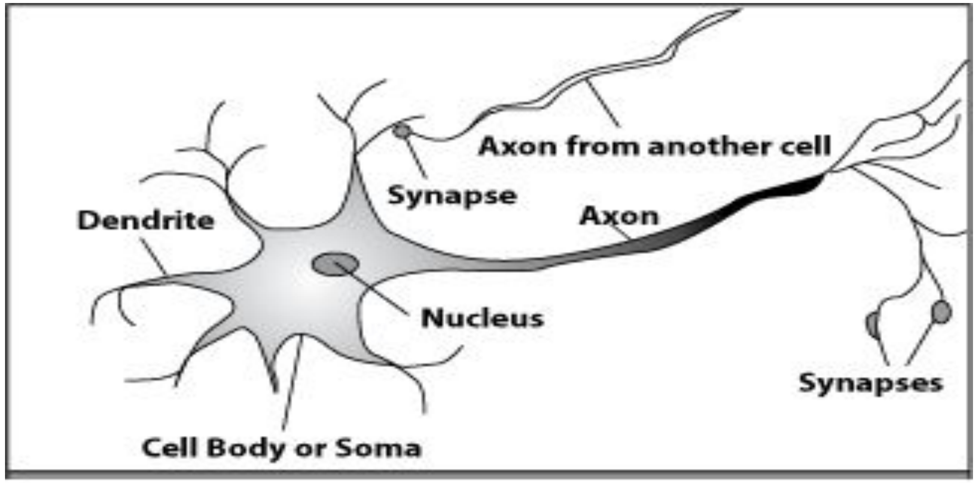
Neural Networks

Perceptron, Multi-Layer Perceptron and Backpropagation

How do our brains work?

- A processing element



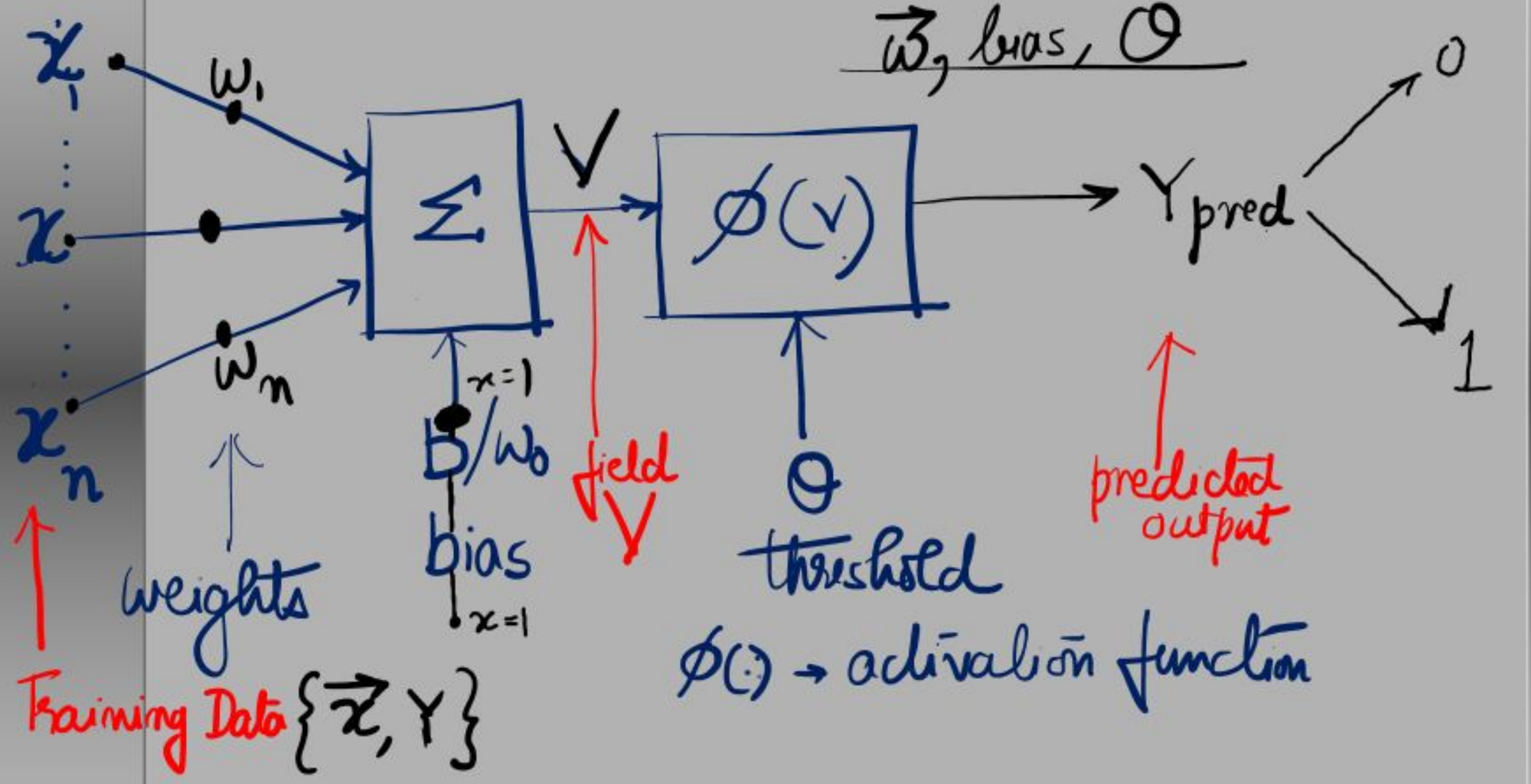
Dendrites: Inputs
Cell body: Processor
Synaptic: Link / Port
Axon: Output Channel

To Emulate

- A biological neuron

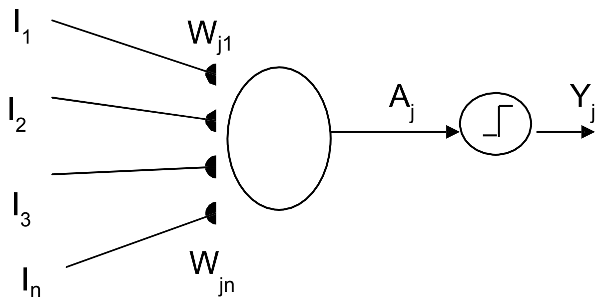
1. Accumulates signals
2. After a threshold gets **activated**
3. Passes signal to next neuron

Arificial Neural Network



A Perceptron

- This vastly simplified model of real neurons is also known as a Perceptron:

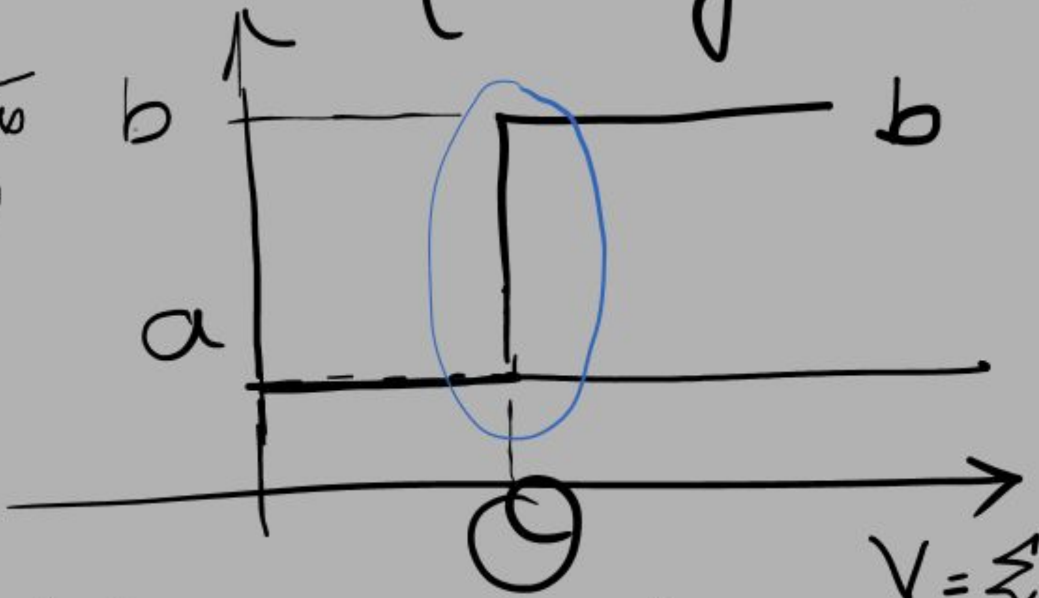


1. A set of synapses (i.e. connections) brings in activations from other neurons
2. A processing unit sums the inputs, and then applies a non-linear activation function
3. An output line transmits the result to other neurons

