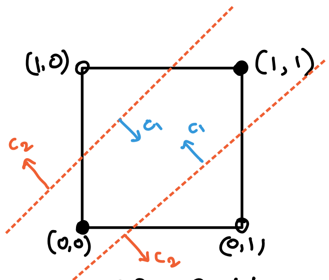
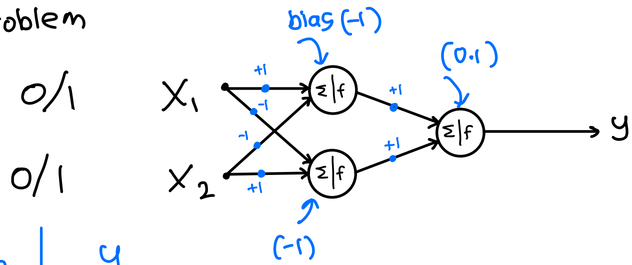


XOR Problem

Cannot be seperated with one line

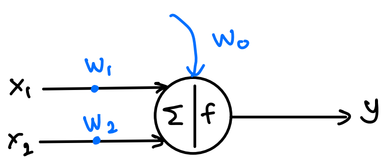
Some problems need more than 1 line.

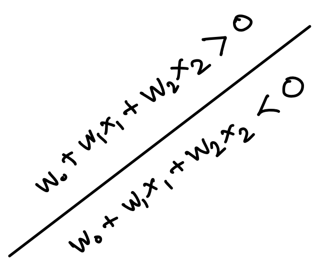


X_1	X_2	y
0	0	0
0	1	1
1	0	1
1	1	0

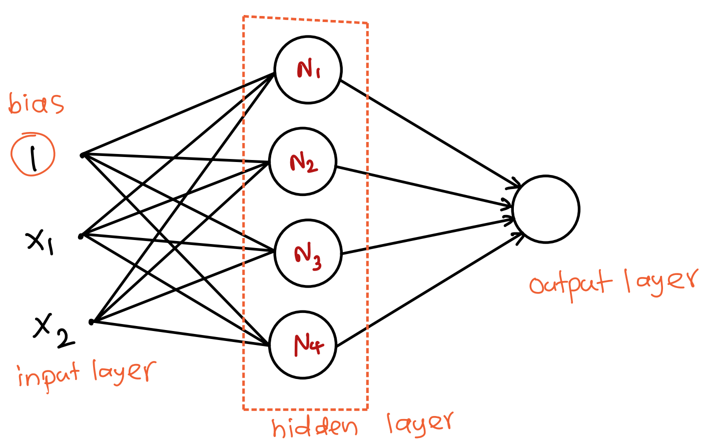
This works if used a "sign" logistic function $f(\cdot)$

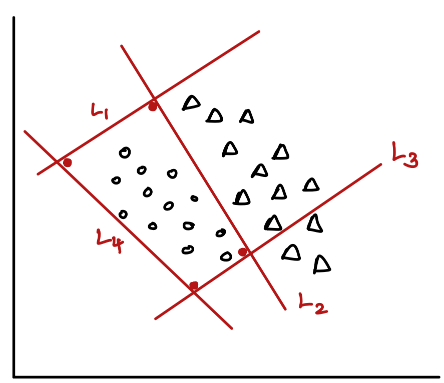
This will enable a non linear seperation via 2 lines. Neurons use the sign function due to its simplicity.



