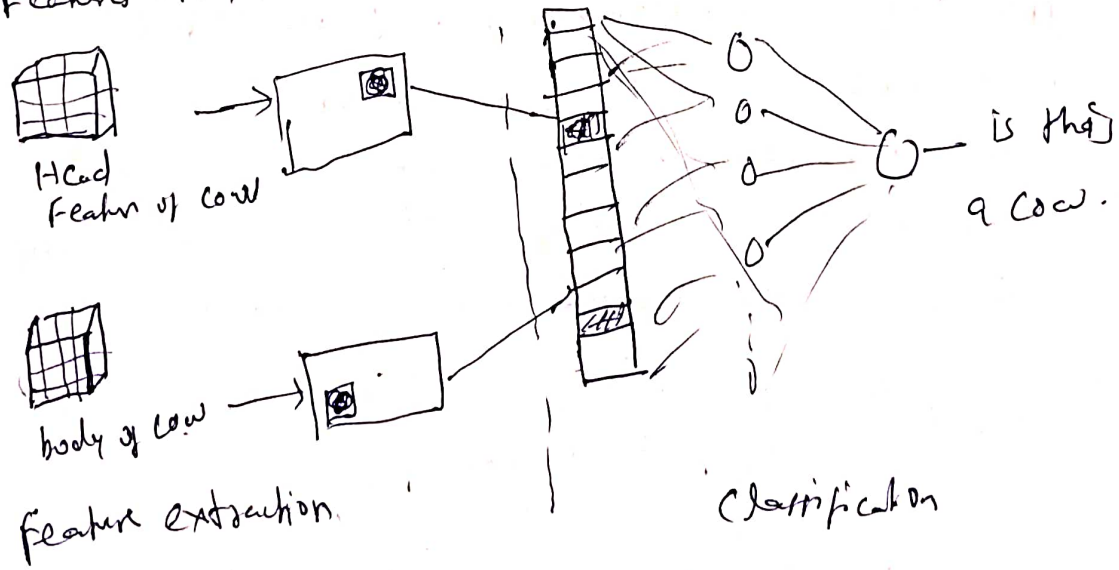


feature Map.

(i) filters are the feature detectors

(ii) filters are Location invariant.

Features are stacked and creates a 3D volume.

