# 18CSE751 – Introduction to Machine Learning

## Lecture 5: Revision of Unit I and Preamble to Unit II

## Lecture 6: Neurons & Neural Networks

Dr.Vani Vasudevan

Professor –CSE, NMIT

# Unit –II

**Neurons, Neural Networks:** The Brain and the Neuron, Neural Networks, The Perceptron, Training a Perceptron, Learning Boolean Functions, Linear Separability, **Multilayer Perceptron :** The Multi-layer Perceptron Algorithm, Initialising the Weights , Different Output Activation Functions, Backpropogation Algorithm, Sequential and Batch Training, local minima, picking up momentum, minibatches and stochastic gradient descent, other improvements

# 18CSE751 – Introduction to Machine Learning

## Lecture 6: Neurons & Neural Networks

Dr.Vani Vasudevan

Professor –CSE, NMIT

# OUTLINE

- THE BRAIN AND THE NEURON
- HEBB'S RULE
- MCCULLOCH AND PITTS MATEHMATICAL MODEL
- NEURAL NETWORKS : THE PERCEPTRON
- TRAINING A PERCEPTRON
- LEARNING BOOLEAN FUNCTIONS
- LINEAR SEPARABILITY

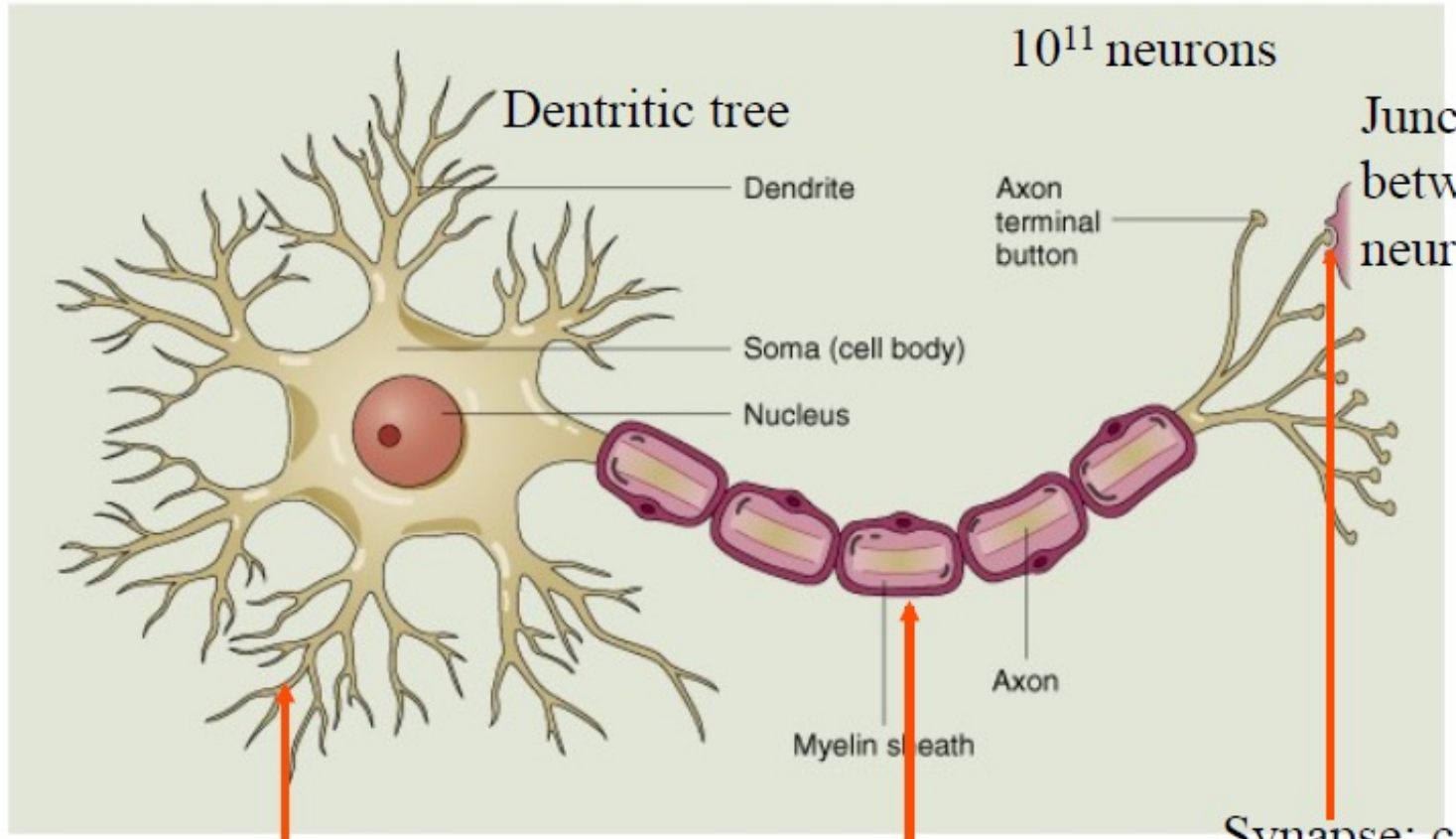
# BRAIN

Images.google.com

# THE BRAIN AND THE NEURON…

- **Nerve cells called neurons** are the processing units of the brain
- Transmitter chemicals within the fluid of the brain raise or lower the electrical potential inside the body of the neuron.
- If this **membrane potential** reaches some threshold, the neuron spikes or fires, and a pulse of fixed strength and duration is sent down the axon

# A Typical Cortical Neuron




Collect chemical signals

Axon: generate Potentials (Fire/not Fire)

Synapse: control release chemical transmitters.

# THE BRAIN AND THE NEURON…

- The axons divide (arborise) into connections to many other neurons**, connecting to each of these neurons in a synapse**
- Each neuron is typically connected to thousands of other neurons, it is estimated that there are about 100 trillion ($= 10^{14}$) synapses within the brain.
- After firing, the neuron must wait for some time to recover its energy (**the refractory period**) before it can fire again.

# THE BRAIN AND THE NEURON…

- Each **neuron can be viewed as a separate processor,** performing a very simple computation: **Deciding whether or not to fire.**
- This makes the brain a **massively parallel computer made up of $10^{11}$ processing elements.**

# THE BRAIN AND THE NEURON…

How does **learning** occur in the brain?

- The principal concept is **plasticity:**

  Modifying the strength of synaptic connections between neurons and creating new connections.

# HEBB'S RULE…

- **Changes in the strength of synaptic connections are proportional to the correlation in the firing of the two connecting neurons.**
  - If two neurons consistently fire simultaneously, then any connection between them will change in strength, becoming stronger.
  - If the two neurons never fire simultaneously, the connection between them will die away.
- **The idea : if two neurons both respond to something, then they should be connected.**

# HEBB'S RULE…

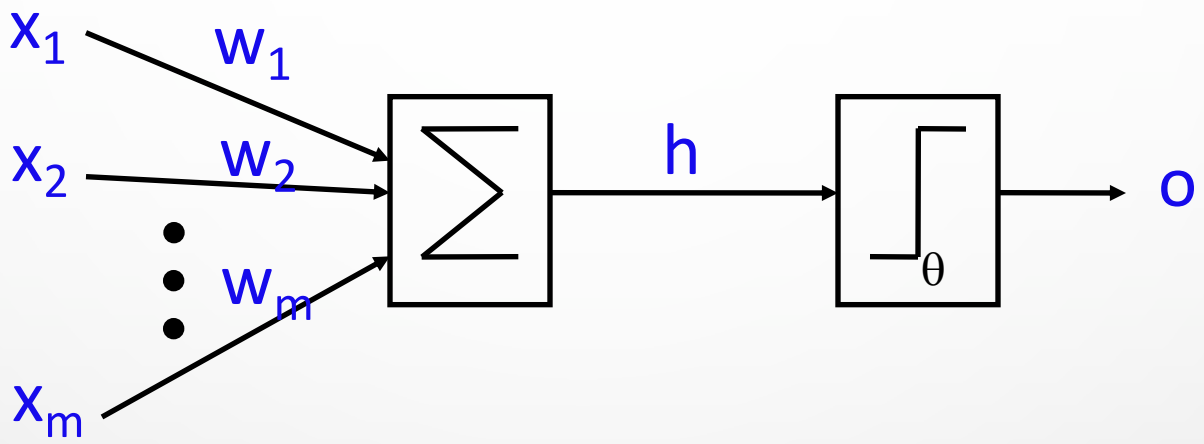
**Trivial Example**

- Suppose that you have a **neuron that recognises your grandmother** (this will probably get input from lots of visual processing neurons)
- If your grandmother always gives you a chocolate bar when she comes to visit, then some neurons, which are happy because you like the taste of chocolate, will also be stimulated. Since these neurons fire at the same time, they will be connected together, and the connection will get stronger over time.
- So eventually, the sight of your grandmother, even in a photo, will be enough to make you think of chocolate.
- This idea is called **classical conditioning.**

# HEBB'S RULE

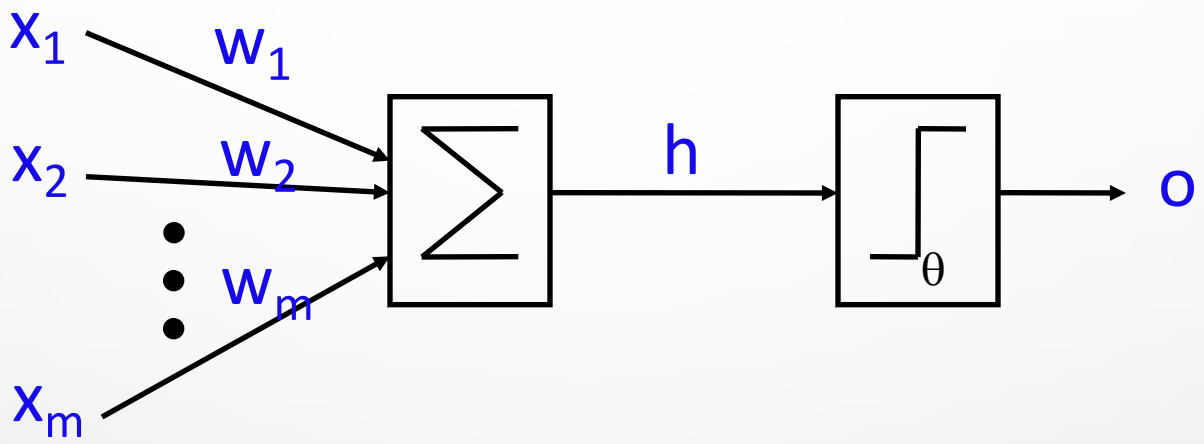
- **The synaptic connections between neurons and assemblies of neurons can be formed when they fire together and can become stronger.**
- It is also known as **long-term potentiation and neural plasticity**, and it does appear to have correlation in real brains.