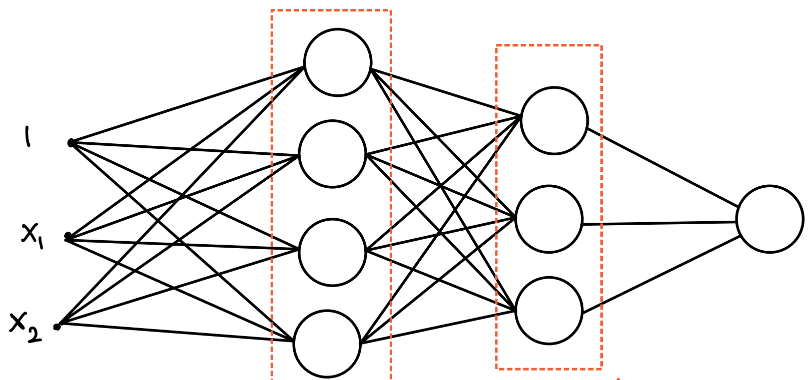


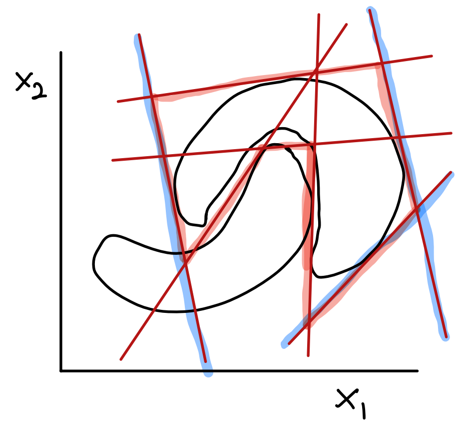
One hidden layer





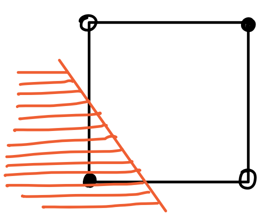
Two hidden layer



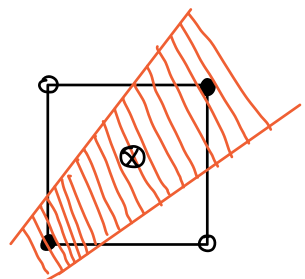


less freedom to locate lines with 2 layers

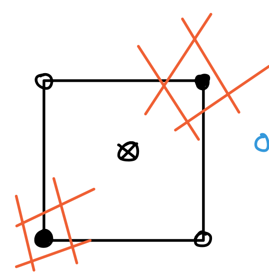
Any shape can be bounded by # of neurons



One neuron cannot solve the problem



2 neurons solves the problem

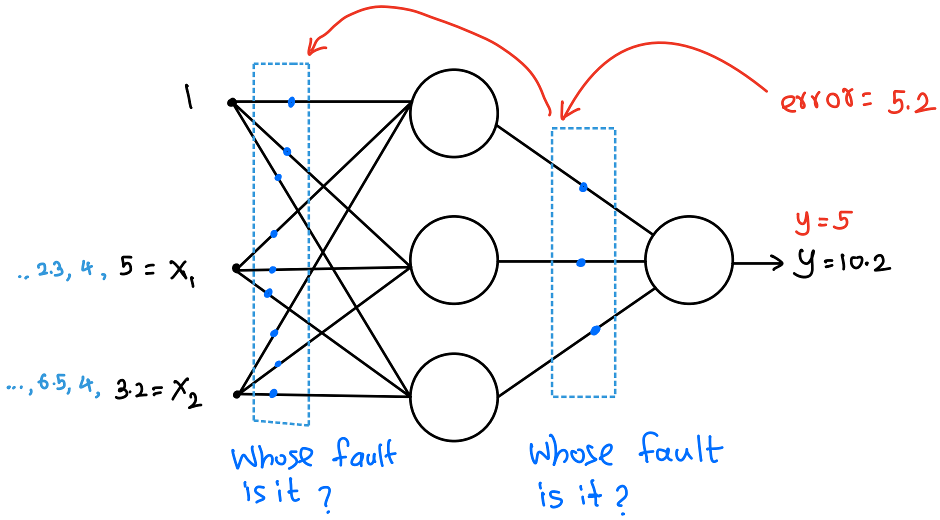


2 layers solves the problem too and more accurately, but do you need that ?

Learning Algorithms

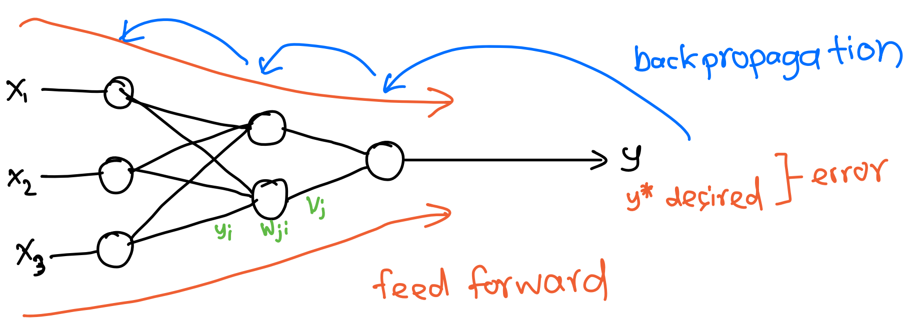
For MLPs (multi layer perceptrons) that we also called feed forward MLPs.