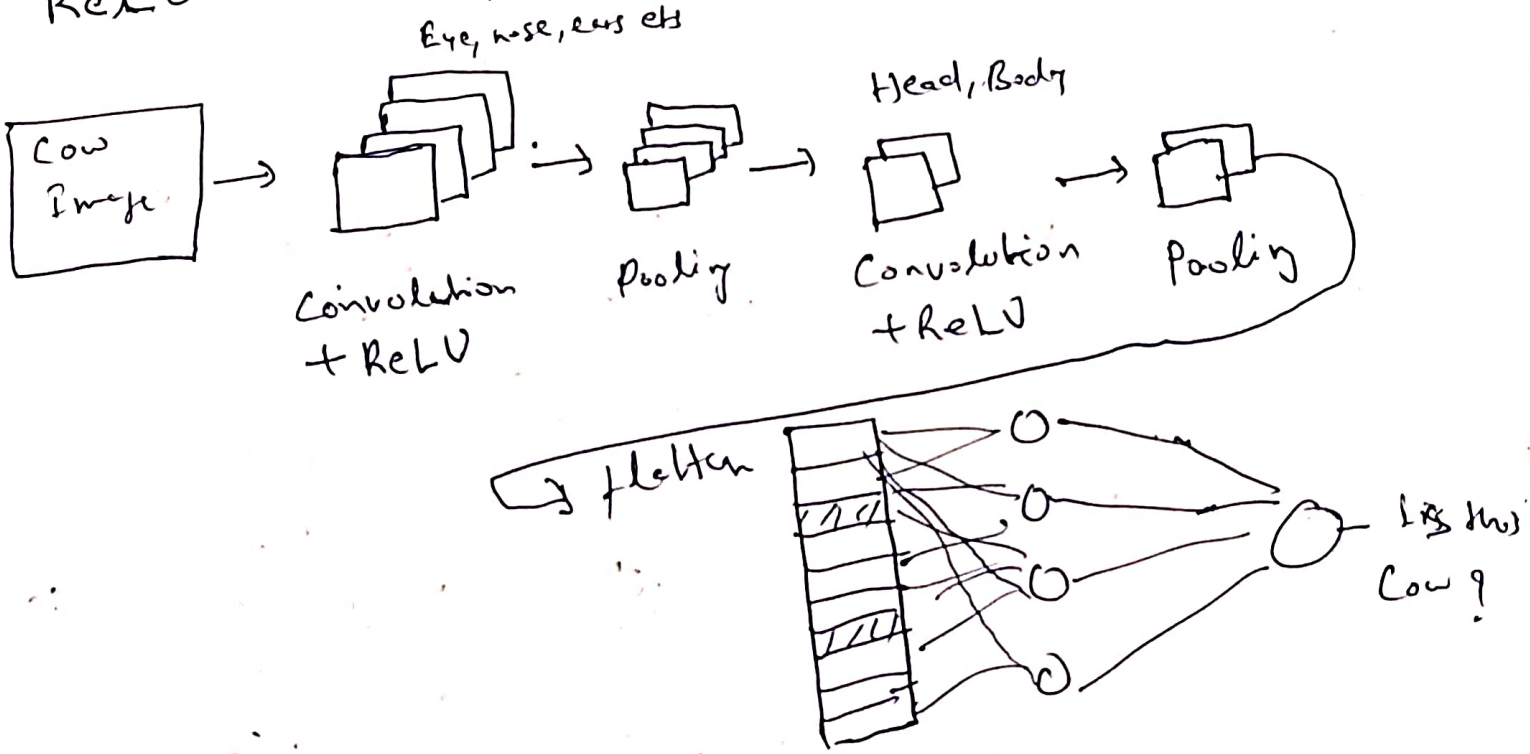


Dimension are minimized using Pooling. After convolution we perform pooling to reduce the dimension.

(I) Max Pooling
(II) Average Pooling.

Benifits of Poolings are
(I) Reduces dimensions & Computations
(II) Reduces overfitting as there are less parameters
(III) Model is tolerant towards variations, distortions.

ReLU → used for non linear activation functions.



Benifits of Convolution
- Connections Sparsity reduces overfitting
- Convolution + Pooling gives location invariant feature detection
- Parameter sharing

Benifits of ReLU
- Introduces non linearity
- Speeds up training, faster to compute

Benefits of Pooling

(I) Reduces dimension, overfitting

(II) Makes the model tolerant towards small distortion and variations.

CNN can not take care of rotation and scale.