# MCCULLOCH AND PITTS MATEHMATICAL MODEL OF A NEURON…




- Greatly simplified biological neurons
- The inputs $x_i$ are multiplied by the weights $w_i$, and the neurons sum their values.
- If the sum is greater than the threshold $\theta$ then the neuron fires; otherwise, it does not.

# MCCULLOCH AND PITTS MATEHMATICAL MODEL OF A NEURON…




(1) a set of weighted inputs $w_i$, that correspond to the synapses

(2) an adder that sums the input signals (equivalent to the membrane of the cell that collects electrical charge)

(3) an activation function (initially a threshold function) that decides whether the neuron fires ('spikes') for the current inputs

# MCCULLOCH AND PITTS NEURONS

$$h = \sum_{i=1}^{m} x_i w_i \qquad o = \begin{cases} 1 & h \geq \theta \\ 0 & h < \theta \end{cases}$$

for some threshold $\theta$

- THE WEIGHT $w_i$ CAN BE POSITIVE OR NEGATIVE
  - INHIBITORY OR EXITATORY
- USE ONLY A LINEAR SUM OF INPUTS
- USE A SIMPLE OUTPUT INSTEAD OF A PULSE (SPIKE TRAIN)

# LIMITATIONS OF MCCULLOCH AND PITTS NEURONS

- **Inhibitory or excitatory types of synapses do exist within the brain.**
- **With the McCulloch and Pitts neurons, the weights can change from positive to negative or vice versa**
- Biologically—
  - **synaptic connections are either excitatory or inhibitor**y, and never change from one to the other.
  - **real neurons can have synapses that link back to themselves** in a feedback loop, but we do not usually allow that possibility when we make networks of neurons

# NEURAL **NETWORKS**

- Can put lots of McCulloch & Pitts neurons together
- Connect them up in any way we like
- In fact, assemblies of the neurons are capable of ***Universal Computation***
  - Can perform any computation that a normal computer can
  - Just have to solve for all the weights $w_{ij}$

# TRAINING NEURONS

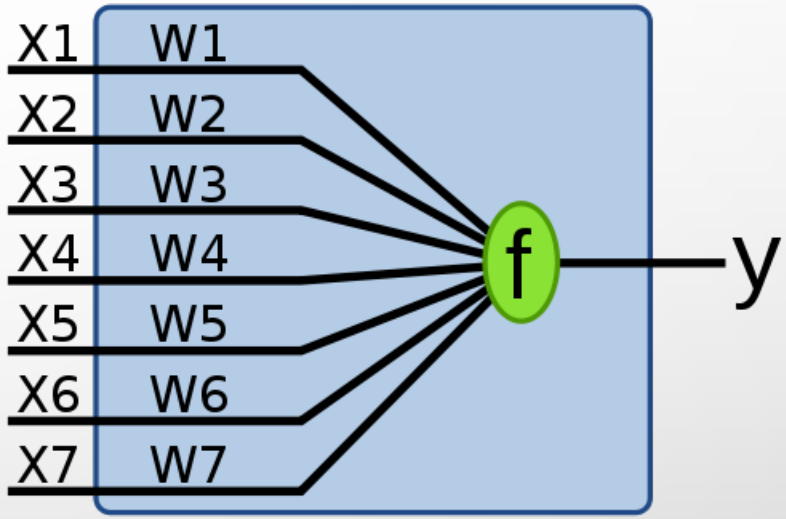
- Adapting the weights is learning
  - How does the network know it is right?
  - How do we adapt the weights to make the network right more often?
- Training set with target outputs
- Learning rule

# THE PERCEPTRON

The perceptron is considered the **simplest kind of feed-forward neural network**.

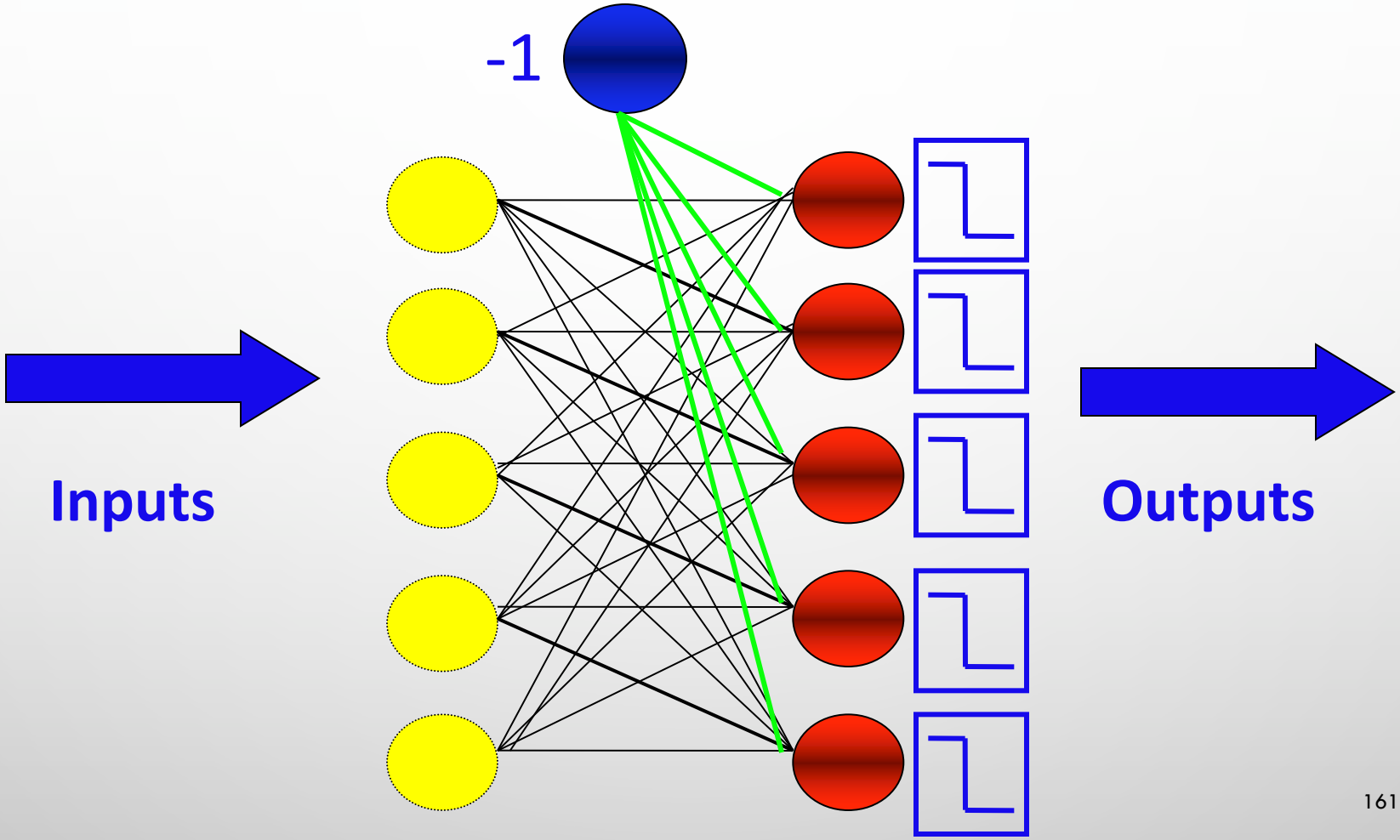
The **perceptron** is a binary classifier which maps its input *x* (a real-valued vector) to an output value *f*(*x*) (a single binary value) across the matrix:

$$f(x) = \begin{cases} 1 & wx - b > 0 \\ 0 & else \end{cases}$$




In order to not explicitly write b, we extend the input vector x by one more dimension that is always set to -1, e.g., **x=(-1,x_1, …, x_7)** with **x_0=-1**, and extend the weight vector to **w=(w_0,w_1, …, w_7)**. Then adjusting w_0 corresponds to adjusting b.

# BIAS REPLACES THRESHOLD



**The Perceptron Algorithm**

- **Initialisation**
  - set all of the weights $w_{ij}$ to small (positive and negative) random numbers
- **Training**
  - for $T$ iterations or until all the outputs are correct:
    - for each input vector:
      - compute the activation of each neuron $j$ using activation function $g$:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } \sum_{i=0}^{m} w_{ij}x_i > 0 \\ 0 & \text{if } \sum_{i=0}^{m} w_{ij}x_i \leq 0 \end{cases} \tag{3.4}$$

      - update each of the weights individually using:

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i \tag{3.5}$$

- **Recall**
  - compute the activation of each neuron $j$ using:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } w_{ij}x_i > 0 \\ 0 & \text{if } w_{ij}x_i \leq 0 \end{cases} \tag{3.6}$$

# PERCEPTRON DECISION = RECALL

- OUTPUTS ARE:

$$y_j = \text{sign}\left(\sum_{i=1}^{n} w_{ij} x_i\right)$$

$$\Rightarrow \mathbf{w} \cdot \mathbf{x} > 0$$

For example, y=(y_1, …, y_5)=(1, 0, 0, 1, 1) is a possible output.
We may have a different function **g** in the place of sign.