Activation functions

$$\text{Step} = \phi(v) = \begin{cases} a & \text{if } v < \theta \\ b & \text{if } v \geq \theta \end{cases}$$

Parameters
a, b, θ



$V = \sum w_i x_i + b$

- Computationally simple
- a, b : parameters

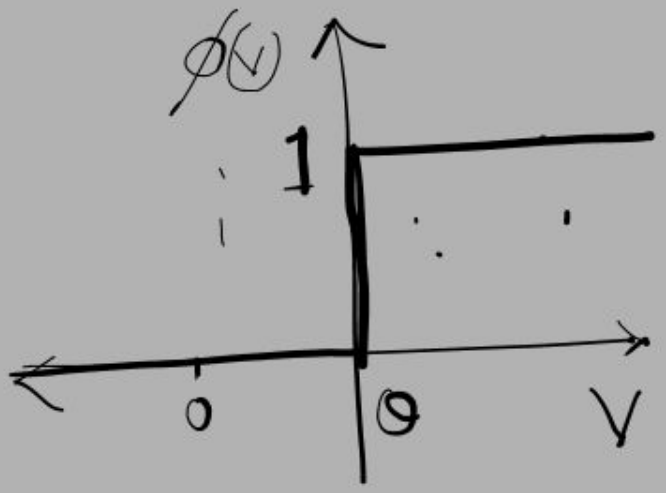
SIGN

Parameter θ

$$\phi(v) = \begin{cases} 0 & v < \theta \\ 1 & v \geq \theta \end{cases}$$

$$V = \sum_i w_i x_i + b$$

$$V = \sum_{i=0,n} w_i x_i$$



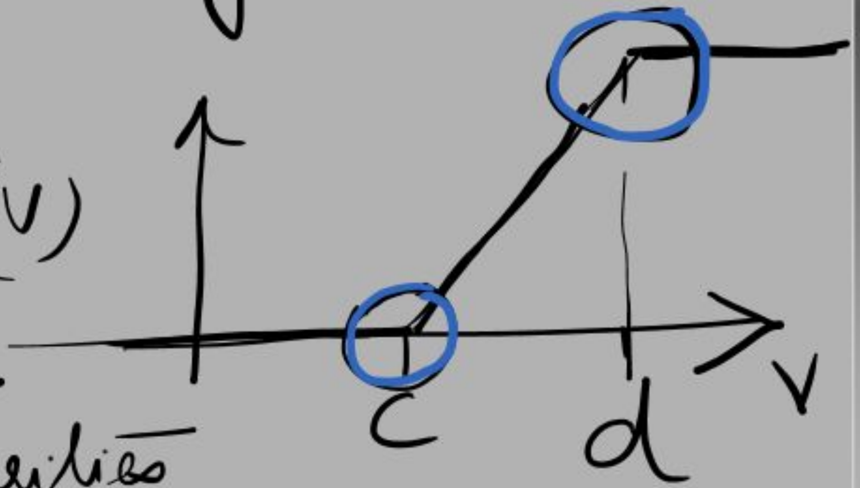
Ramp Activation

Ramp

$$\phi(v) = \begin{cases} 0 & \text{if } v < c \\ 1 & \text{if } v > d \\ v & \text{if } c \le v \le d \end{cases}$$