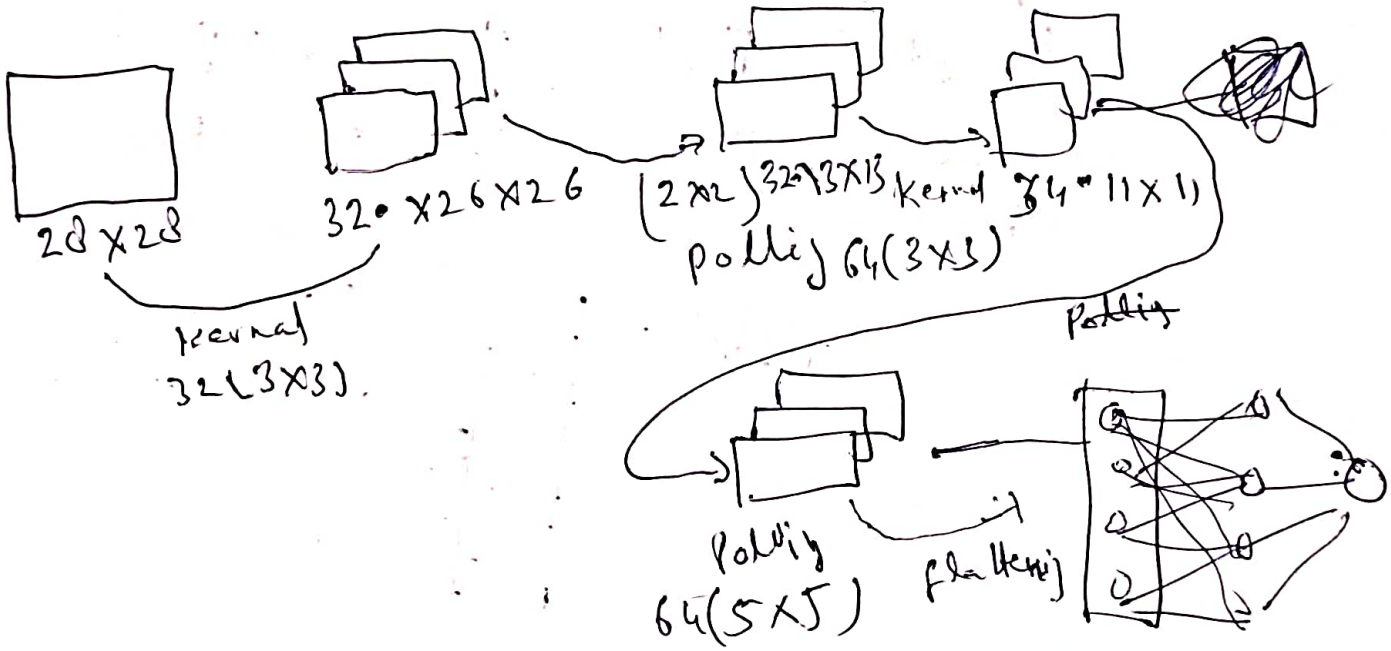
CNN has following parts

(I) Convolution + ReLU

(II) Padding

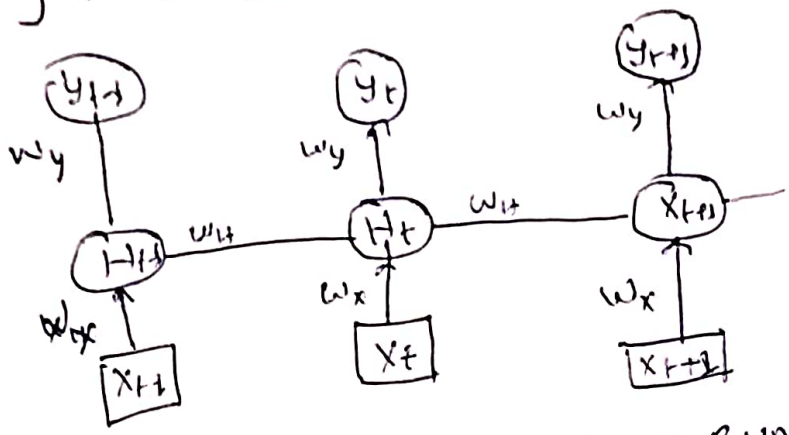
(III) Pooling.

(IV) Stride



LSTM

RNN → used for time series analysis which has some memory to connect previous state to next state.

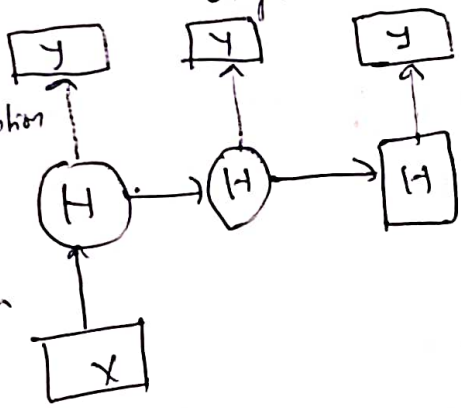


Unfold Representation of RNN

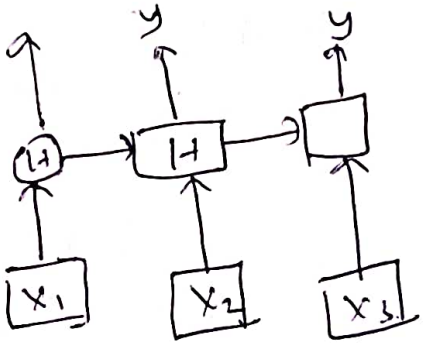
Fold RNN

Single RNN
One to one

use
(i) Text description ~~generation~~ of Image
(ii) Voice generation

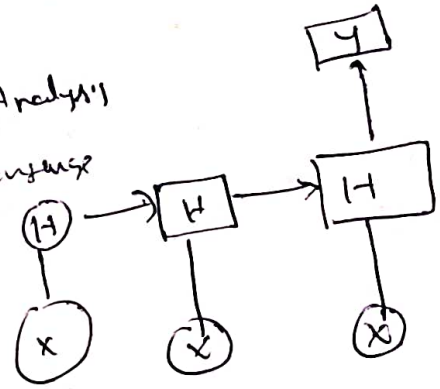


one to many



Many to many

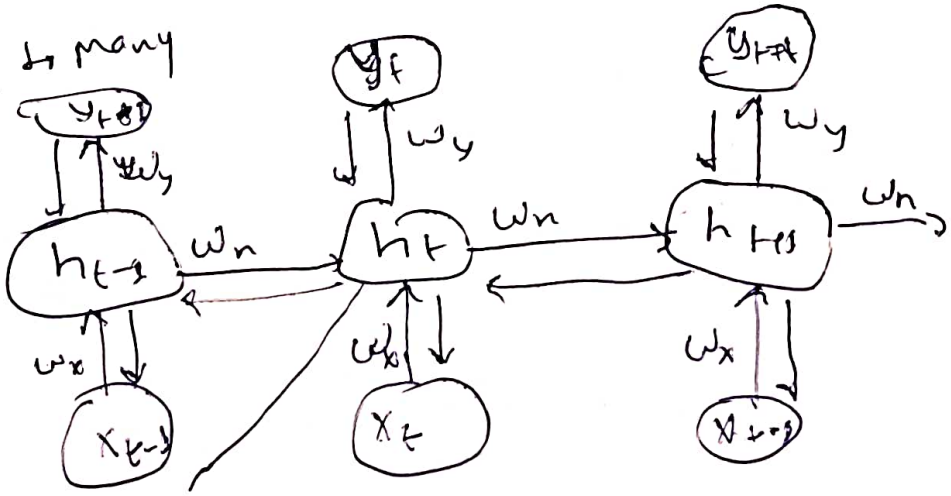
use
(i) Sequential Analysis
(ii) Sentence Language Recognition