# PERCEPTRON LEARNING = UPDATING THE WEIGHTS

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

- WE WANT TO CHANGE THE VALUES OF THE WEIGHTS
- AIM: MINIMISE THE *ERROR* AT THE OUTPUT
- IF $E = T\text{-}Y$, WANT $E$ TO BE 0
- USE:

Learning rate

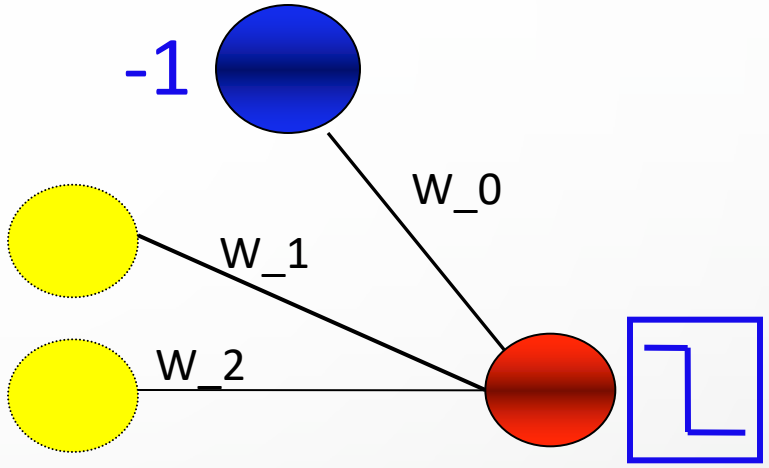
Input

$$\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$$

Error

# EXAMPLE 1: THE LOGICAL OR

| X_1 | X_2 | t |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

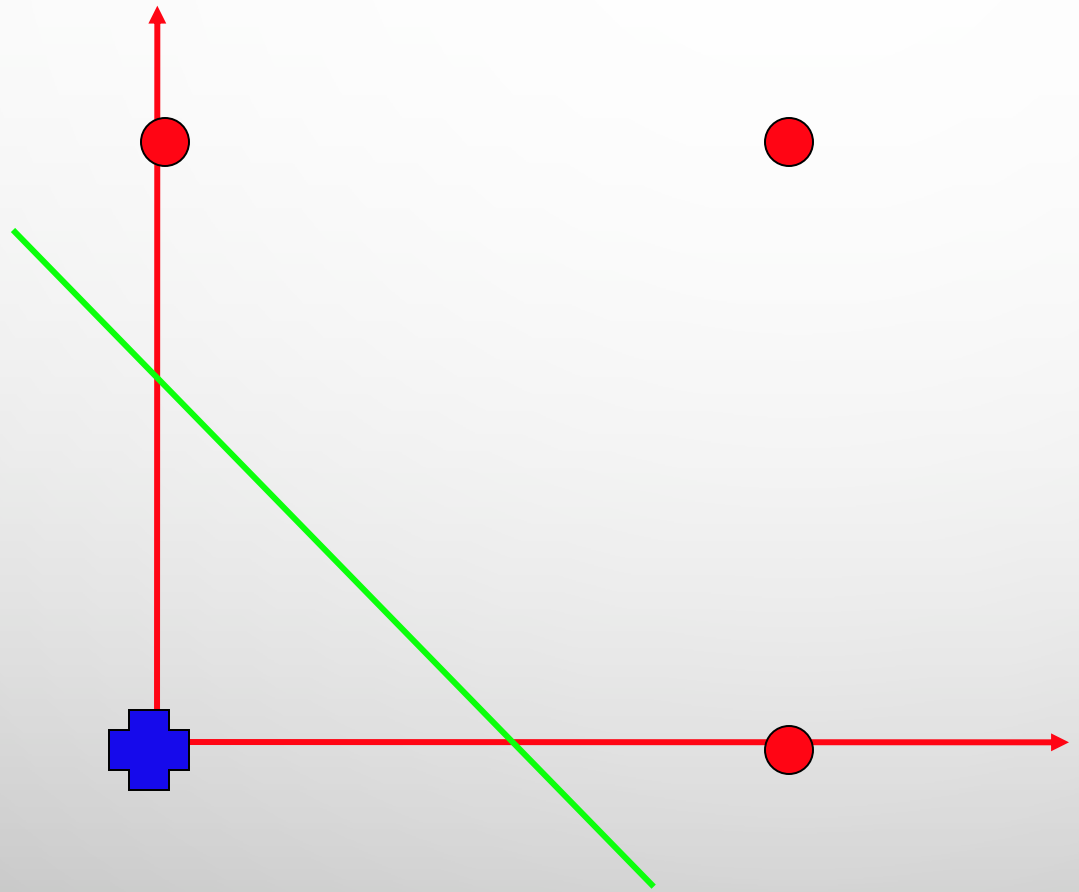



Initial values: $w_0(0)=-0.05$, $w_1(0)=-0.02$, $w_2(0)=0.02$, and $\eta=0.25$
Take first row of our training table:
$y_1= \text{sign}( -0.05\times-1 + -0.02\times0 + 0.02\times0 ) = 1$

$w_0(1) = -0.05 + 0.25\times(0-1)\times-1=0.2$
$w_1(1) = -0.02 + 0.25\times(0-1)\times0=-0.02$
$w_2(1) = 0.02 + 0.25\times(0-1)\times0=0.02$

We continue with the new weights and the second row, and so on
We make several passes over the training data.

# DECISION BOUNDARY FOR OR PERCEPTRON



# LINEAR SEPARABILITY

- OUTPUTS ARE:

$$y_j = \text{sign}\left(\sum_{i=1}^{n} w_{ij} x_i\right)$$

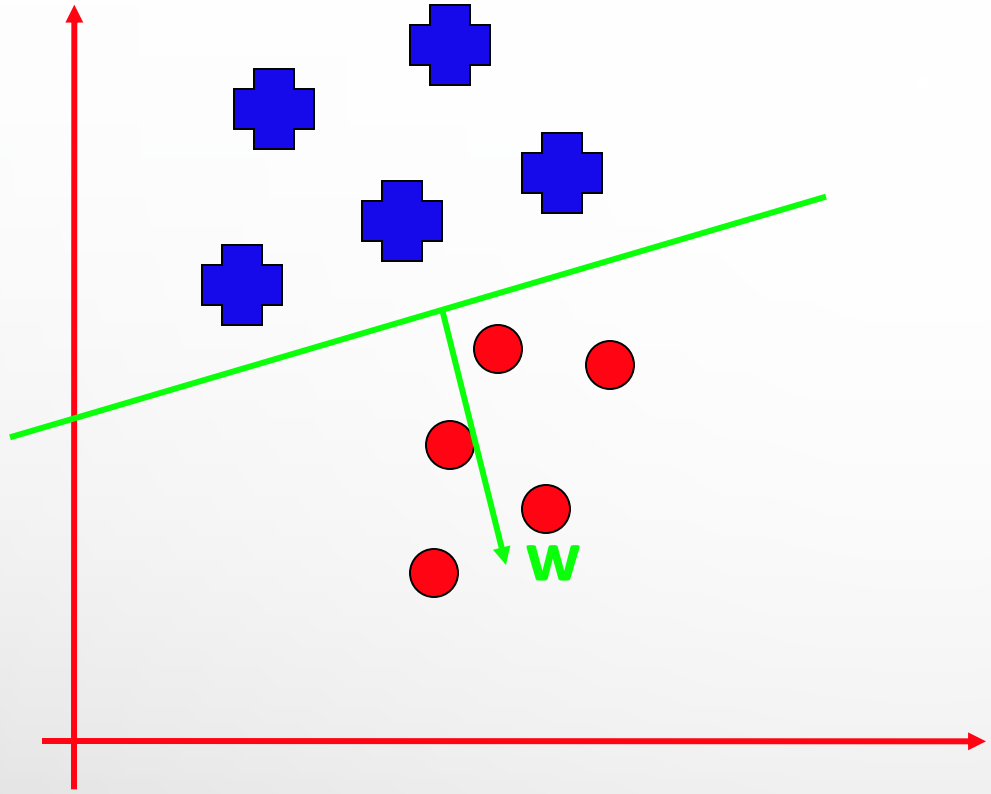
$$\Rightarrow \mathbf{w} \cdot \mathbf{x} > 0$$

where

$$\mathbf{w} \cdot \mathbf{x} = \| \mathbf{w} \| \times \| \mathbf{x} \| \cos\alpha$$

and $\alpha$ is the angle between vectors **x** and **w**.

# GEOMETRY OF LINEAR SEPARABILITY




The equation of a line is
$w_0 + w_1*x + w_2*y=0$
It also means that point (x,y) is on the line

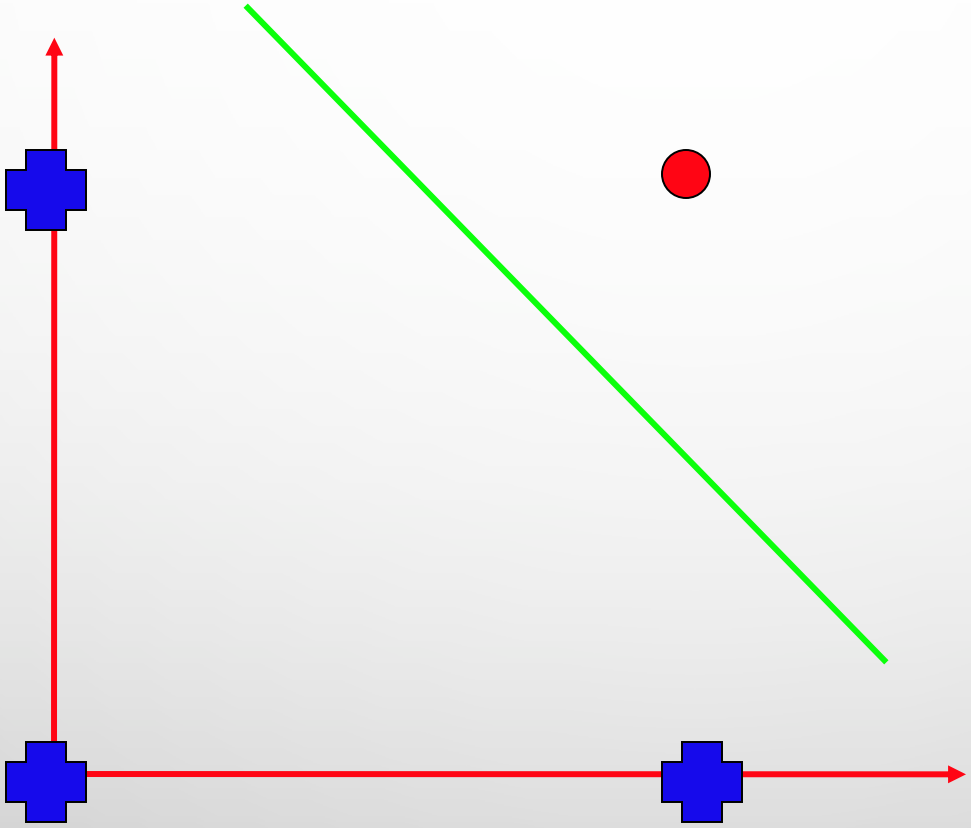
This equation is equivalent to

$\mathbf{w}\cdot\mathbf{x} = (w_0, w_1, w_2) \cdot (1,x,y) = 0$

If $\mathbf{w}\cdot\mathbf{x} > 0$, then the angle
between **w** and **x**
is less than 90 degree, which means that
**w** and **x** lie on the same side of the line.