parameter are c, d

$\phi(v)$



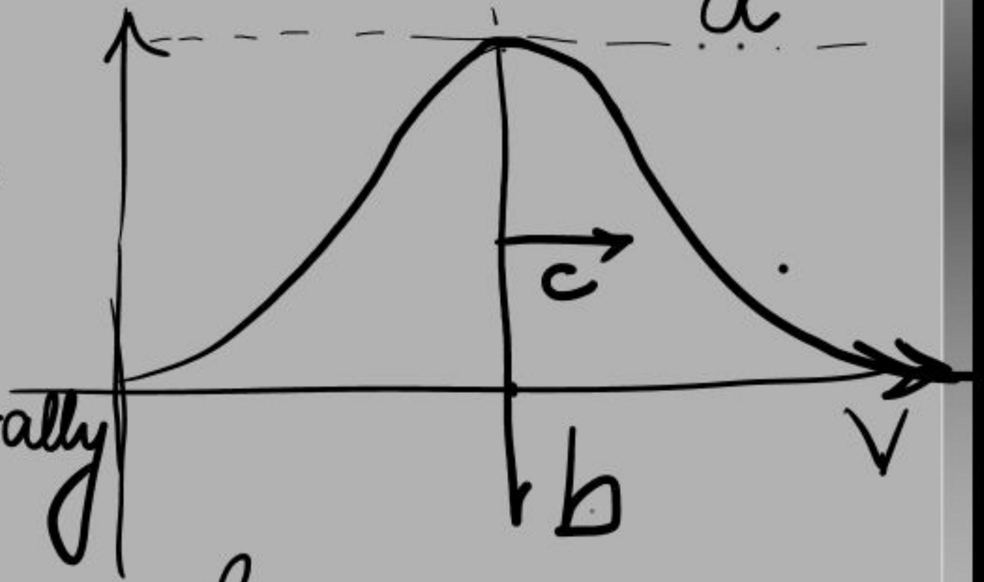
piecewise linear

○ non-linearities

Gaussian Activation Fundtion

Gaussian

$$\phi(v) = a e^{-\frac{(v-b)^2}{2c^2}}$$

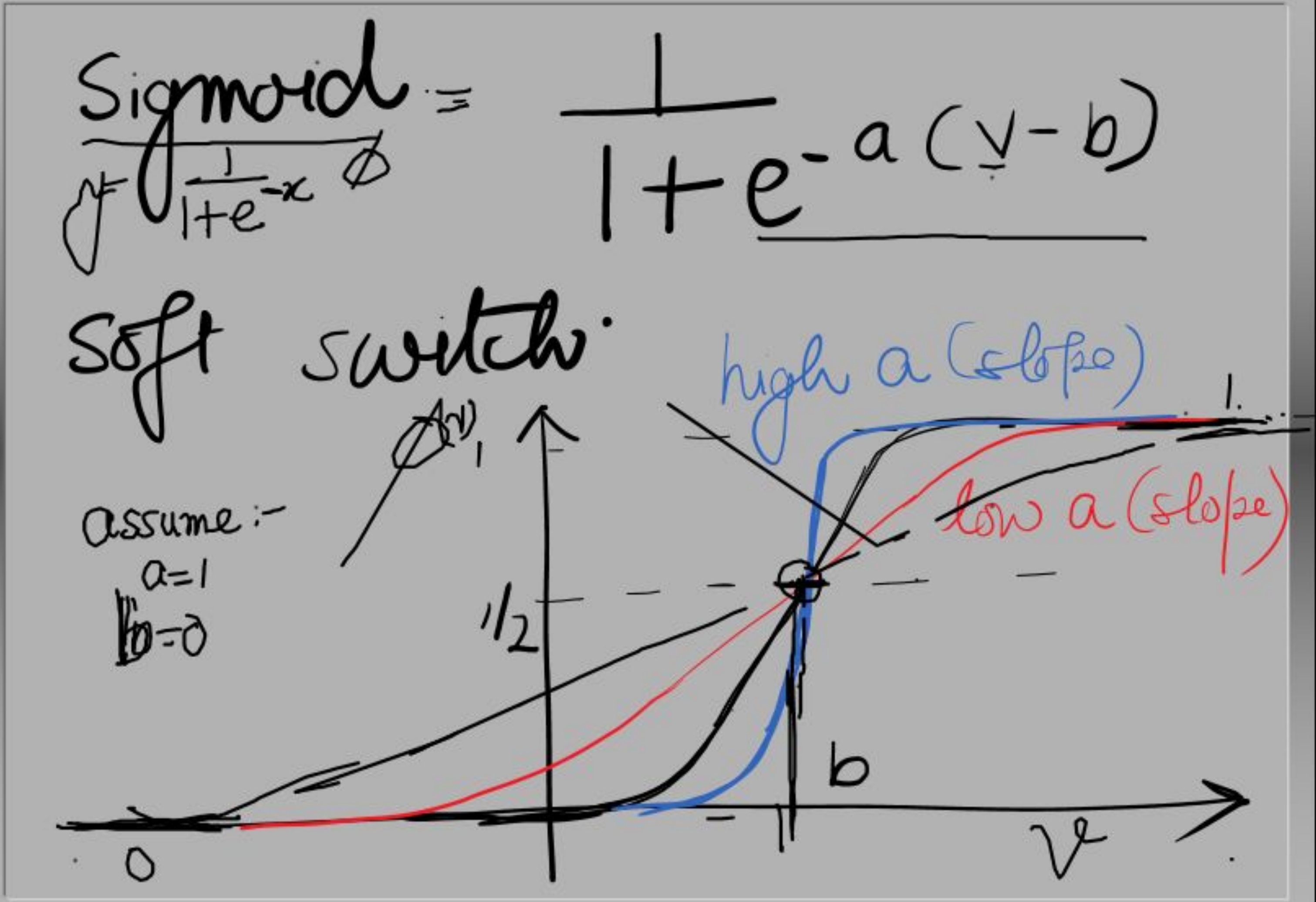


1) a, b, c are parameters

2) Computationally complex

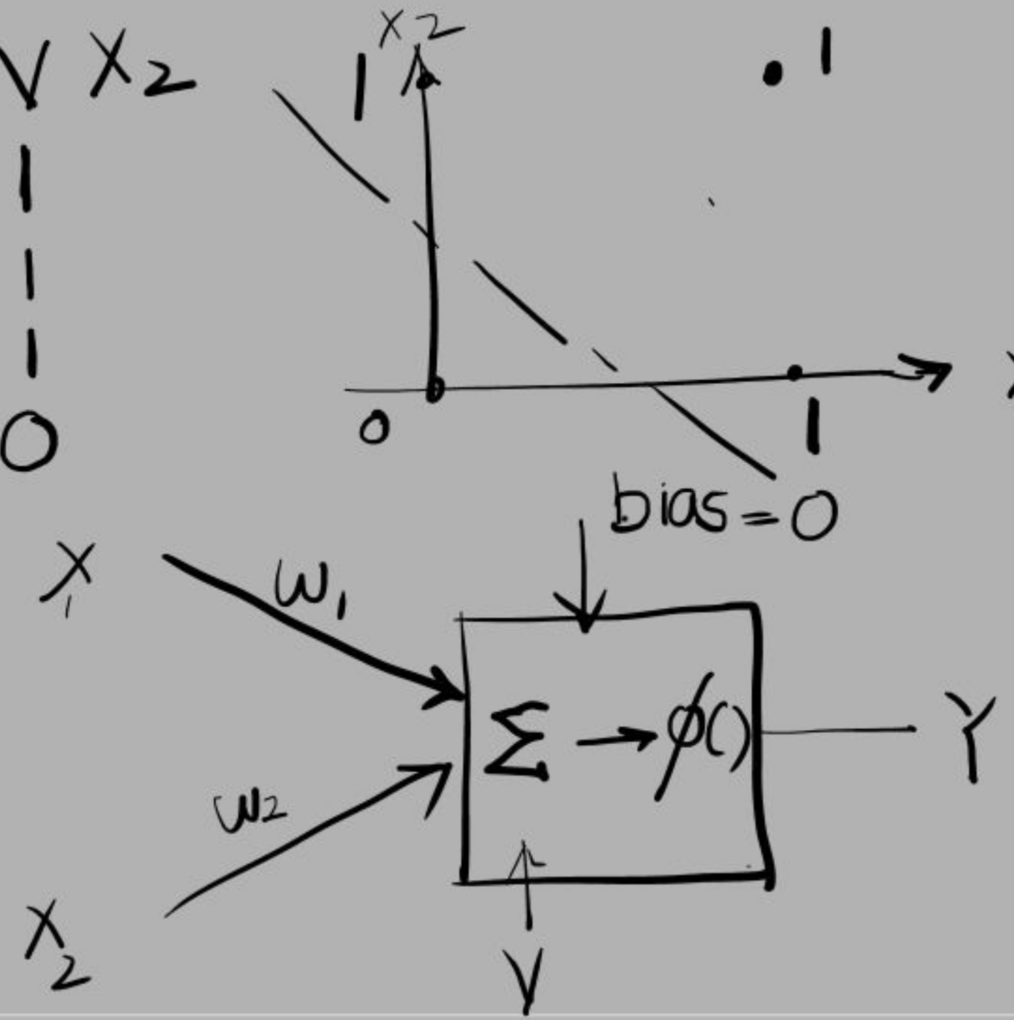
3) Smoothly non-linear

Sigmoid Activation



Perceptron example

X_1	X_2	$X_1 \vee X_2$
1	1	1
1	0	1
0	1	1
0	0	0



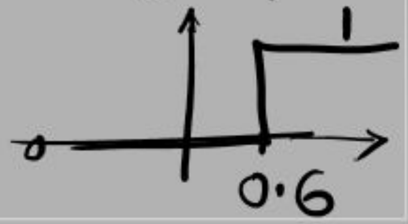
det

$\phi(\cdot)$ - STEP

$\theta = 0.6$

$a = 0$

$b = 1$



Run each training example EPOCH 1

Let $w_{1i} = w_{2i} = 0.5 \rightarrow$ Random

$T_1 = \langle 1, 1 \rangle \quad Y = 1$

$Y_{pred} = \phi(0.5 \times 1 + 0.5 \times 1 + 0) = \phi(1) = 1$

CORRECT OUTPUT
DO NOTHING.

$T_2 = \langle 1, 0 \rangle \quad Y = 1$

$Y_{pred} = \phi(0.5 \times 1 + 0.5 \times 0 + 0) = 0$

INCORRECT OUTPUT
ADJUST w, threshold

Training weights and threshold

$$Error = Y - Y_{pred} = 1 - 0 = 1 \qquad \underline{w \uparrow} \quad \underline{\theta \downarrow}$$

$$w_1 = 0.5 + \eta \times Error \times x_i \qquad \eta \rightarrow \text{learning rate}$$

$$= 0.5 + 0.2 \times 1 \times 1 = 0.7 \checkmark$$

$$w_2 = 0.5 + 0.2 \times 1 \times 0 = 0.5 \checkmark$$

$$\theta_{new} = \underline{0.6} - \underbrace{0.2 \times 1}_{\eta \times Error} = 0.4$$

$$T_3 = \langle 0, 1 \rangle \quad Y = 1$$

$$Y_{pred} = \phi(0.7 \times 0 + 0.5 \times 1 + 0) = \varnothing(0.5) = 1$$

NOW CORRECT

Go to next epoch

$T_4 \ \langle 0, 0\rangle, \ Y = 0$

$Y_{pred} = \phi(0.7 \times 0 + 0.5 \times 0 + 0) = 0$ CORRECT

EPOCH2

$T_1 \ \langle 1,1\rangle\langle 1\rangle \rightarrow \phi(0.7 \times 1 + 0.5 \times 1) = 1$

$T_2 \ \langle 1,0\rangle\langle 1\rangle \rightarrow \phi(0.7 \times 1 + 0.5 \times 0 + 0) = 1$

$T_3 \ \langle 0,1\rangle\langle 1\rangle \rightarrow \phi(0.7 \times 0 + 0.5 \times 1 + 0) = 1$

$T_4 \ \langle 0,0\rangle\langle 0\rangle \rightarrow \phi(0.7 \times 0 + 0.5 \times 0 + 0) = 0$

Add O.K.

Universal Approximator Function

Approxim.

define:

$$F(x_1,\dots,x_{m_0}) = \sum_{i=1}^{m_1} a_i \phi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i\right)$$

as an approximate realisation of function f(•); that is:

$$|F(x_1,\dots,x_{m_0}) - f(x_1,\dots,x_{m_0})| < \varepsilon$$

for all $x_1, \dots, x_{m0}$ that lie in the input space.