Many to many

use
(i) Language Translation
(ii) Encoder and Decoder

Many to many



$$Z_n^{(t)} = [X_t W_x + h_{t-1} W_n] + b_n$$

$$h_t = a(z_h^{(t)})$$

$$z_y^{(t)} = (h_t * w_y) + b_y$$

$$y_t = a(z_y^{(t)})$$

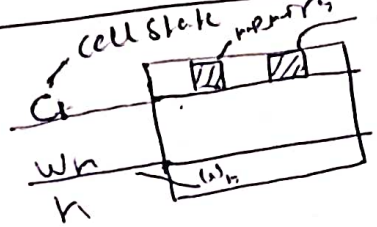
$$L = \sum_i L_i$$

$$\Delta w = \eta \frac{\partial L}{\partial w}$$

$$w = w + \Delta w$$

RNN Suffers vanishing gradient problem. To shoot out the problem of vanishing gradient we use LSTM method.

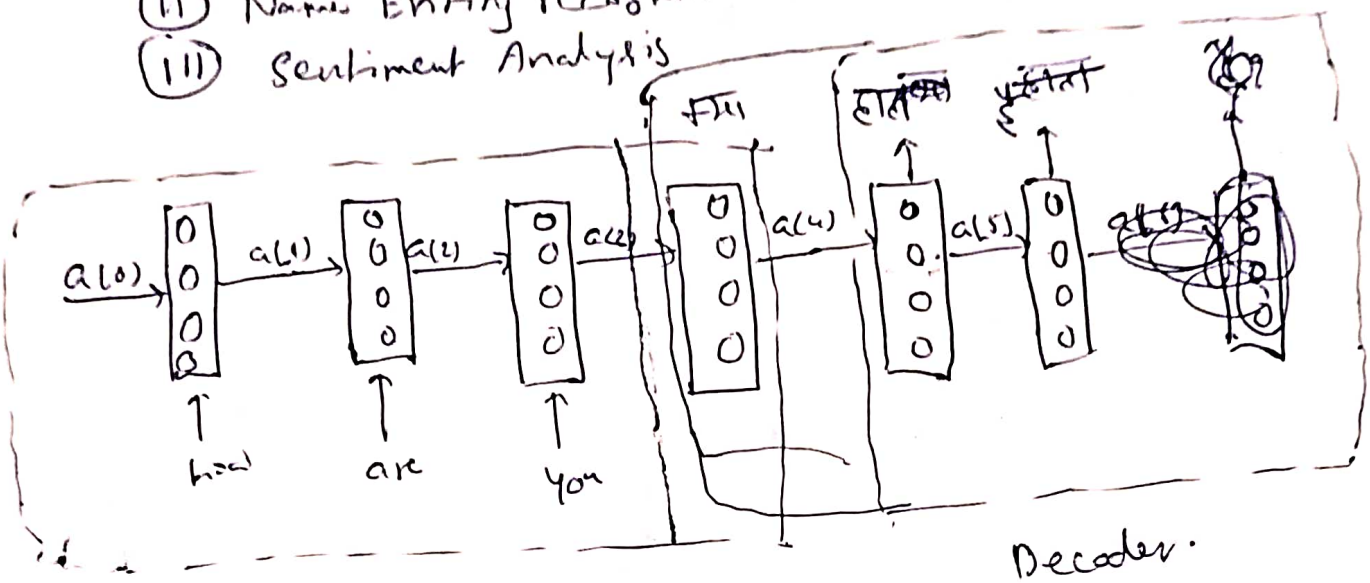
LSTM (Long short term memory) :-



one memory is to remove unnecessary memory, information another memory is to add important information or memory.

Recurrent Neural Network :

Uses (I) Translation from one language to other (Number of neuron become a problem)
(II) Name Entity recognition
(III) Sentiment Analysis



Langage Translation.

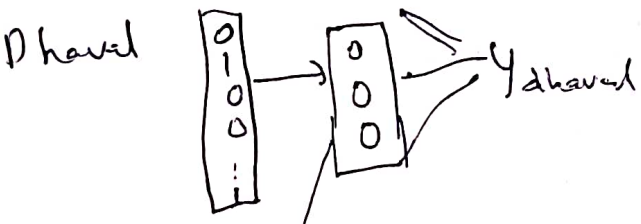
Problem With ANN :

(I) Variable size of input / output Neurons
(II) Too much computation

Named Entity Recognition :