Each output node of perceptron tries to separate the training data
Into two classes (fire or no-fire) with a linear decision boundary,
i.e., straight line in 2D, plane in 3D, and hyperplane in higher dim.

# LINEAR SEPARABILITY



The Binary
AND
Function

# PERCEPTRON LEARNING MOVIE

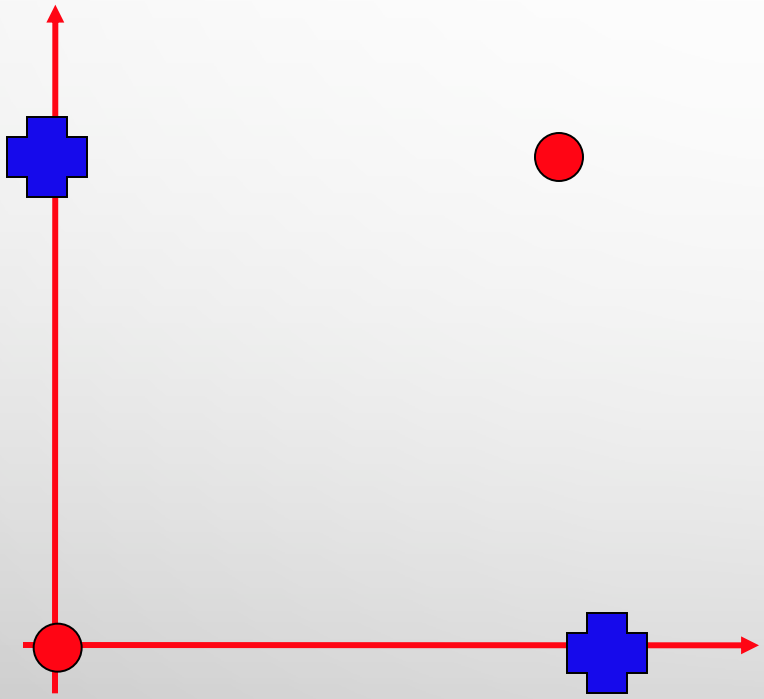
- HTTPS://WWW.YOUTUBE.COM/WATCH?V=VGWEMZHPLSA

# Perceptron Convergence Theorem

- For any data set which is linearly separable, the algorithm is guaranteed to find a solution in a finite number of steps (Rosenblatt, 1962; Block 1962; Nilsson, 1965; Minsky and Papert 1969; Duda and Hart, 1973; Hand, 1981; Arbib, 1987; Hertz et al., 1991)

# Limitations of the Perceptron…

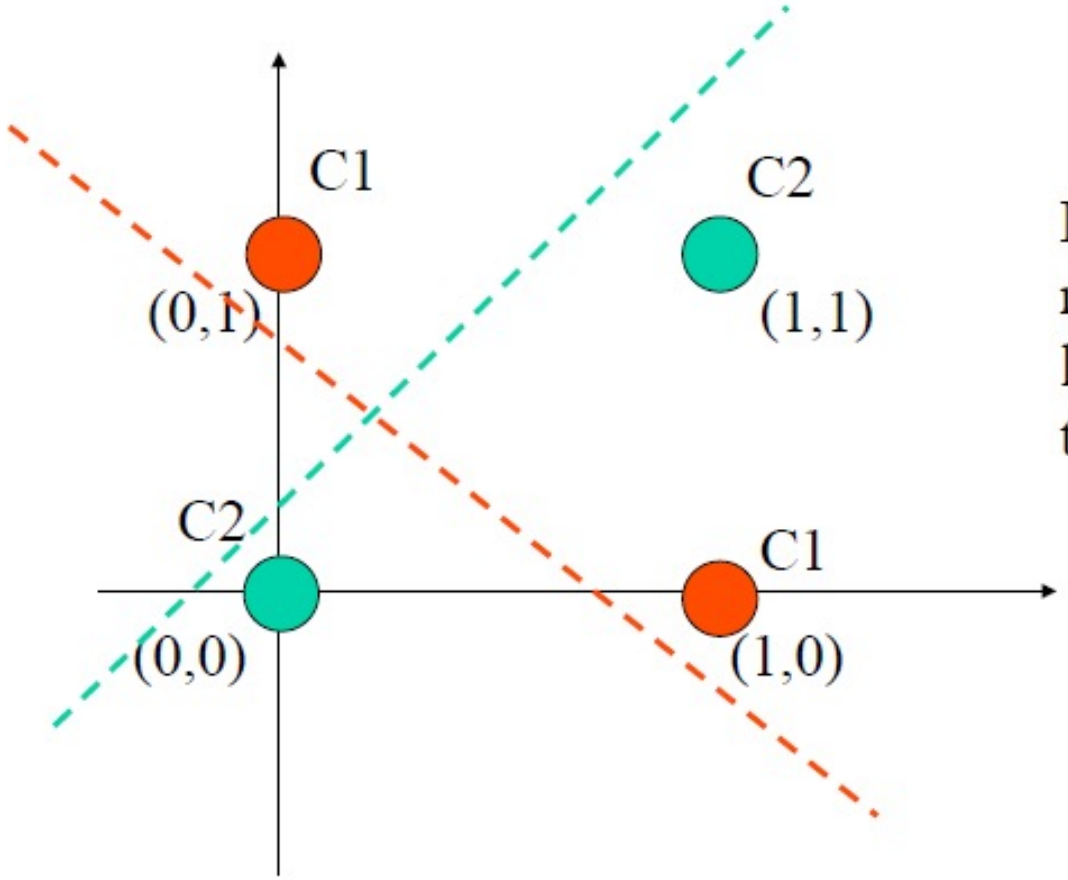
## Linear Separability



## The Exclusive Or (XOR) function.

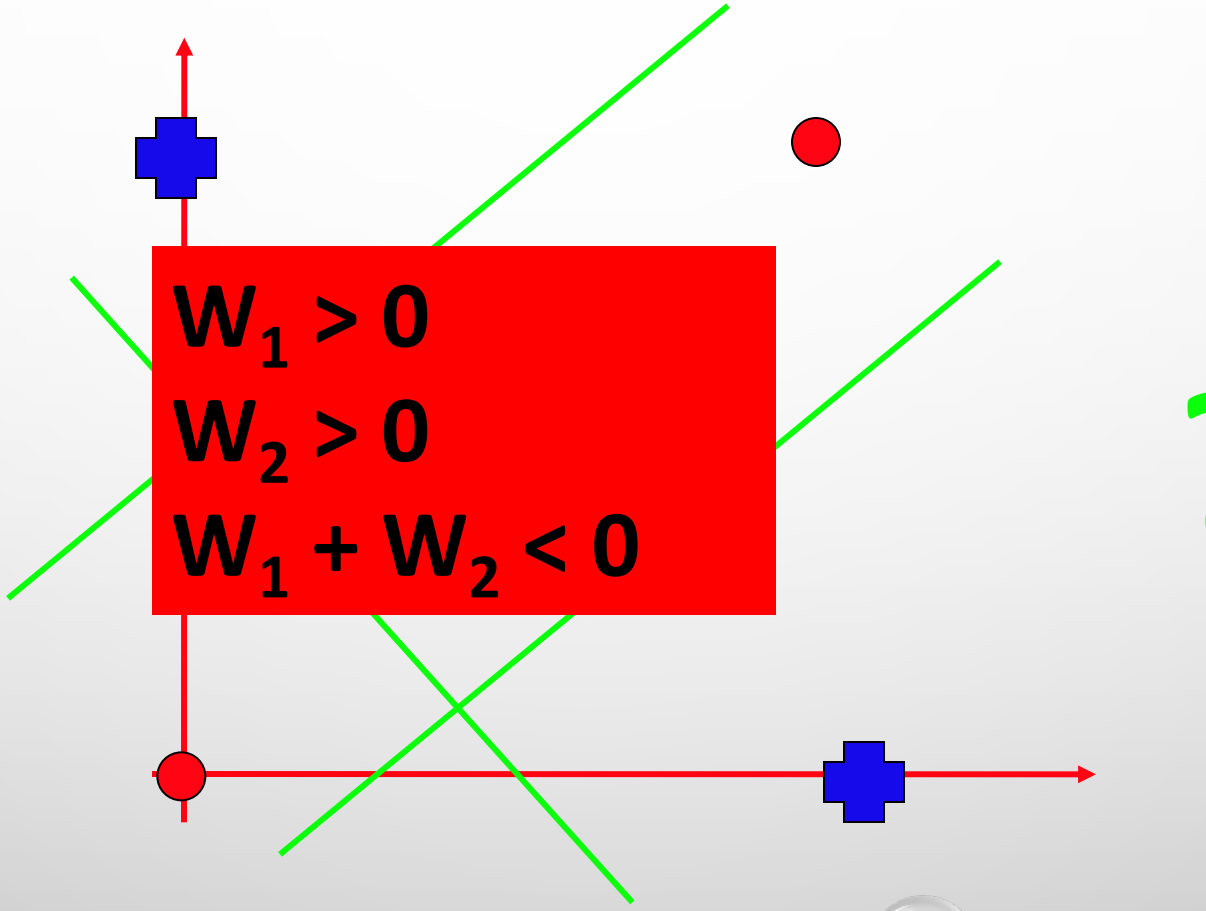
| A | B | Out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Exclusive OR Problem