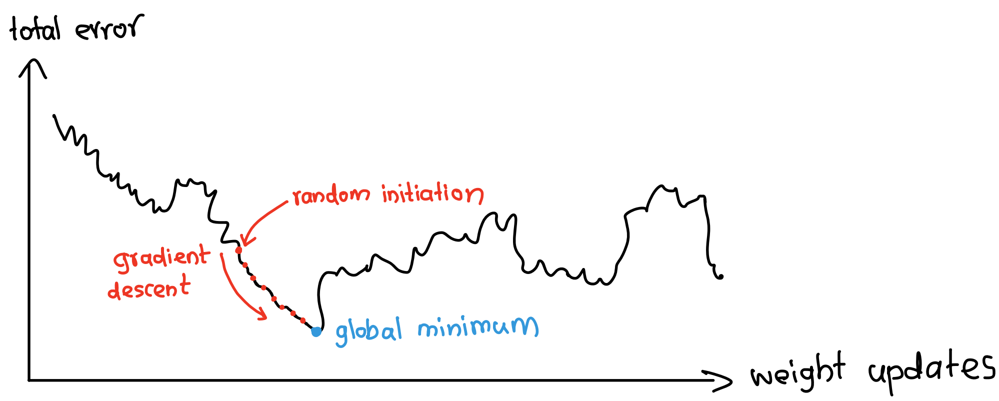
We need solve the Credit Assignment Problem.



Intelligence is updating the weights (= solving the credit assignment problem)

→ Backpropergating the error into the network.





Take a step in the direction resulting in a maximum decrease of the network error E.

This direction is the opposite of gradient of E.

Intelligence is updating the weights.

$$w_{ji}^{(n+1)} = w_{ji}^{(n)} + \Delta w_{ji}^{(n)}$$

$\Delta w_{ji} = -\eta \dfrac{\partial E}{\partial w_{ji}}$ (opposite of gradient of error at local neuron)

$$\eta \in (0,1)$$

The input of the jth neuron: (In hidden layer internal neurons)

$$V_j = \sum_{i=1,2,\dots m} w_{ji} \cdot y_i$$

Using the chain rule:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial V_j} \cdot \frac{\partial V_j}{\partial w_{ji}}$$

Local gradient of the jth neuron:

$$\delta_j = -\frac{\partial E}{\partial V_j}$$

Then from $\dfrac{\partial V_j}{\partial w_{ji}} = y_i$ we get

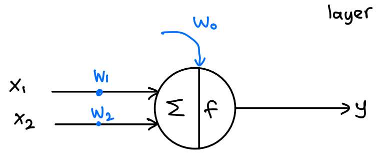
$$\boxed{\Delta w_{ji} = \eta \cdot \delta_j \cdot y_i} \longleftarrow \text{Delta Rule}$$

$y_i \longrightarrow w_{ji} \longrightarrow V_j$

logistic function

desired

$$\delta_j = \begin{cases} f'(V_j)\,(y_j^* - y_j) & \text{if } j \text{ is an output neuron} \\ f'(V_j) \sum\limits_{k \text{ of next layer}} \delta_k w_{jk} & \text{if } j \text{ is an hidden neuron} \end{cases}$$



$y = f(\sum x_i w_i + w_0)$

$f(x)$ is the logistic function,

for instance, $f(x) = \dfrac{1}{1+e^{-ax}}$

$$\frac{\partial}{\partial x} f(x) = \frac{\partial}{\partial x}(1+e^{-ax})^{-1}$$

$$= -1 \cdot (1+e^{-ax})^{-2}(-a e^{-ax})$$

a is just a factor

$$= \frac{a \cdot e^{-ax}}{(1+e^{-ax})^2}$$

$$= a \cdot \frac{1}{(1+e^{-ax})} \cdot \frac{(1+e^{-ax} - 1)}{(1+e^{-ax})}$$

$$= \frac{a}{(1+e^{-ax})} \cdot \left(1 - \frac{1}{1+e^{-ax}}\right)$$

$$= a \cdot f(x) \cdot [1 - f(x)]$$

$f'(x) = a \cdot f(x)[1 - f(x)]$

$f'(v_j) = a\, y_j (1 - y_j)$

Backpropagation

$n = 1$

Initialize weights randomly $w(n)$

while (stopping criterion not satisfied)

for each example $(x, \underline{y^*})$ supervised learning

Run network with x and get y (feed f)
Update weights in backpropagation.
end for loop
$n = n+1$
end while

Backpropagation in Batch mode

Update weights only after all examples have been pushed forward through the network.

$$W_{ji}^{(n+1)} = W_{ji}^{(n)} + \sum_{x = \text{training sample}} \Delta W_{ji}^{x}$$

Training is epoch-by-epoch

Stopping criteria

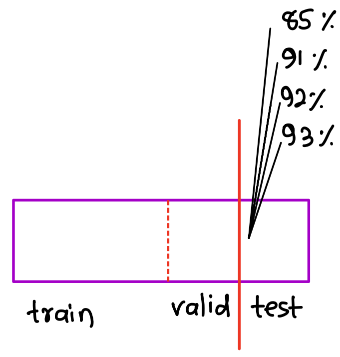
① Look at the MSE change

Network converged ($E \approx 0$) if the absolute rate of change in the average squared error per epoch is sufficiently small i.e, [0.1, 0.01...]