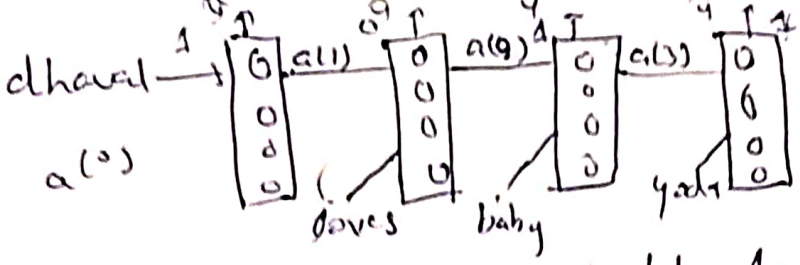
For example "Dhaval loves baby Yoda"
Entity Entity

1 0 1 1



Ydhaval

It needs memory

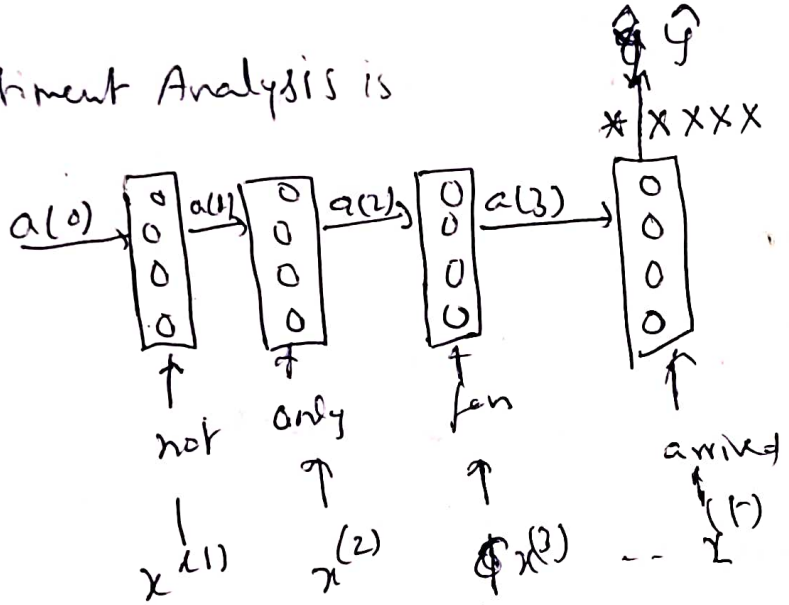


→ MANY to Many RNN

there is only one hidden layers. This is time travel.

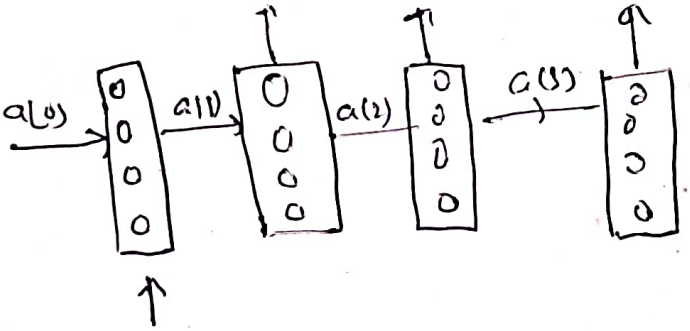
Total loss = Loss 1 + Loss 2 + Loss 3 + Loss 4