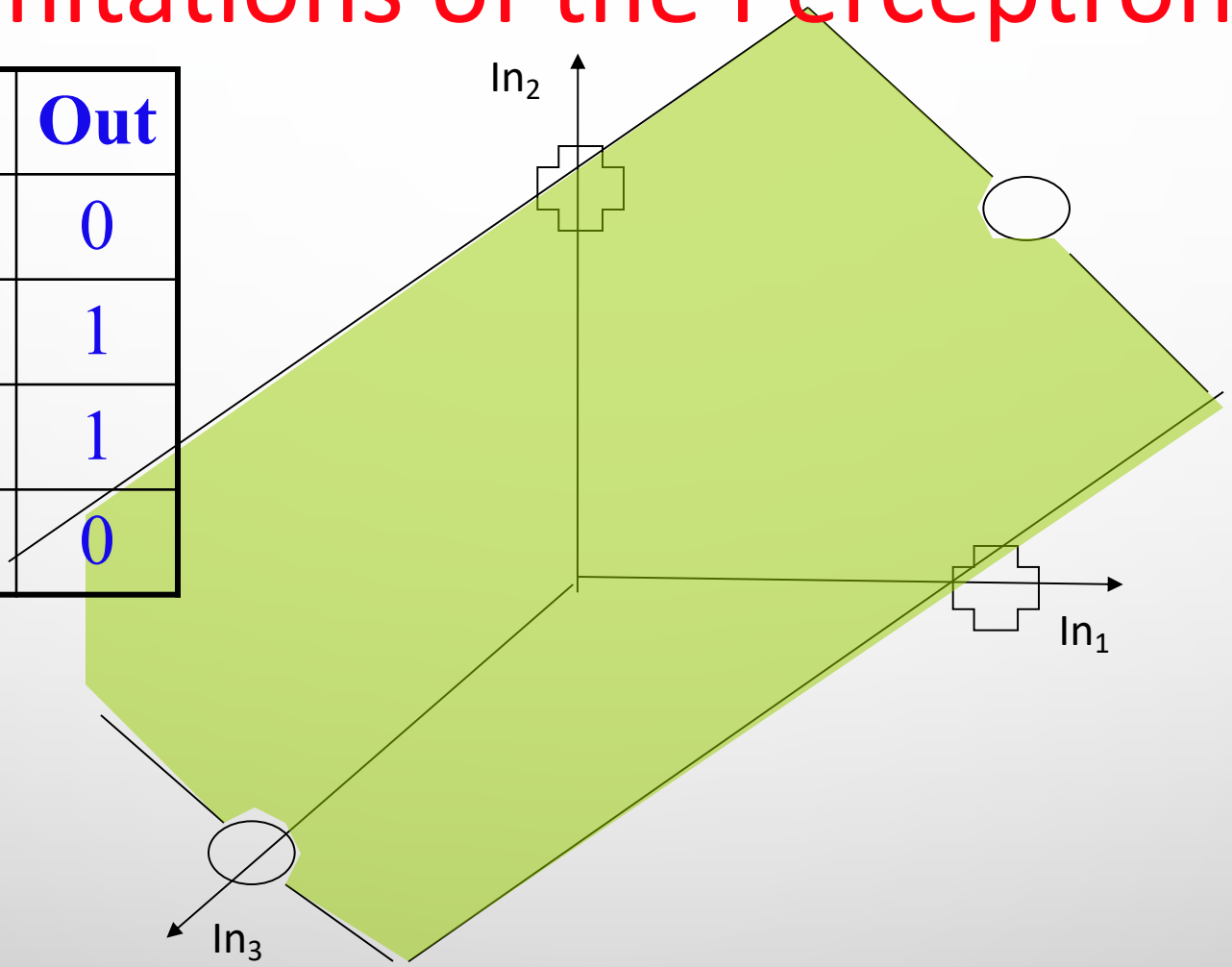

Perceptron (or one-layer neural network) can not learn a function to separate the two classes perfectly.

# Limitations of the Perceptron…



# Limitations of the Perceptron?

| A | B | C | Out |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |



# 18CSE751 – Introduction to Machine Learning

## Lecture 7: Neural Networks Overview

Dr.Vani Vasudevan

Professor –CSE, NMIT

# 18CSE751 – Introduction to Machine Learning

## Lecture 8: Neural Networks & Backpropagation in a simple way

Dr.Vani Vasudevan

Professor –CSE, NMIT

# RECAP

- THE BRAIN AND THE NEURON
- HEBB'S RULE
- MCCULLOCH AND PITTS MATEHMATICAL MODEL
- NEURAL NETWORKS : THE PERCEPTRON
- TRAINING A PERCEPTRON
- LEARNING BOOLEAN FUNCTIONS
- LINEAR SEPARABILITY

# WORTHY READS

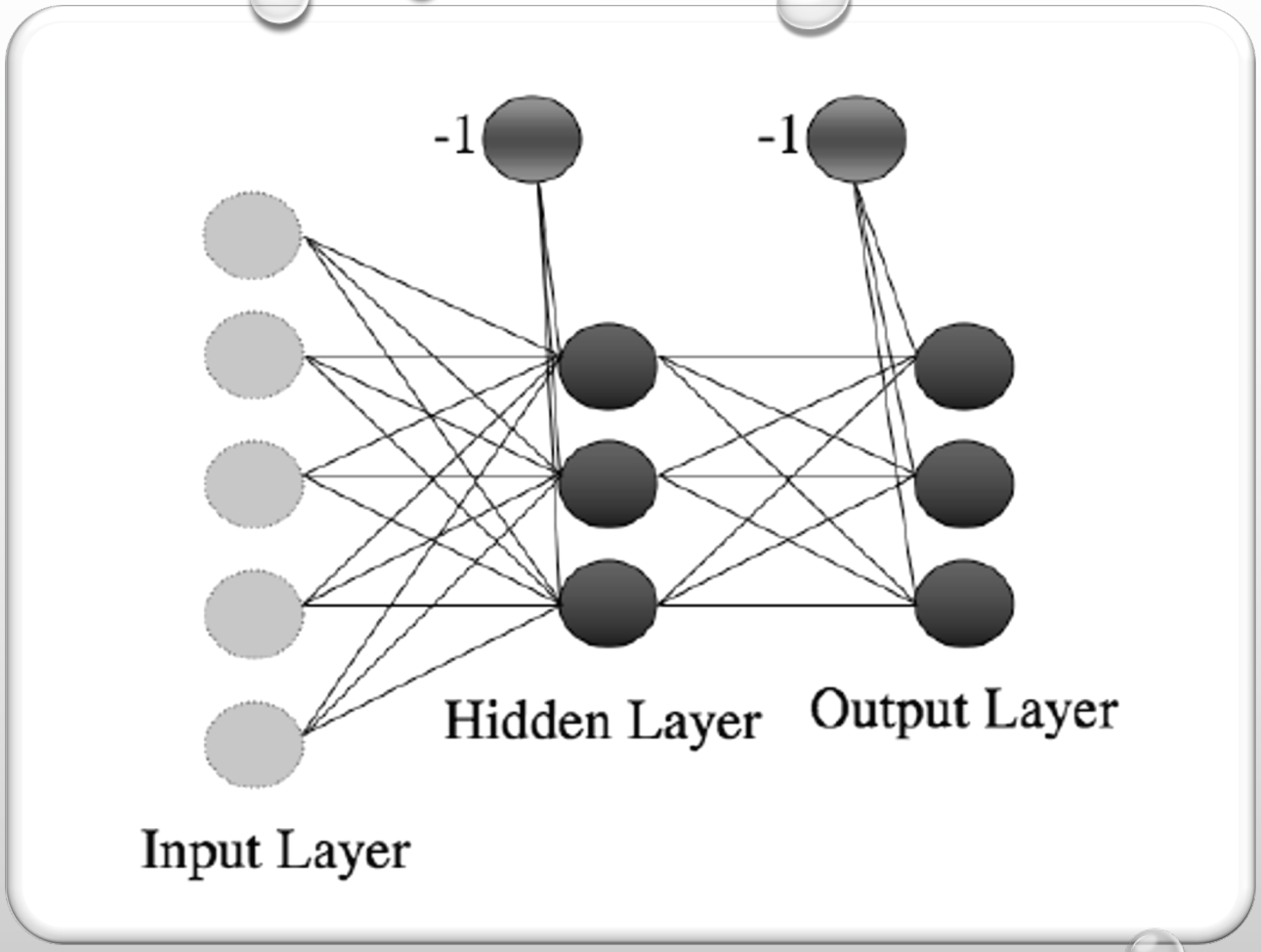
- HTTPS://MEDIUM.COM/DATATHINGS/NEURAL-NETWORKS-AND-BACKPROPAGATION-EXPLAINED-IN-A-SIMPLE-WAY-F540A3611F5E
- HTTP://RUDER.IO/OPTIMIZING-GRADIENT-DESCENT/
- HTTPS://MEDIUM.COM/THE-THEORY-OF-EVERYTHING/UNDERSTANDING-ACTIVATION-FUNCTIONS-IN-NEURAL-NETWORKS-9491262884E0
- HTTPS://WWW.ANALYTICSVIDHYA.COM/BLOG/2021/04/ACTIVATION-FUNCTIONS-AND-THEIR-DERIVATIVES-A-QUICK-COMPLETE-GUIDE/

# 18CSE751 – Introduction to Machine Learning

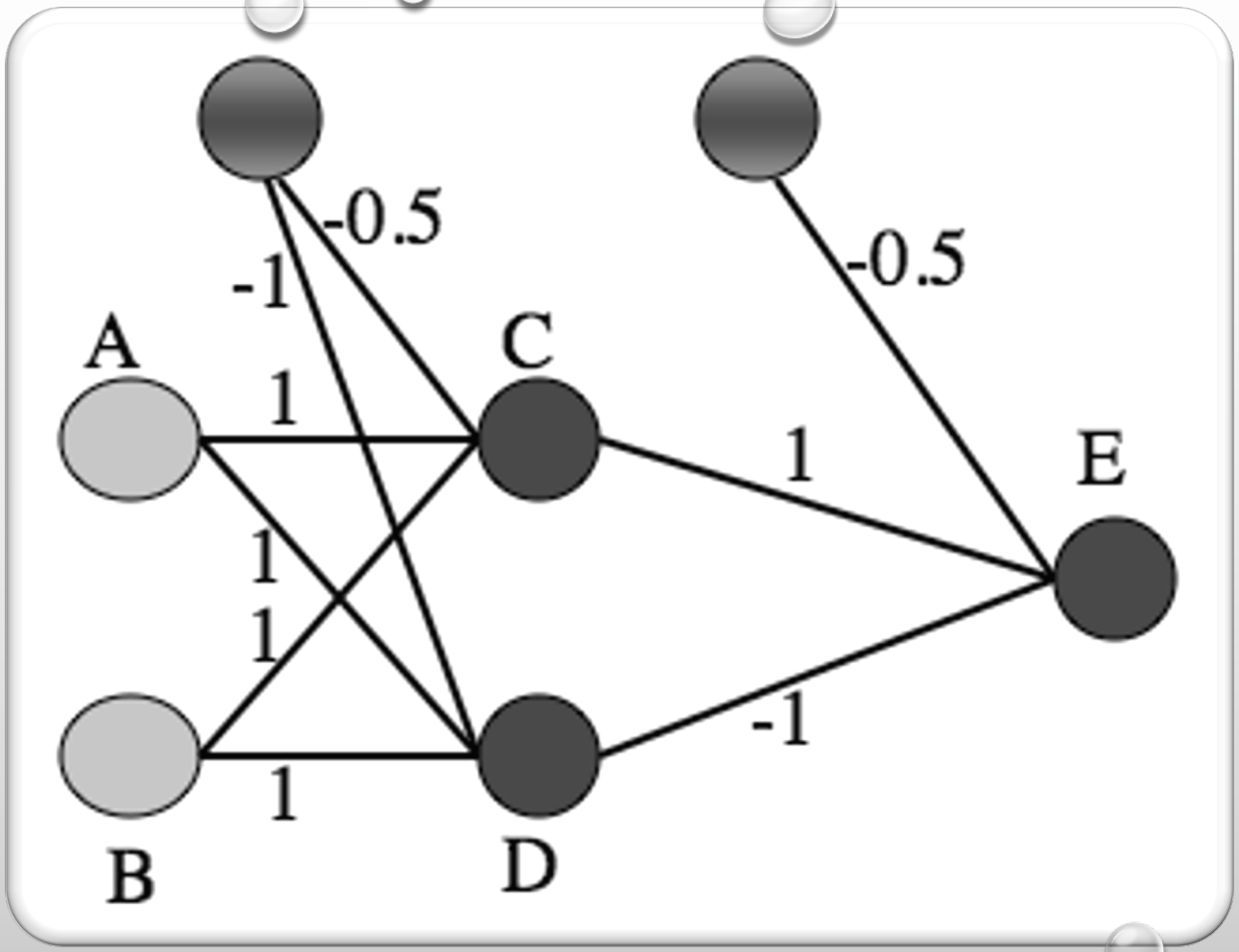
## Lecture 9: Single Layer and Multi Layer Perceptron