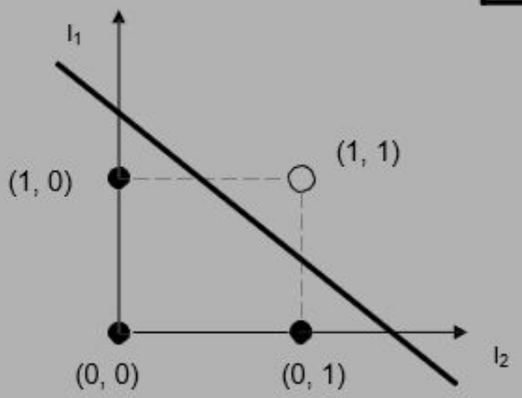
Some Decision Boundaries for AND and OR

We can now plot the decision boundaries of our logic gates

AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1

AND

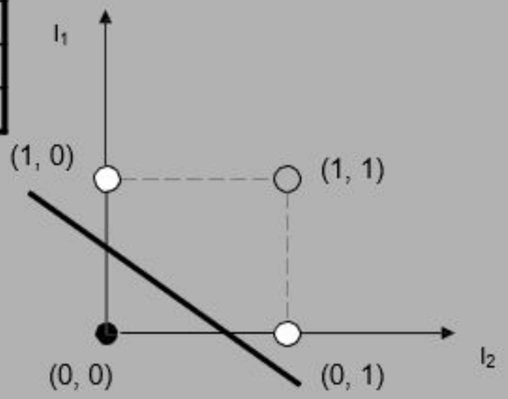
w1=1, w2=1, θ =1.5



OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1

OR

w1=1, w2=1, θ=0.5



$\phi(w_1 x_1 + w_2 x_2)$

Decision Boundary for XOR

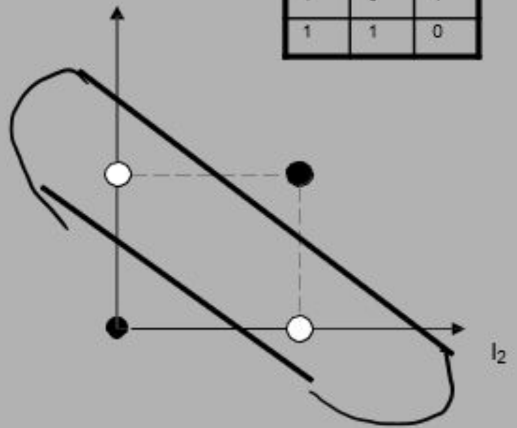
The difficulty in dealing with XOR is rather obvious. We need two straight lines to separate the different outputs/decisions:

I_1

I_1

XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0

I_2



Solution: either change the transfer function so that it has more than one decision boundary, or use a more complex network that is able to generate more complex decision boundaries.

ANN Architectures

Mathematically, ANNs can be represented as weighted directed graphs. The most common ANN architectures are:

Single-Layer Feed-Forward NNs: One input layer and one output layer of processing units. No feedback connections (e.g. a Perceptron)

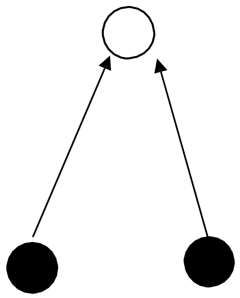
Multi-Layer Feed-Forward NNs: One input layer, one output layer, and one or more hidden layers of processing units. No feedback connections (e.g. a Multi-Layer Perceptron)

Recurrent NNs: Any network with at least one feedback connection. It may, or may not, have hidden units

Examples of Network Architectures

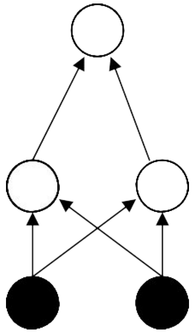
Single Layer

Feed-Forward



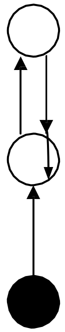
Multi-Layer

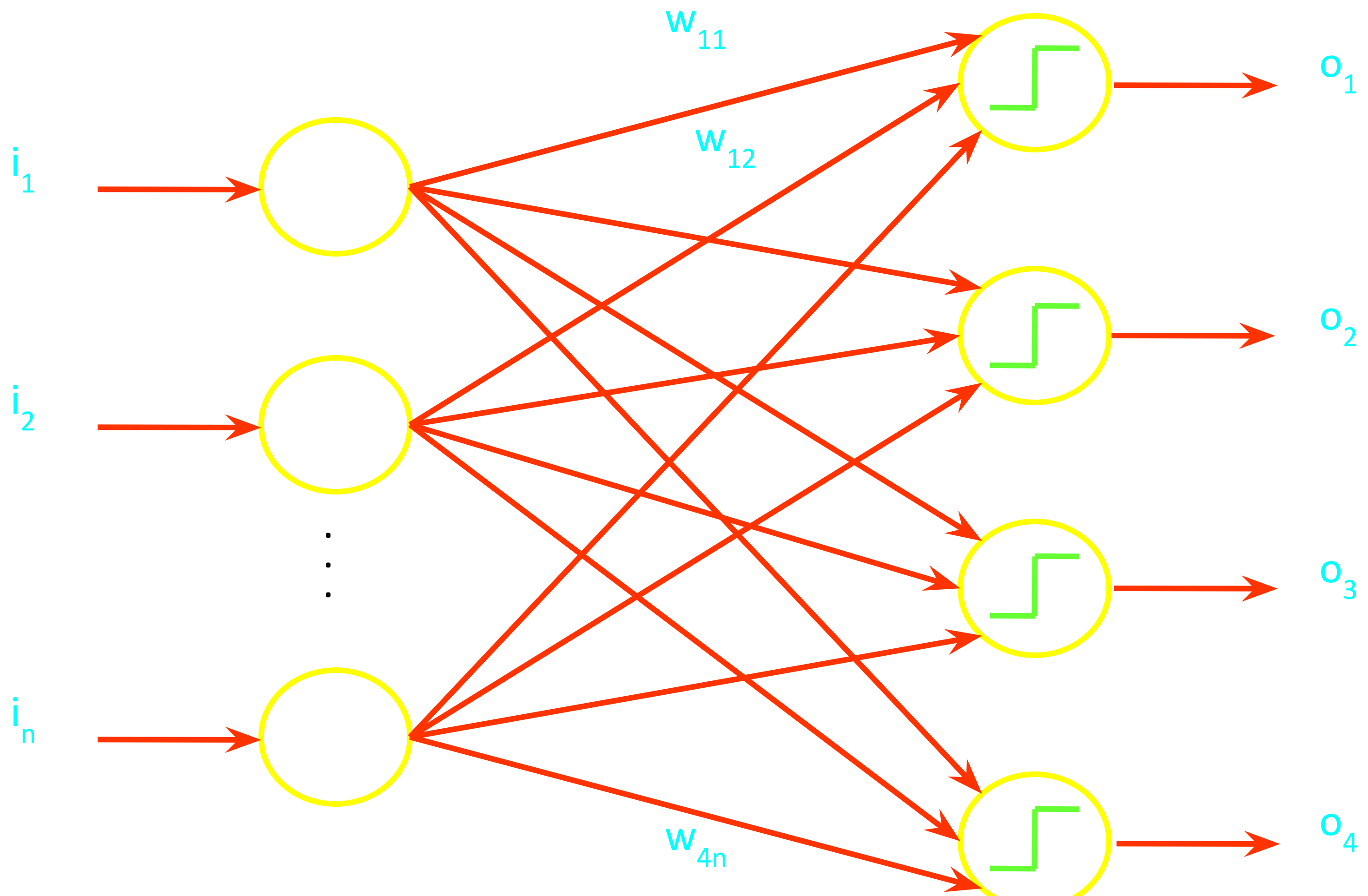
Feed-Forward



Recurrent

Network





- A four-node perceptron for a four-class problem in n-dimensional input space

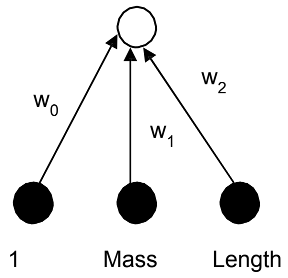
A typical neural network application is classification. Consider the simple example of classifying trucks given their masses and lengths:

Mass	*Length*	*Class*
10.0	6	Lorry
20.0	5	Lorry
5.0	4	Van
2.0	5	Van
2.0	5	Van
3.0	6	Lorry
10.0	7	Lorry
15.0	8	Lorry
5.0	9	Lorry

How do we construct a neural network that can classify any Lorry and Van?

For our truck example, our inputs can be direct encodings of the masses and lengths.