② Generalization based method

Test for generalization after each epoch.

if adquate Generalization ⟶ stop

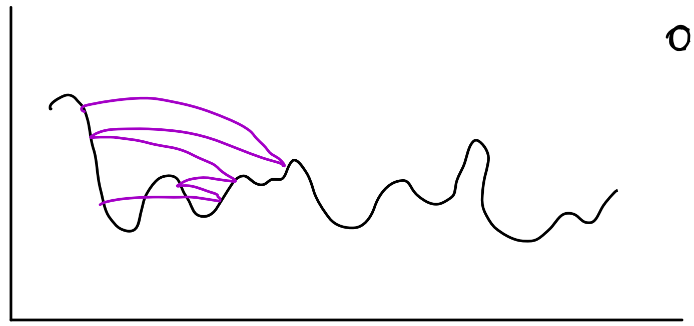


Delta Rule

$\Delta W_{ji}^{(n)} = \eta \, \delta_j(n) \, y_i(n)$

$\eta \longrightarrow 0$: no learning

$\eta \longrightarrow 1$: large changes $\longrightarrow$ unstable (leads to weight oscillation)



$$\underbrace{\Delta W_{ji}(n) = \eta(n) \, \delta_j(n) \, y_i(n) + \alpha(n) \, \Delta W_{ji}(n-1)}_{\text{Generalized Delta rule}}$$

begin (dominant)

$\alpha \in [0,1]$ momentum

end (dominant)

$\delta_j(n) = \frac{-\partial E}{\partial v_j}$

Topology of the network

\# of layers

\# of neurons per layer } entirely task dependent

Mostly done via trial and error

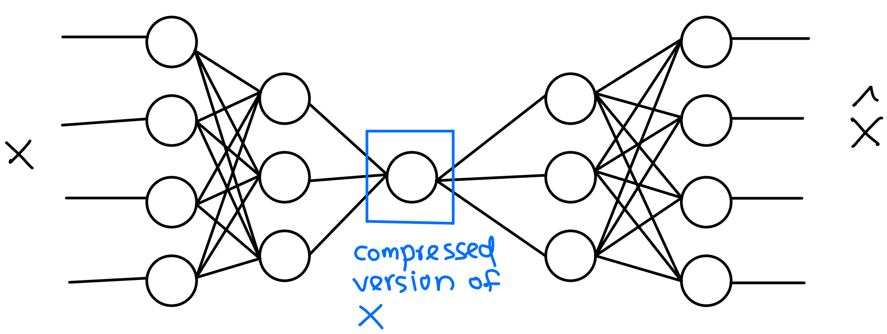
model size ① too small : underfitting

② too large : overfitting

Large net $\longrightarrow$ Remove neurons until performance starts to degrade $\longrightarrow$ Stop

Small net $\longrightarrow$ Add more neurons until performance is acceptable $\longrightarrow$ Stop

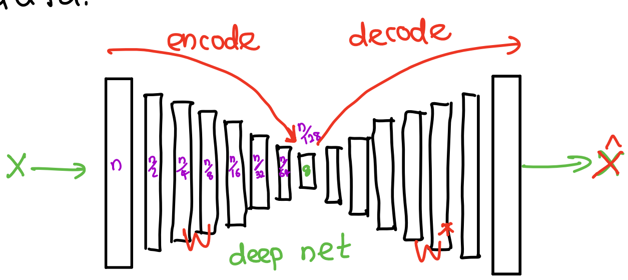
Autoencoders



$\|x - \hat{x}\| \longrightarrow 0$

Bottleneck concept

Force the network to reduce dimentionality of data.



* PCA is easy and more practical

$W^* = W^T$

* backprop doesnt work

inputs $x \in [0,1]^d$

encoding $y \in [0,1]^{d'}$ $\quad d' \ll d$

decoding $z = g(W^* y + b^*)$

Error $L(x,\hat{x}) = \|x - \hat{x}\|^2$

bit vector $L_H(x,\hat{x}) = -\sum_{k=1}^{d} \left[x_k \log \hat{x}_k + (1-x_k)\log(1-\hat{x}_k)\right]$

(error as tradition of entrophy) cross entrophy loss function

How to train deepnets

idea : Layerwise pretraining followed by greedy layerwise supervised training (with fine tuning)