Sentiment Analysis is



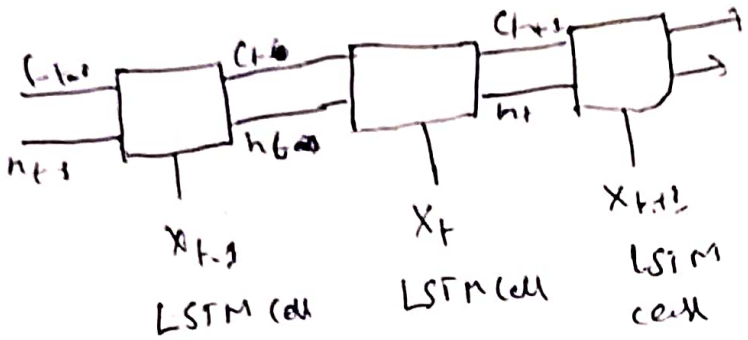
many to one

Writing Poem

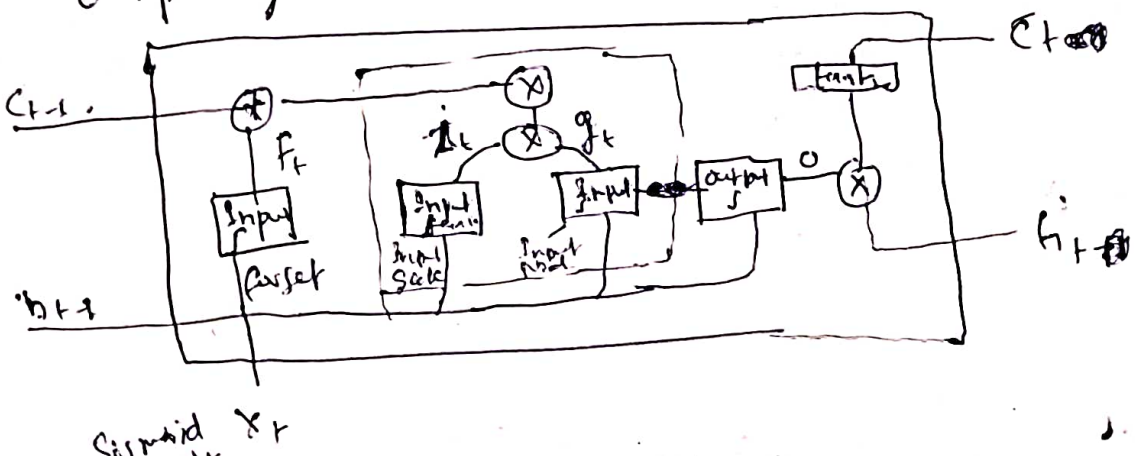


one to Many

LSTM :- LSTM is used to solve the problem of vanishing gradient.



(I) A Simple RNN cell
(II) Cell State Long term memory
(III) Forget gate
(IV) Input gate
(V) Output gate



$$f_t = \sigma(W_f h \cdot h_{t-1}) + (W_f x \cdot X_t) + b_t$$

$$C_t^f = C_{t-1} * f_t$$

$$i_t = \sigma(W_{ih} \cdot h_{t-1}) + W_{ix} \cdot X_t + b_i)$$

$$g_t = \tanh(W_{gt} \cdot h_{t-1}) + (W_{gx} * X_t) + b_g$$

$$C_t^i = i_t * g_t$$

$$C_t = C_t^i + C_t^f$$

tanh = activation function

$$= \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$O_t = \int\left(\left(W_{oh} * h_{t-1}\right) + W_{ox} \cdot X_t\right) + b\right)$$

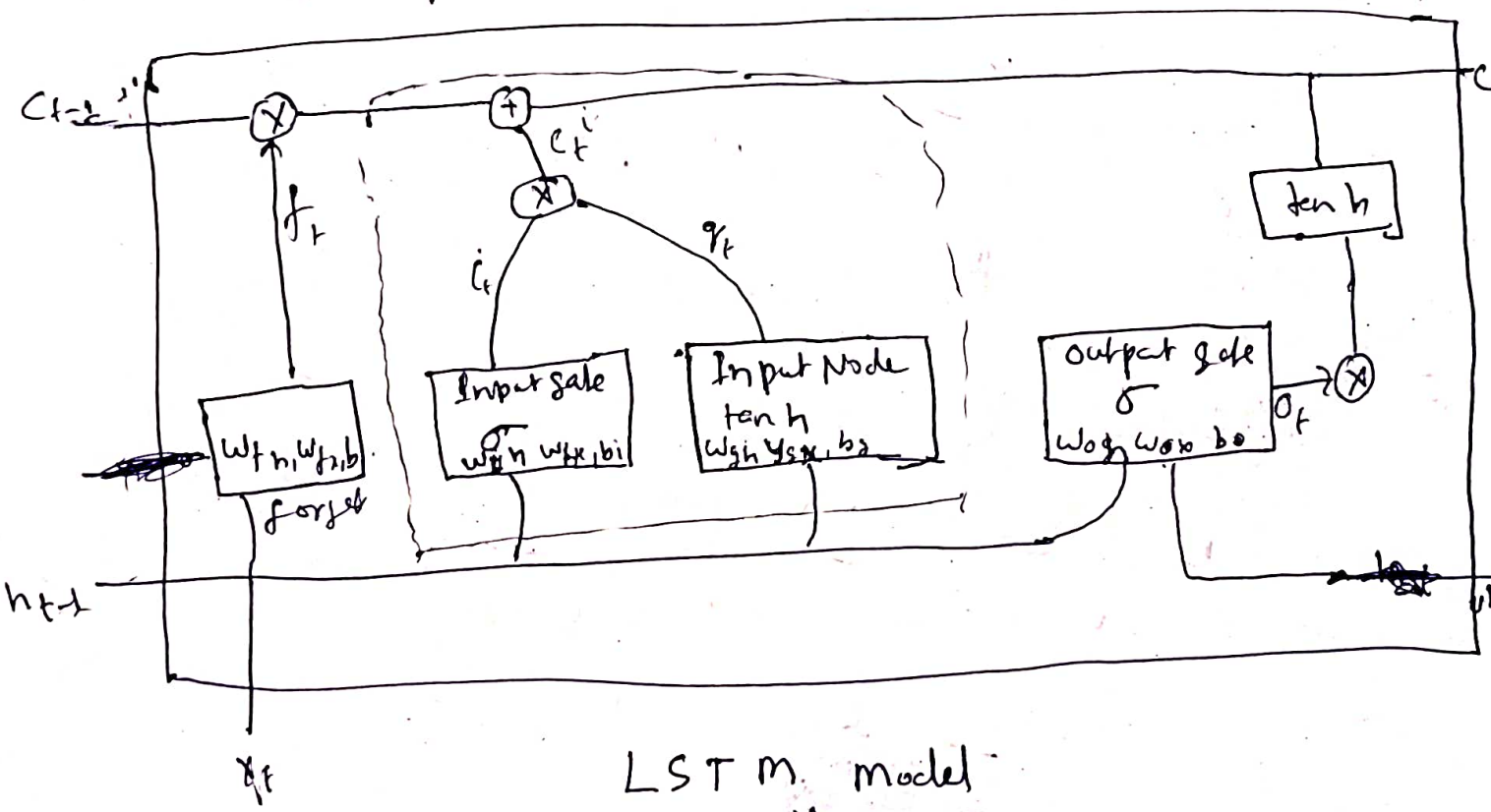
$$h_t = \tanh\left(C_t \times O_t\right)$$