Class=sgn(w_1.Mass+w_2.Length + bias)



Perceptron Learning Rule

1. Initialize weights at random
2. For each training pair/pattern (**x**, $\mathbf{y}_{\mathbf{target}}$)

- Compute output y
- Compute error, $\delta = (y_{target} - y)$
- Use the error to update weights as follows:

  $w_{new} = w_{old} + \eta * \delta * x$

where is called the **learning rate** or **step size** and it determines how smoothly the learning process is taking place.

3. Repeat 2 until convergence (i.e. error δ is zero)

The **Perceptron Learning Rule** is then given by

$$w_{new} = w_{old} + \eta * \delta * x$$

where

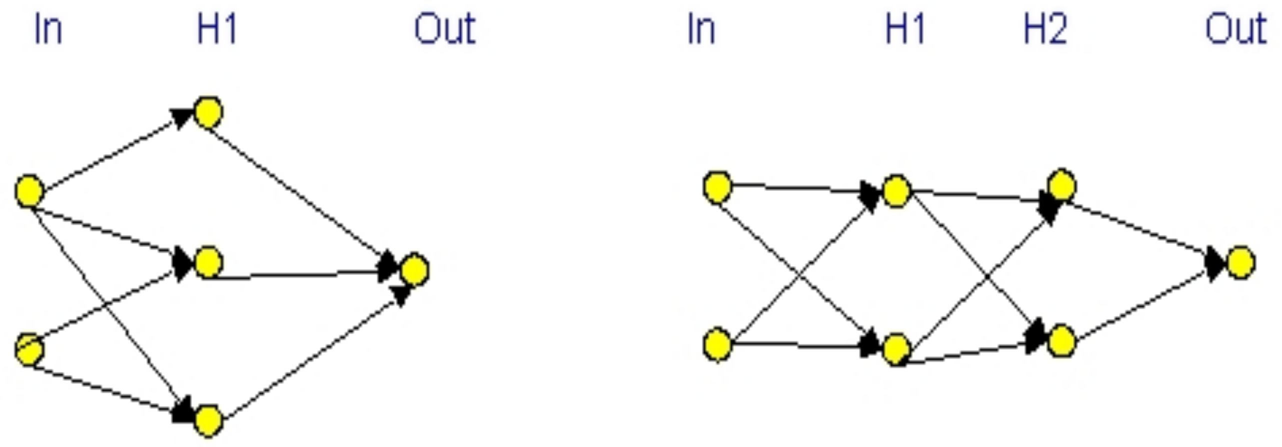
$$\delta = (y_{target} - y)$$

Back Propagation

Multi Layer Perceptron

MLP Model

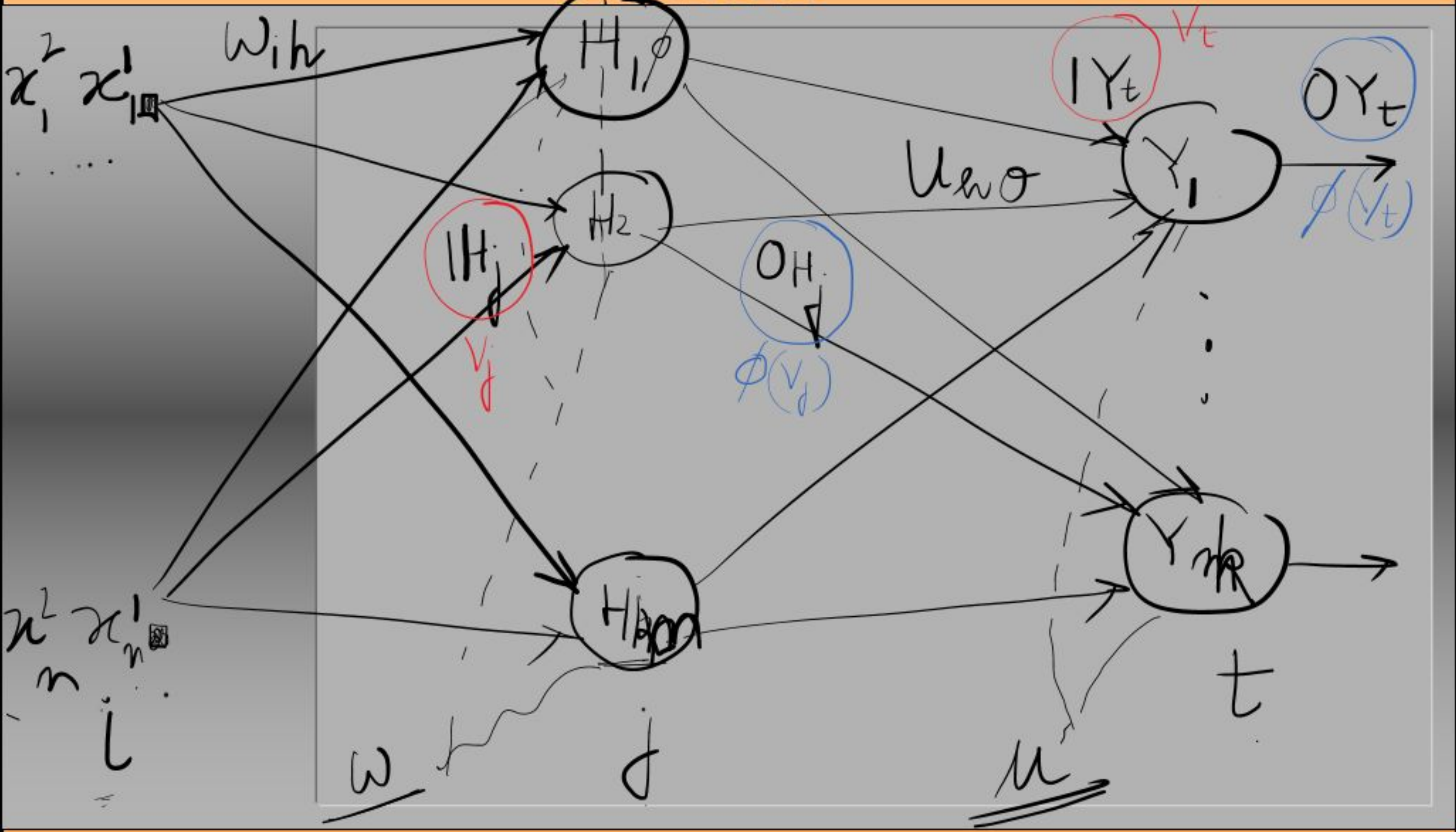
- "Neurons" are positioned in layers. There are Input, Hidden and Output Layers



A. Single Hidden Layer

B. Two Hidden Layers

FEEDFORWARD NEURAL NETWORK WITH HIDDEN LAYER



Multi Layer Perceptron Output

MLP Model

- The output y for j^{th} neuron is calculated by:

$$y_j(n) = \varphi_j(v_j(n)) = \varphi_j\Bigg(\underbrace{\sum_{i=0}^{m} w_{ji}(n)\, y_i(n)}_{V}\Bigg)$$

Where $w_0(n)$ is the **bias**.

- The function $\phi_j(\bullet)$ is a *sigmoid* function.

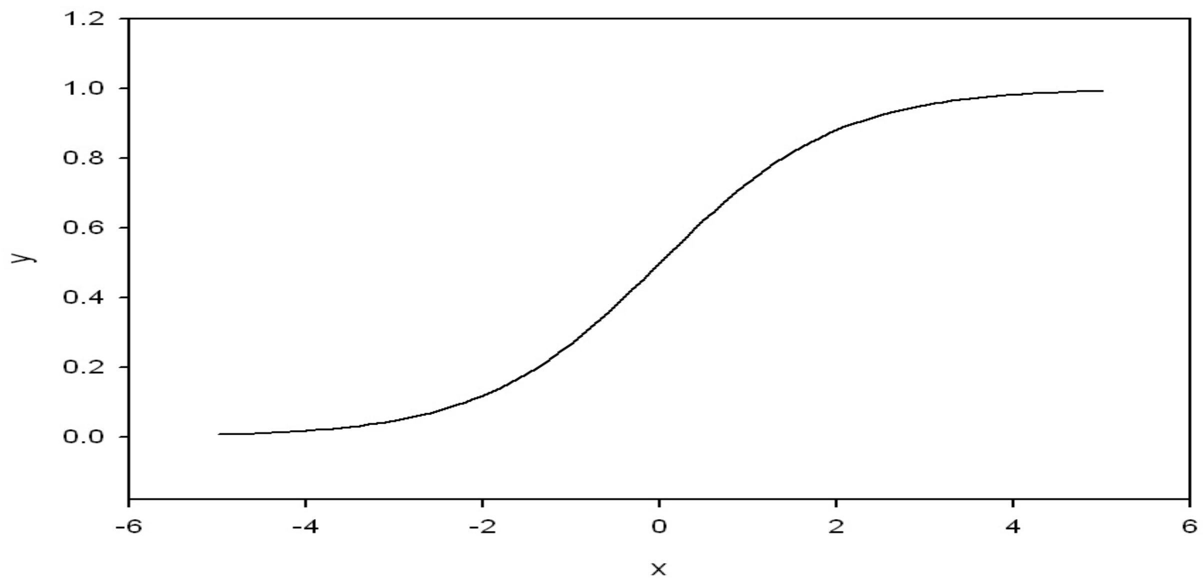
$$\frac{1}{1+e^{-V}}$$

Transfer Functions

MLP Model

- The *logistic sigmoid*:

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

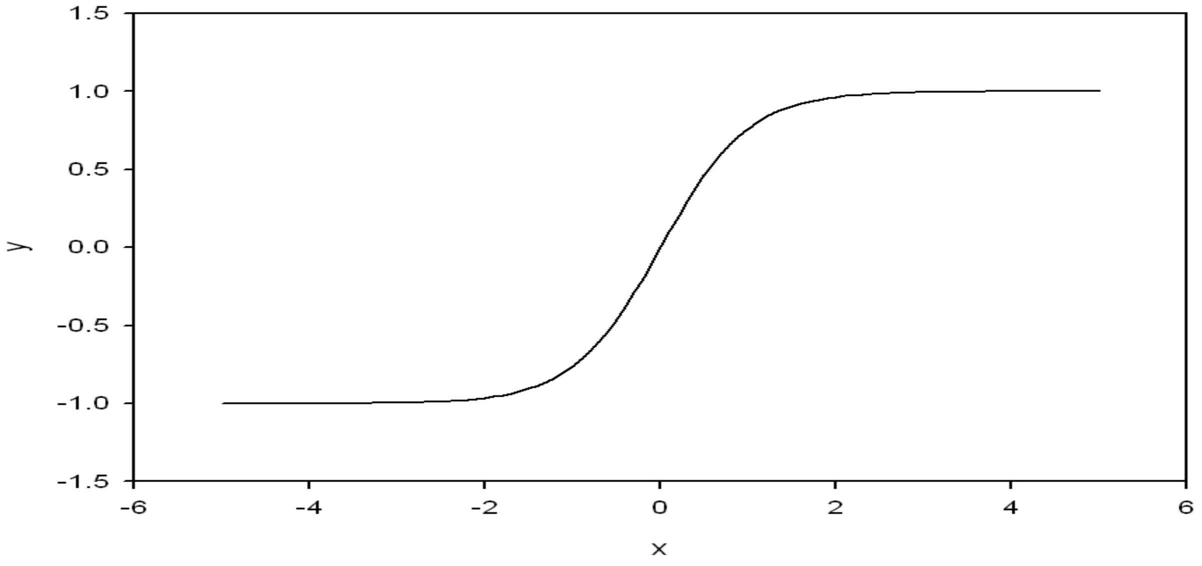


Transfer Functions II

MLP Model

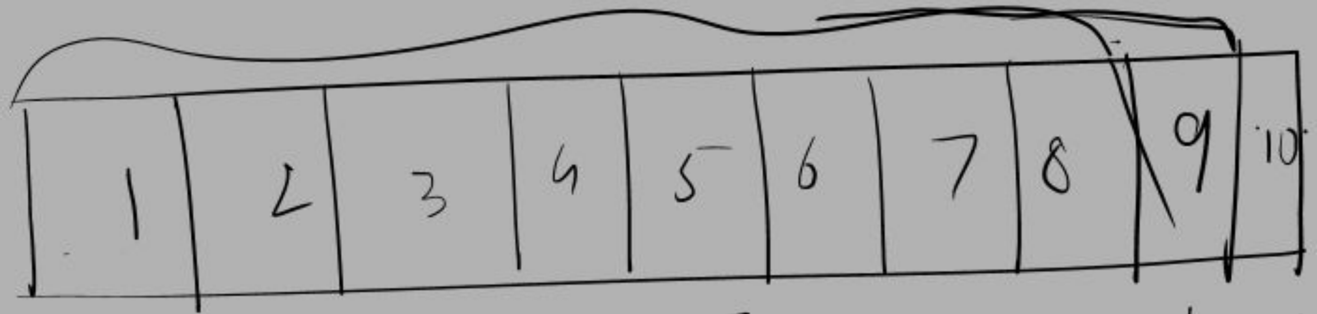
- The *hyperbolic tangent sigmoid*:

$$y = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{\frac{(\exp(x) - \exp(-x))}{2}}{\frac{(\exp(x) + \exp(-x))}{2}}$$



- **Training Set**
  A collection of input-output patterns that are used to train the network
- **Testing Set**
  A collection of input-output patterns that are used to assess network performance
- **Learning Rate-η**
  A scalar parameter, analogous to step size in numerical integration, used to set the rate of adjustments

10 fold cross validation



→ 1-9 → training, 10 testing

1 test 2-10 training

2 test 1, 3, 10 "

⋮

- Total-Sum-Squared-Error (TSSE)

$$TSSE = \frac{1}{2} \sum_{patterns} \sum_{outputs} (desired - actual)^2$$

- Root-Mean-Squared-Error (RMSE)

$$RMSE = \sqrt{\frac{2 * TSSE}{\# \, patterns * \# \, outputs}}$$

- Randomly choose the initial weights
- While error is too large
  - For each training pattern (presented in random order)
    - Apply the inputs to the network
    - Calculate the output for every neuron from the input layer, through the hidden layer(s), to the output layer
    - Calculate the error at the outputs
    - Use the output error to compute error signals for pre-output layers
    - Use the error signals to compute weight adjustments
    - Apply the weight adjustments
  - Periodically evaluate the network performance

Details of Learning Algorithm

BP Algorithm

- $\Im=\{\mathbf{x}(n),\mathbf{d}(n)\}$, $n=1,\ldots,N$ is given. $\mathbf{x}(n)$
- d(n) is the *desired response* vector of dimension M
- Thus an *error signal*, $e_j(n)=d_j(n)-y_j(n)$ can be defined for the output neuron j.

BP Algorithm

- We can derive a learning algorithm for an MLP by assuming an optimization approach which is based on the **steepest descent direction**, I.e.
- $$\mathbf{\Delta w}(n) = -\eta \mathbf{g}(n)$$
- Where **g**(n) is the gradient vector of the COST / LOSS function
- η is the *learning rate*.

SUM OF SQUARED ERRORS (SSE) LOSS FUNCTION

BP Algorithm

- **back-propagation**
  - Assume that we define a SSE instantaneous cost function (I.e. per example) as follows:

$$\mathrm{E}(n) = \frac{1}{2} \sum_{j \in C} e_j^{\,2}(n)$$

Where C is the set of all *output neurons*.

  - If we assume that there are N examples in the set $\Im$ then the *average squared error* is:

$$\mathrm{E}_{av} = \frac{1}{N} \sum_{n=1}^{N} \mathrm{E}(n)$$

BATCH AND STOCHASTIC MODES

BP Algorithm

BP Algorithm

- In the case of E_{av} we have the **Batch** mode of the algorithm (all N data) .
- In the case of E(n) we have the **Online** or **Stochastic** mode of the algorithm (single data)

$$e_j(n)=d_j(n)-y_j(n)$$

- Assume that we use the online mode for the rest of the calculation. Error is:

The gradient is defined as:

$$g(n) = \frac{\partial E(n)}{\partial w_{ji}(n)}$$

APPLYING CHAIN RULE

BP Algorithm

- Using the chain rule of calculus we can write:

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

- We calculate the different partial derivatives as follows:

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

$$\mathrm{E}(n) = \frac{1}{2} \sum_{j \in C} e_j^{\,2}(n)$$

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

$e_j(n)=d_j(n)-y_j(n)$

CHAIN RULE

BP Algorithm

- And,

$$y_j(n) = \varphi_j(v_j(n)) = \varphi_j\left(\sum_{i=0}^{m} w_{ji}(n)y_i(n)\right)$$