Dr.Vani Vasudevan

Professor –CSE, NMIT




# THE MULTI-LAYER PERCEPTRON NETWORK, CONSISTING OF MULTIPLE LAYERS OF CONNECTED NEURONS.



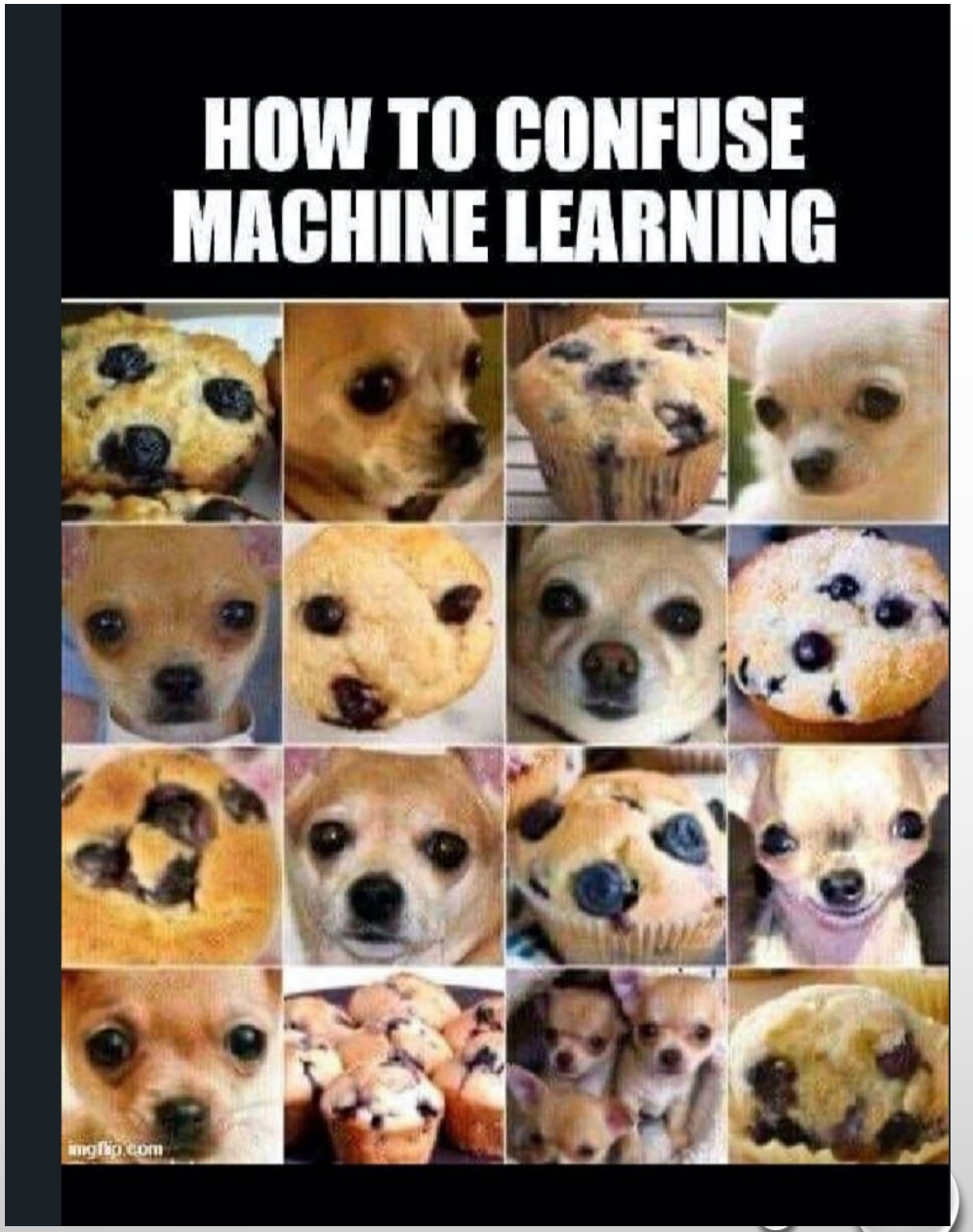

# A MULTI-LAYER PERCEPTRON NETWORK SHOWING A SET OF WEIGHTS THAT SOLVE THE XOR PROBLEM.

# 18CSE751 – Introduction to Machine Learning

## Lecture 10: Multi-Layer Perceptron, Activation Functions and Variants in Gradient Descent

Dr.Vani Vasudevan

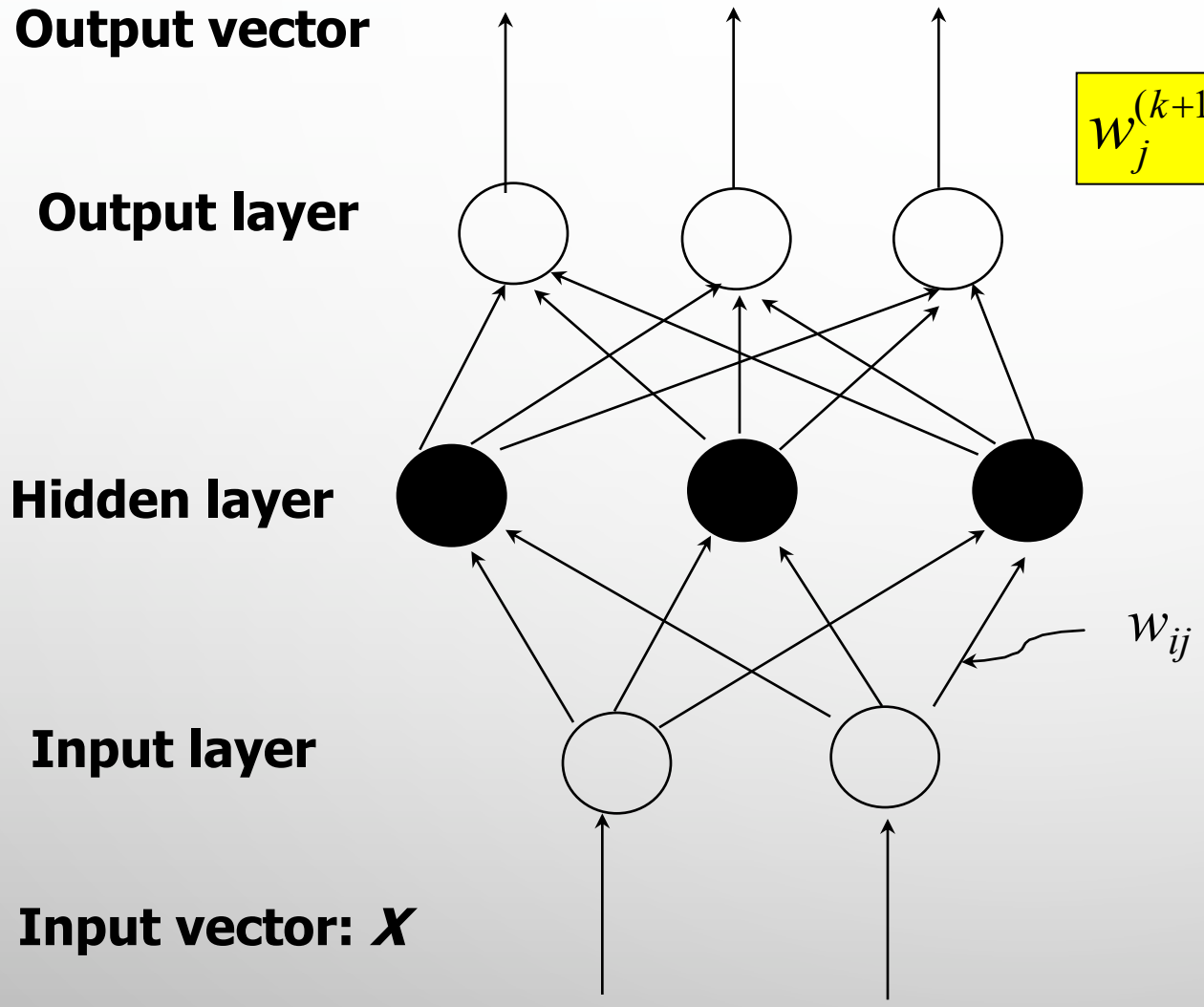
Professor –CSE, NMIT


HOW TO CONFUSE
MACHINE LEARNING
imgflip.com

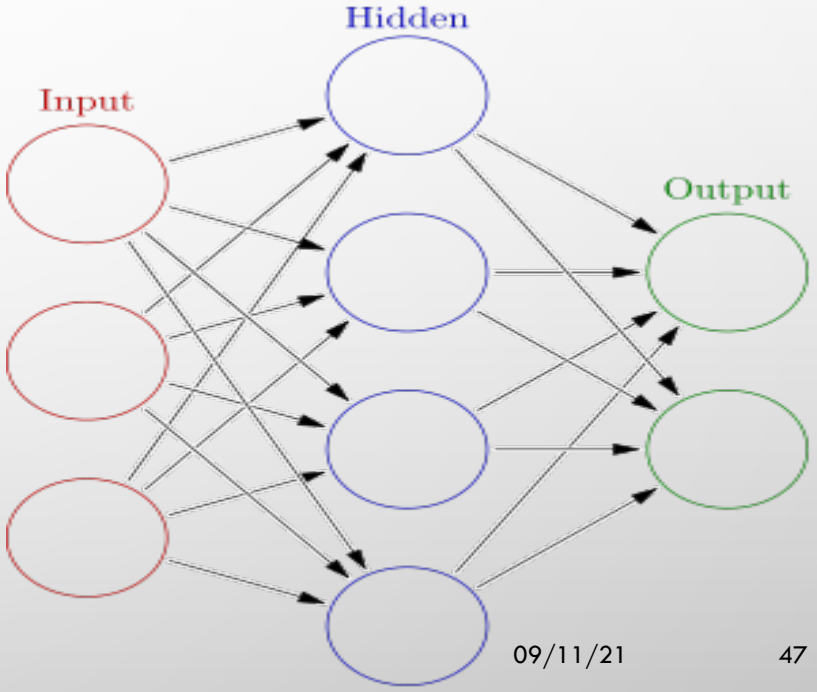
# OUTLINE

- MULTI-LAYER FFNN
- ACTIVATION/TRANSFER FUNCTION
- FORWARD PROPAGATION
- ADJUSTING WEIGHTS USING GRADIENT DESCENT (BACK PROPAGATION)
- INCREMENTAL STOCHASTIC GRADIENT DESCENT (SGD)

# A MULTI-LAYER FEED-FORWARD NEURAL NETWORK

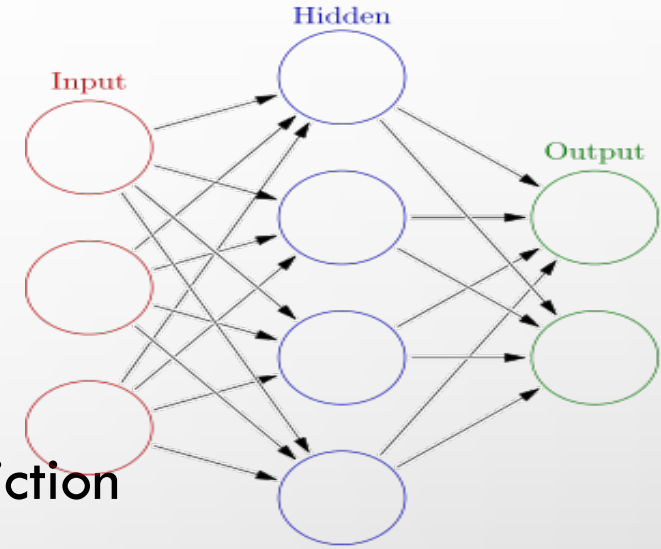



$$w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i - \hat{y}_i^{(k)})x_{ij}$$



# HOW A MULTI-LAYER NEURAL NETWORK WORKS

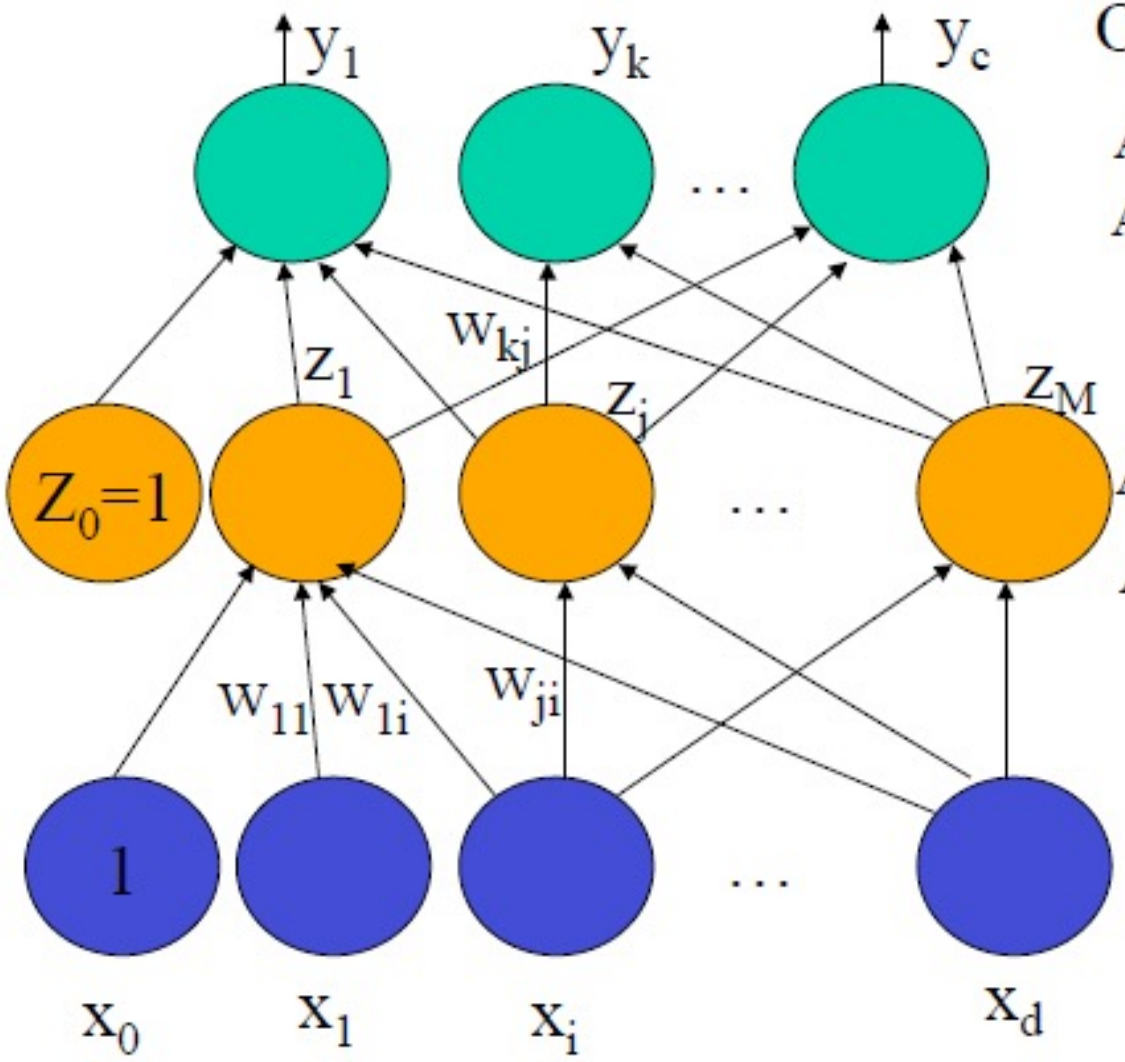
- The **inputs** to the network correspond to the attributes measured for each training tuple
- Inputs are fed simultaneously into the units making up the **input layer**
- They are then weighted and fed simultaneously to a **hidden layer**