$$w = \begin{cases} w_f \\ w_i \\ w_g \\ w_o \end{cases}$$

$$b = \begin{cases} b_f \\ b_i \\ b_g \\ b_o \end{cases}$$

$$h = \begin{cases} w_{fh} \\ w_{ih} \\ w_{gh} \\ w_{oh} \end{cases}$$

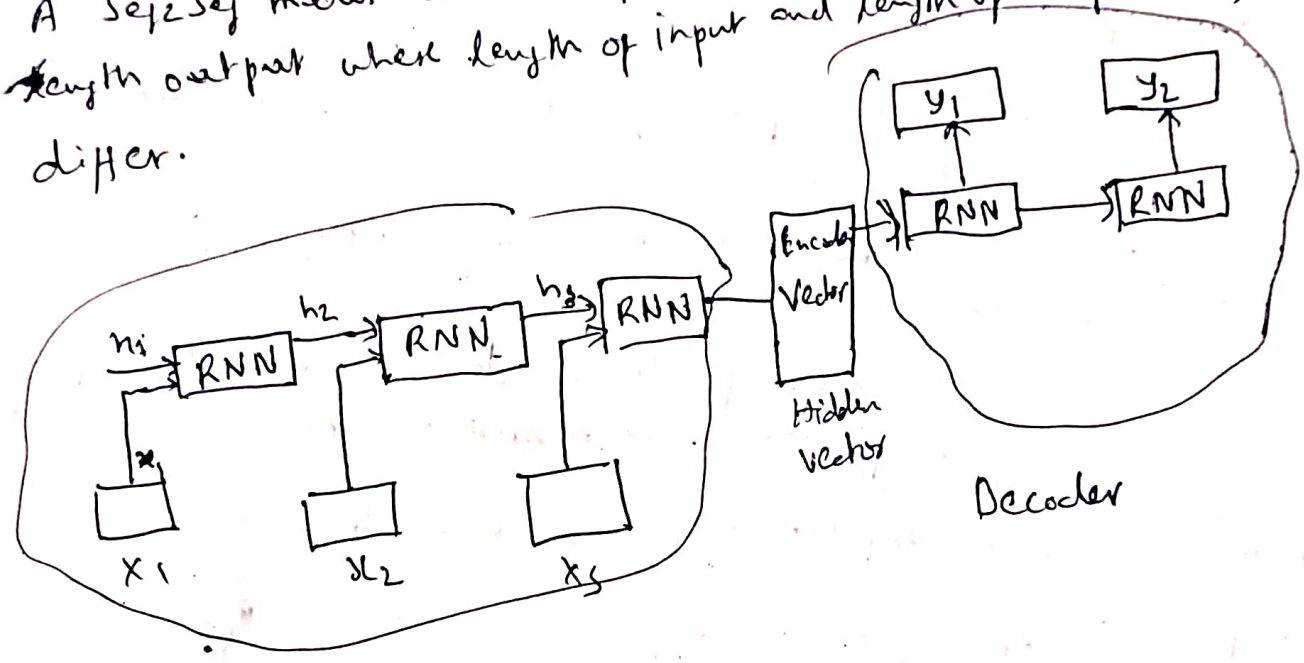
$$g = \begin{cases} f_t \\ i_t \\ g_t \\ O_t \end{cases}$$