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

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

- Combining all the previous equations we get finally:

$$\Delta w_{ij}(n) = -\eta \frac{\partial \mathrm{E}(n)}{\partial w_{ji}(n)} = \eta e_j(n)\phi_j'(v_j(n))y_i(n)$$

WEIGHT ADJUSTMENT AND LOCAL GRADIENT

BP Algorithm

- The equation regarding the weight corrections can be written as:

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

Where $\delta_j(n)$ is defined as the *local gradient* and is given by:

$$\delta_j(n) = -\frac{\partial \mathrm{E}(n)}{\partial v_j(n)} = -\frac{\partial \mathrm{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = e_j(n) \phi_j'(v_j(n))$$

- We need to distinguish two cases:
  - j is an output neuron
  - j is a hidden neuron

CHAIN RULE

BP Algorithm

- And,

$$y_j(n) = \varphi_j(v_j(n)) = \varphi_j\left(\overbrace{\sum_{i=0}^{m} w_{ji}(n)\, y_i(n)}^{v}\right)$$

o/p of prev layer

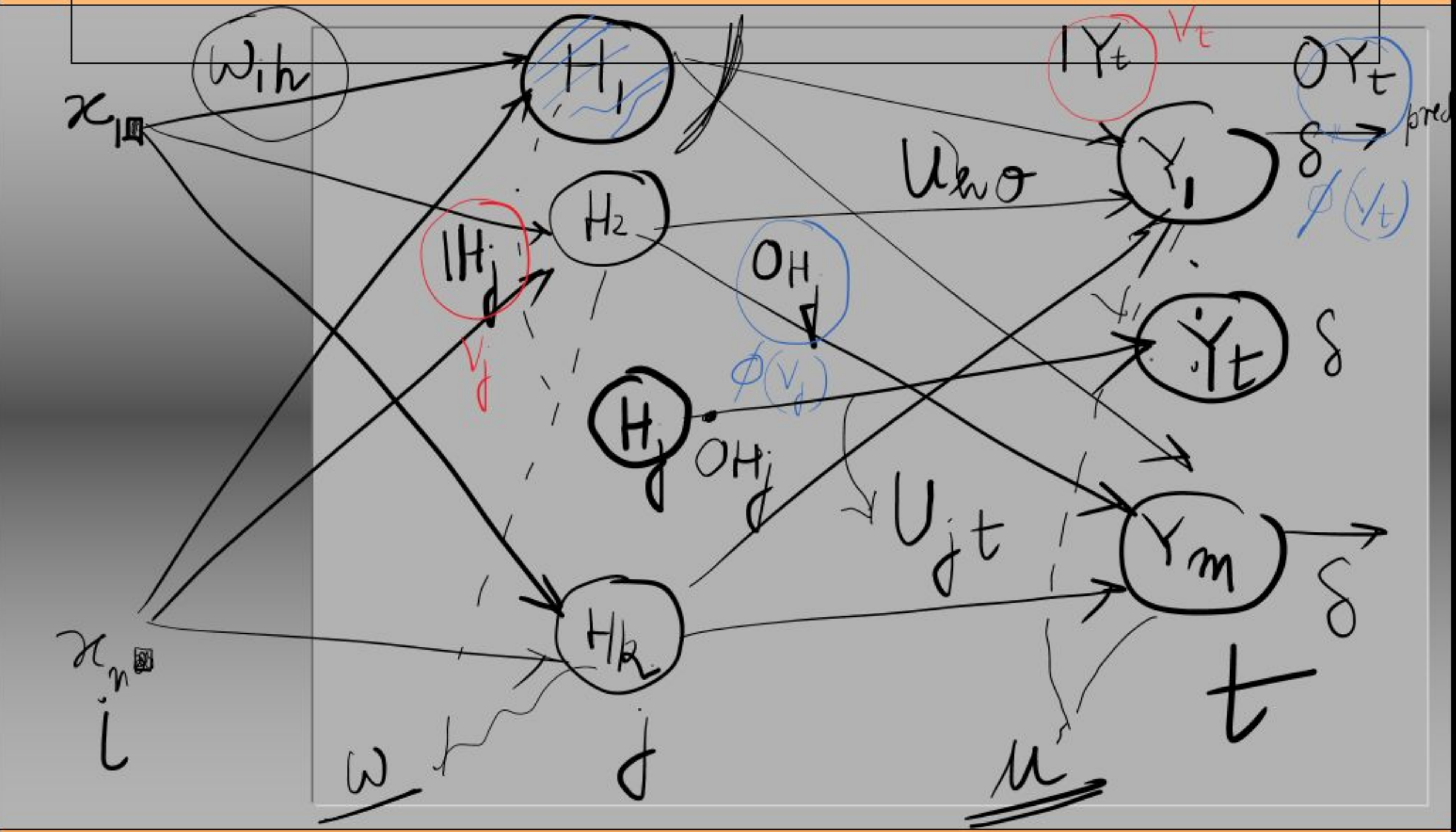
$$\frac{\partial y_j(n)}{\partial v_j(n)} = \phi_j'(v_j(n))$$

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

- Combining all the previous equations we get finally:

$$\Delta w_{ij}(n) = -\eta \frac{\partial \mathrm{E}(n)}{\partial w_{ji}(n)} = \eta \underbrace{\boxed{e_j(n)\phi_j'(v_j(n))}}_{\delta_j(n)} y_i(n)$$