- The number of hidden layers is arbitrary, although usually only one
- The weighted outputs of the last hidden layer are




input to units making up the **output layer**, which emits the network's prediction

- The network is **feed-forward**: none of the weights cycles Back to an input unit or to an output unit of a previous layer
- From a statistical point of view, networks perform **nonlinear regression**
  - Given enough hidden units and enough training samples, they can closely approximate ANY FUNCTION

# Two-Layer Neural Network




Output

Activation function: $f$ (linear,sigmoid, softmax)

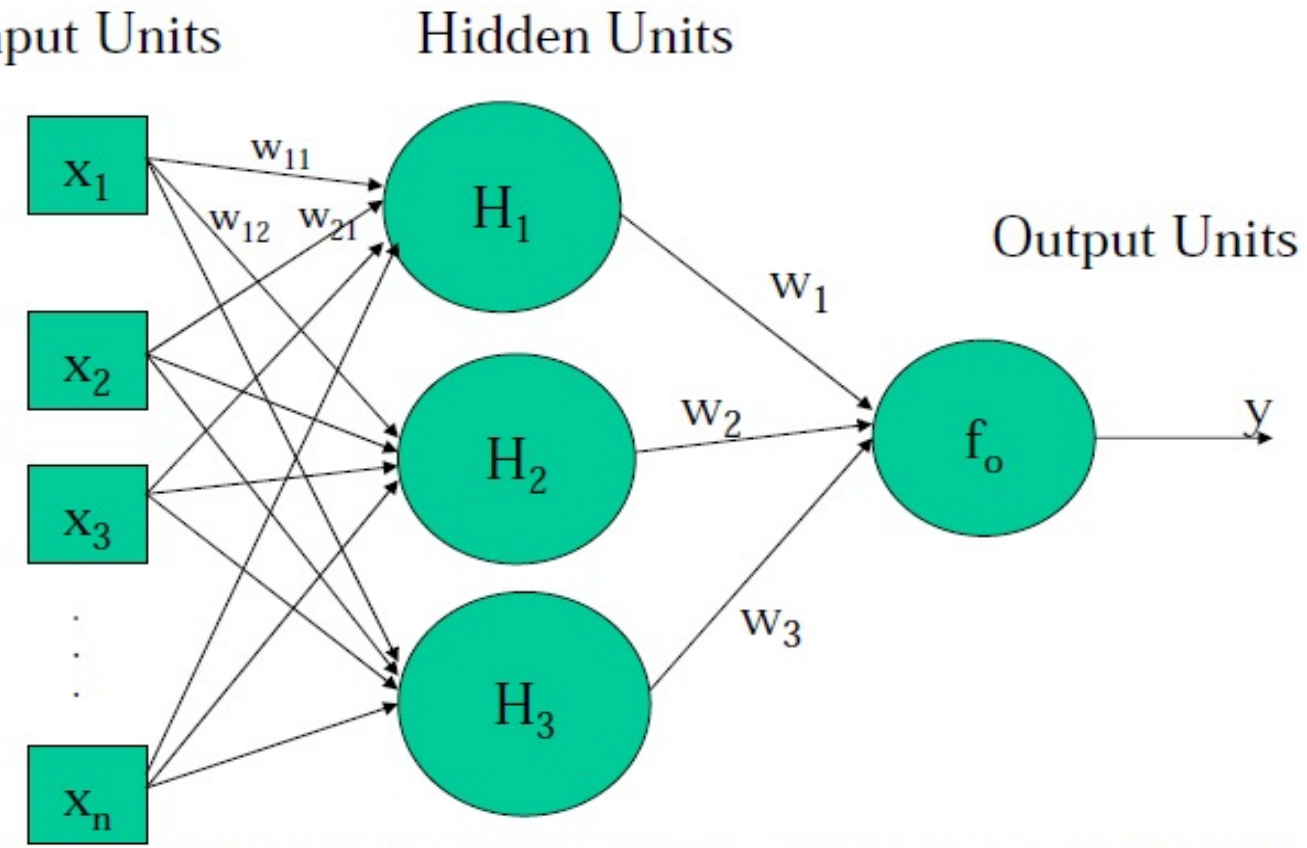
Activation of unit $a_k$: $\sum_{j=0}^{M} w_{kj} z_j$

Activation function: $g$ (linear, tanh, sigmoid)

Activation of unit $a_j$: $\sum_{i=0}^{d} w_{ji} x_i$

$$y_k = f\left(\sum_{j=0}^{M} w_{kj} \times g\left(\sum_{i=0}^{d} w_{ji} x_i\right)\right)$$

# A General Neural Network




Each weighted connection means the product of the output of one unit and the weight is sent to another unit as input. Each hidden unit and output unit have a transfer function to convert the sum of inputs into an output. Let transfer function of hidden unit be $f_h$ (e.g., identity function) output unit to be $f_o$ ( e.g., sigmoid function, $1/(1+e^{-x})$).