LSTM model

Encoder - Decoder - Sequence-to-sequence Architecture

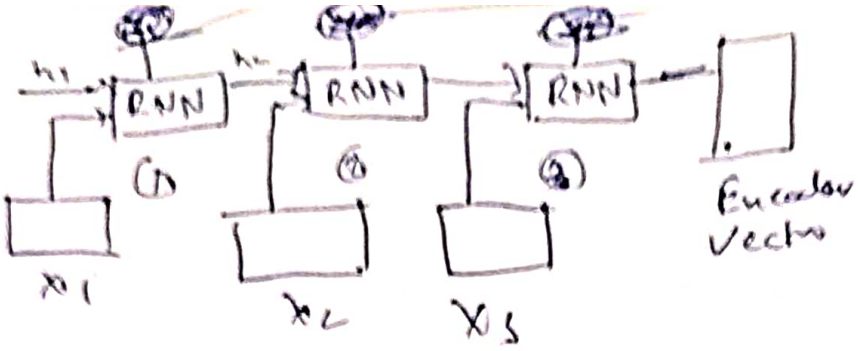
- Sequence to sequence (Seq2Seq) models is a special class of recurrent Neural Network architecture
- Seq2Seq models are used to solve complex language problem
  - (I) Machine translation
  - (II) ~~Questing~~ Question Answering
  - (III) Creating chatbots
  - (IV) Text Summerization (Summary of given text)
- A Seq2Seq model aims to map a fixed-length input with fixed length output where length of input and length of output may differ.



- The encoder will convert the input sequence into a single dimensional vector (hidden vector)
- The decoder will convert the hidden vector into the output sequence.

Encoder

- Multiple RNN cells (LSTM or GRU cells for better performance) can be stacked together to form the encoder. RNN reads each input sequentially
- For every timestamp (each input) t, the hidden state (hidden vector) h is update according to the input at that time stamp $X(t)$



Encoder

- After all the inputs are read by encoder model, the final hidden state of the model represents the context/summary of the whole input sequence.

Example:

- Consider the input sequence "I am a student" to be encoded.
- There will be totally 4 time steps (4 tokens) for the encoder model.
- At each time step, the hidden state h will be updated using the previous hidden state and the current input

- h_0 can be considered as to be zero or randomly.
- Each layer outputs two things
  1. Updated hidden states
  2. Output for each stage.
- The output at each layer is rejected and only hidden states will be propagated to the next layer.

$$h_t = f\left(W^{hh} \cdot h_{t-1} + W^{(hx)} \cdot x_t\right)$$

W_{hh} → weight the recurrent neuron
h_{t-1} → Previous state
W_{hx} → input weight neuron
x_t → Current input

- This encoder vector h_t aims to encapsulate the information for all input element in order to help the decoder make accurate predictions.
- It acts as the initial hidden state of the decoder part of the model.

Decoder

- A stack of several recurrent units where each unit predicts an output y_t at a time step t
- The input for the decoder is the final hidden vector obtained at the end of encoder model.

- Each Recurrent Unit

Accept:

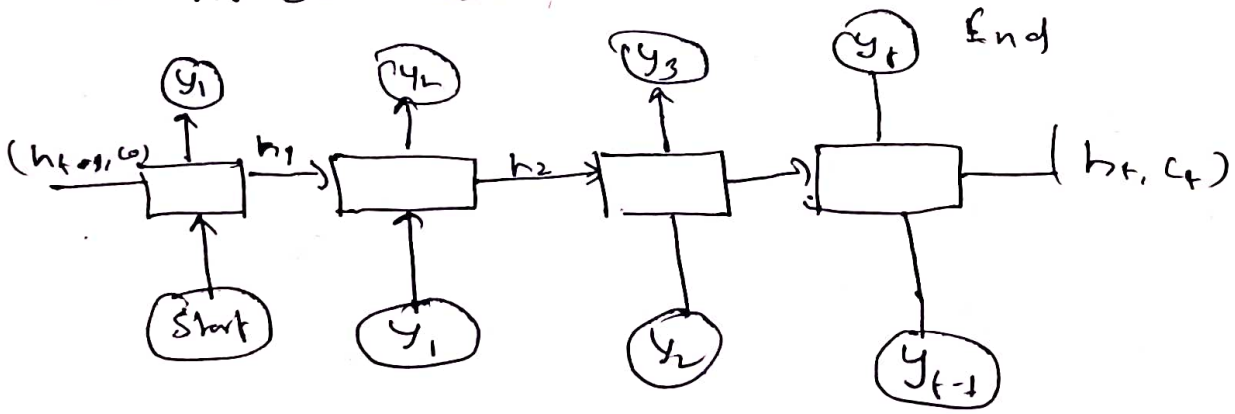
(i) a hidden state from the previous unit h_{t-1}

(ii) Previous layer output y_{t-1}

Produces:

(i) An output as well as

(ii) Its own hidden state



Any hidden state h_t is computed using

$$h_t = f\left(W^{(hh)} h_{t-1}\right)$$

- The output occurred at each time step of decoder is actual output
- The model could predict the output until the end symbol occurs

Output layer

- We use softmax activation function at the output layer
- It is used to produce the probability distribution from a vector of values with the target class of highest probability

$$y_t = \text{Softmax}\left(W^S h_t\right)$$

- Loss is calculated on the predicted output from each time step and the errors are backpropagated through time in order to update the parameters of the network.
- Training the network over a longer period with sufficiently large amount of data results in pretty good predictions