FEEDFORWARD NEURAL NETWORK WITH HIDDEN LAYER

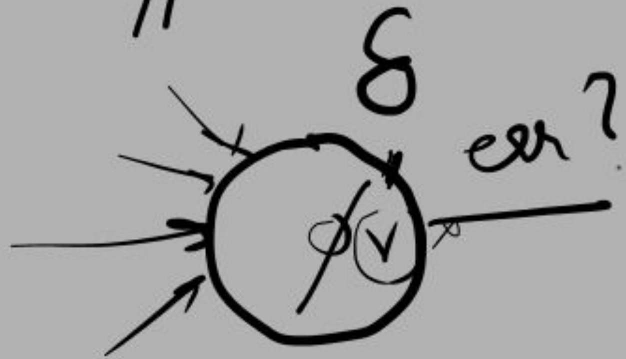


GENERALIZED DELTA RULE

1. $\Delta W = \eta \times \delta \times Y_{prev}$

2. $\delta = (Y - Y_{pred}) \times \phi'(V)$

3. $E = \frac{1}{2} \sum_{o/p} (Y - Y_{pred})^2$



HIDDEN TO OUTPUT

1. $\Delta U_{jt} = \eta \, \underline{\delta_{(t)}} \times \underline{OH_j}$

2. $\delta_{(t)} = (Y_t - Y_{t\,pred})\, \phi'(IY_t)$

3. $IY_t = \sum_{j=1}^{k} U_{jt} \times OH_j + u$

4. $U_{jt_{new}} = U_{jt_{old}} + \Delta U_{jt}$

INPUT TO HIDDEN

1 $\Delta W_{ij} = \eta \, \delta_j \times X_i$

2 $\delta_j = IS\delta_j \times \phi'(IH_j)$

3 $\boxed{IS\delta_j = \sum_{t=1}^{m} \delta(t) \times U_{jt}}$ BP

4 $W_{ij} = W_{ij} + \Delta W_{ij}$

5 $|IH_j| = \sum_{i} W_{ij} x_i + \omega$

SIGMOID DELTA

$$\delta = (Y_t - Y_{t\,pred})\,\phi'(IY_t)$$

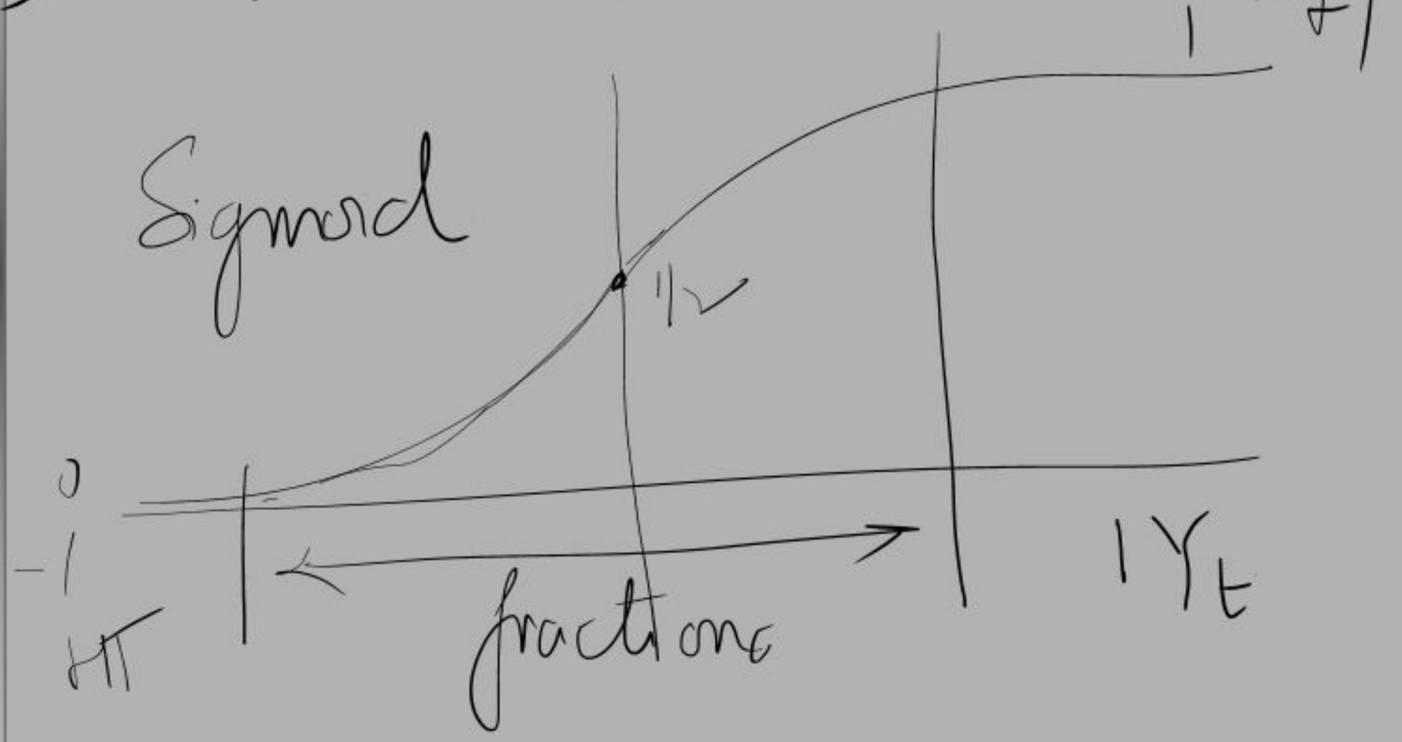
$$\phi'(IY_t) = \left(\frac{1}{1+e^{-IY_t}}\right)' = \frac{e^{-IY_t}}{(1+e^{-IY_t})^2}$$

$$= \left(\frac{1}{1+e^{-IY_t}}\right)\left(1 - \frac{1}{1+e^{-IY_t}}\right)$$

$$\Rightarrow (OY_t)(1 - OY_t)$$

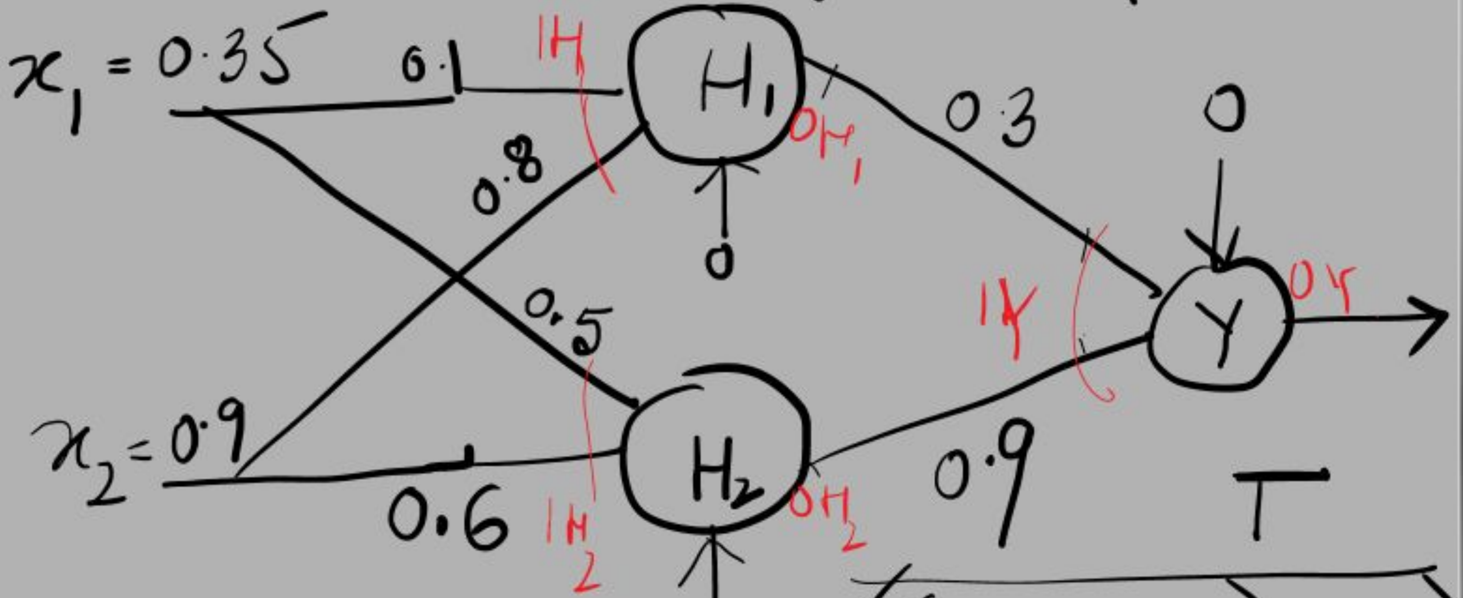
HYPERBOLIC TANGENT DELTA

$$\phi'(IY_t) = (1 - OY_t)(1 + OY_t)$$



ASSIGNMENT

Calc. ΔU, ΔW, for 1 epoch.



$$\left((0.35, .9), 0.5 \right)$$

$\eta = 1$

$$\frac{1}{1+e^{-y}}$$

$$IH_1 = 0.35 \times 0.1 + .9 \times .8 = (.752)$$

$$OH_1 = \frac{1}{1+e^{-.752}} =$$

$IH_2 =$

$OH_2 =$

$$IY = .3 \times OH_1 + .9 \times OH_2 =$$

$$OY = \frac{1}{1+e^{-\circ}}$$

$$\Delta U_1 = \eta \times \delta \times OH_1$$

$$\Delta U_2 = \eta \times \delta \times OH_2$$

$$\delta_0 = (O_t - OY_t)(1 - OY_t)(OY_t)$$

$$\delta_0 \times U_1 \, (1 - OH_j)(OH_j)$$

$$\delta_0 \times U_2 \, (1 - OH_j)(OH_j)$$

$$\Delta U_1 = \eta \times \delta \times OH_1$$

$$\Delta U_2 = \eta \times \delta \times OH_2$$

$$\underline{\delta_0}$$

$$= (O_t - OY_t)(1 - OY_t)(OY_t)$$

$$\delta_0 \times U_1 \; (1 - OH_j)(OH_j)$$

$$\delta_0 \times U_2 \; (1 - OH_j)(OH_j)$$

$$\Delta w_{11} = 1 \times \delta_1 \times X_1$$

$$\Delta w_{12} = \delta_2 \times X_1$$

$$\Delta w_{21} = \delta_1 \times X_2$$

$$\Delta w_{22} = \delta_2 \times X_2$$

Types of problems

Conclusions

- The BP algorithm is used in a great variety of problems:
  - Time series predictions
  - Credit risk assessment
  - Pattern recognition
  - Speech processing
  - Cognitive modelling
  - Image processing
  - Control
  - Etc
- BP is the **standard** algorithm against which all other NN algorithms are compared!!

- MLP and BP is used in Cognitive and Computational Neuroscience
- The algorithm can be used to make encoding / decoding and compression systems. Useful for data pre-processing operations
- The MLP with the BP algorithm is a universal approximator of functions

Advantages & Disadvantages II

- The algorithm is computationally efficient as it has O(W) complexity to the model parameters
- The algorithm has “local” **robustness**
- The convergence of the BP can be very slow, especially in large problems, depending on the method
- The BP algorithm suffers from the problem of local minima

Conclusions