# ACTIVATION / TRANSFER FUNCTION

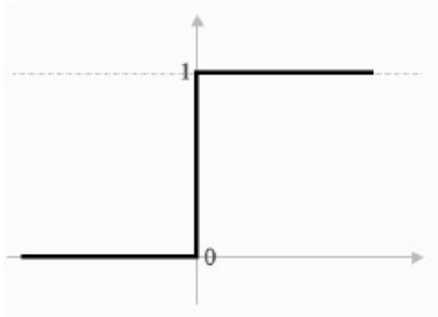



FIGURE 4.4 The threshold function that we used for the Perceptron. Note the discontinuity where the value changes from 0 to 1.

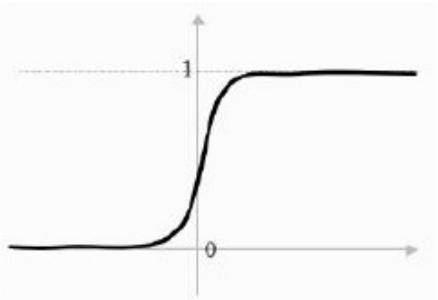
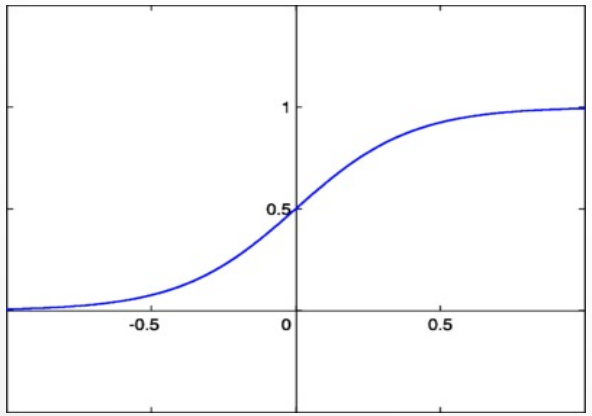



FIGURE 4.5 The sigmoid function, which looks qualitatively fairly similar, but varies smoothly and differentiably.

# ACTIVATION / TRANSFER FUNCTION

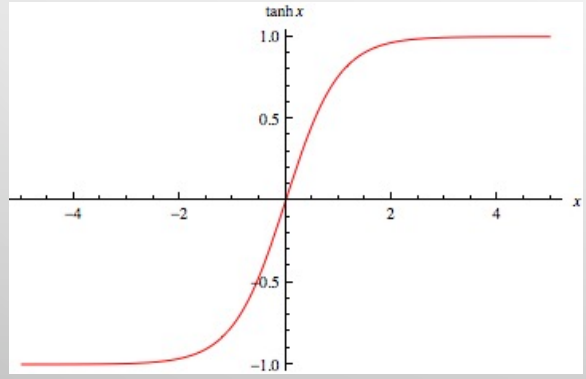
- SIGMOID FUNCTION:




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

**What is derivative of f(x)?**

- TANH FUNCTION:




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

# ACTIVATION FUNCTIONS & ITS DERIVATIVES

| Function Type | Equation | Derivative |
|---|---|---|
| **Linear** | $f(x) = ax + c$ | $f'(x) = a$ |
| **Sigmoid** | $f(x) = \frac{1}{1+e^{-x}}$ | $f'(x) = f(x)\,(1 - f(x))$ |
| **TanH** | $f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$ | $f'(x) = 1 - f(x)^2$ |
| **ReLU** | $f(x) = \begin{cases} 0 \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} 0 \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$ |
| **Parametric ReLU** | $f(x) = \begin{cases} \alpha x \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$ |
| **ELU** | $f(x) = \begin{cases} \alpha(e^x - 1) \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} f(x) + \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$ |

# Neural Network is a Universal Function Approximator

We can represent neural network as an function:

$$y = f_o\left(\sum_i W_i f_h\left(\sum_j x_j W_{ij}\right)\right)$$

This function is universal, which means that any function y=*f*(x) can be approximated by this function accurately, given a set of appropriate weights W.

So, the key is to adjust weights W to make neural network to approximate the function of our interest. e.g., given input of sequence features, tell if it is a gene or not (1: yes, 0: no)?

# Adjust Weights by Training

- How to adjust weights?
- Adjust weights using known examples (training data) $(x_1,x_2,x_3,\ldots x_n,y)$. This process is called training or learning
- Try to adjust weights so that the difference between the output of the neural network and y (called target) becomes smaller and smaller.
- Goal is to minimize Error (difference)

# Adjust Weights using Gradient Descent (back-propagation)

Known:

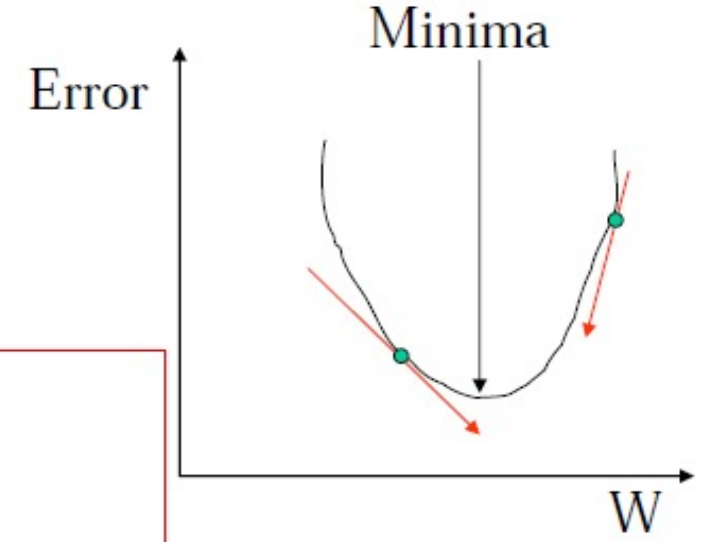
Data: $(x_1,x_2,x_3,\ldots,x_n)$ $(y)$

Unknown weights $w$:

$w_{11}$, $w_{12}$,.....

Randomly initialize weights
Repeat
for each example, compute output $o$
calculate error $E = (o-y)^2$
compute the derivative of $E$ over $w$: $dw=\frac{\partial E}{\partial w}$
$w_{new} = w_{prev} - \eta * dw$
Until error doesn't decrease or max num of iterations

Note: $\eta$ is learning rate or step size.



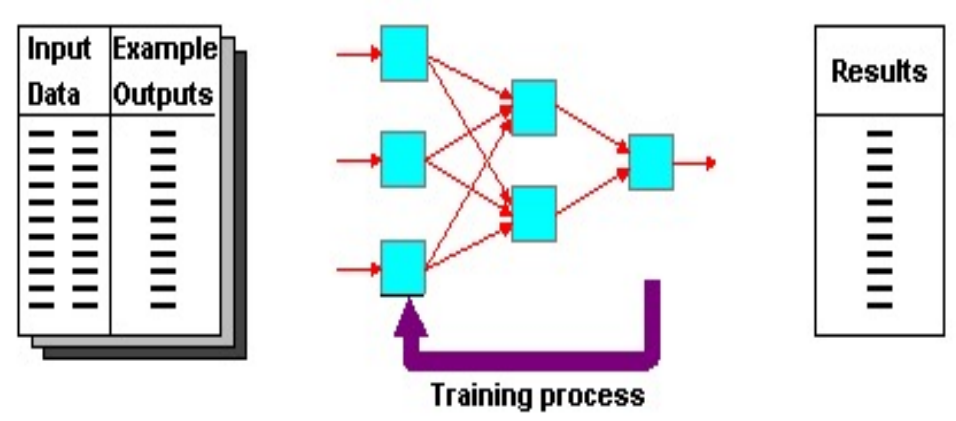
# GRADIENT DESCENT LEARNING RULE

- CONSIDER LINEAR UNIT WITHOUT THRESHOLD AND CONTINUOUS OUTPUT O (NOT JUST –1,1)
  - $Y=W_0 + W_1 X_1 + \ldots + W_N X_N$
- TRAIN THE $W_I$'S SUCH THAT THEY MINIMIZE THE SQUARED ERROR
  - $E[W_1,\ldots,W_N] = \frac{1}{2} \sum_{D \in D} (T_D - Y_D)^2$

  WHERE D IS THE SET OF TRAINING EXAMPLES

# SUPERVISED LEARNING

- TRAINING AND TEST DATA SETS
- TRAINING SET; INPUT & TARGET




| Sepal length | Sepal width | Petal length | Petal width | Class |
|---|---|---|---|---|
| 5.1 | 3.5 | 1.4 | 0.2 | 0 |
| 4.9 | 3.0 | 1.4 | 0.2 | 2 |
| 4.7 | 3.2 | 1.3 | 0.2 | 0 |
| 4.6 | 3.1 | 1.5 | 0.2 | 1 |

# GRADIENT DESCENT

D={<(1,1),1>,<(-1,-1),1>,
<(1,-1),-1>,<(-1,1),-1>}

Gradient:
$\nabla E[w]=[\partial E/\partial w_0,\ldots \partial E/\partial w_n]$

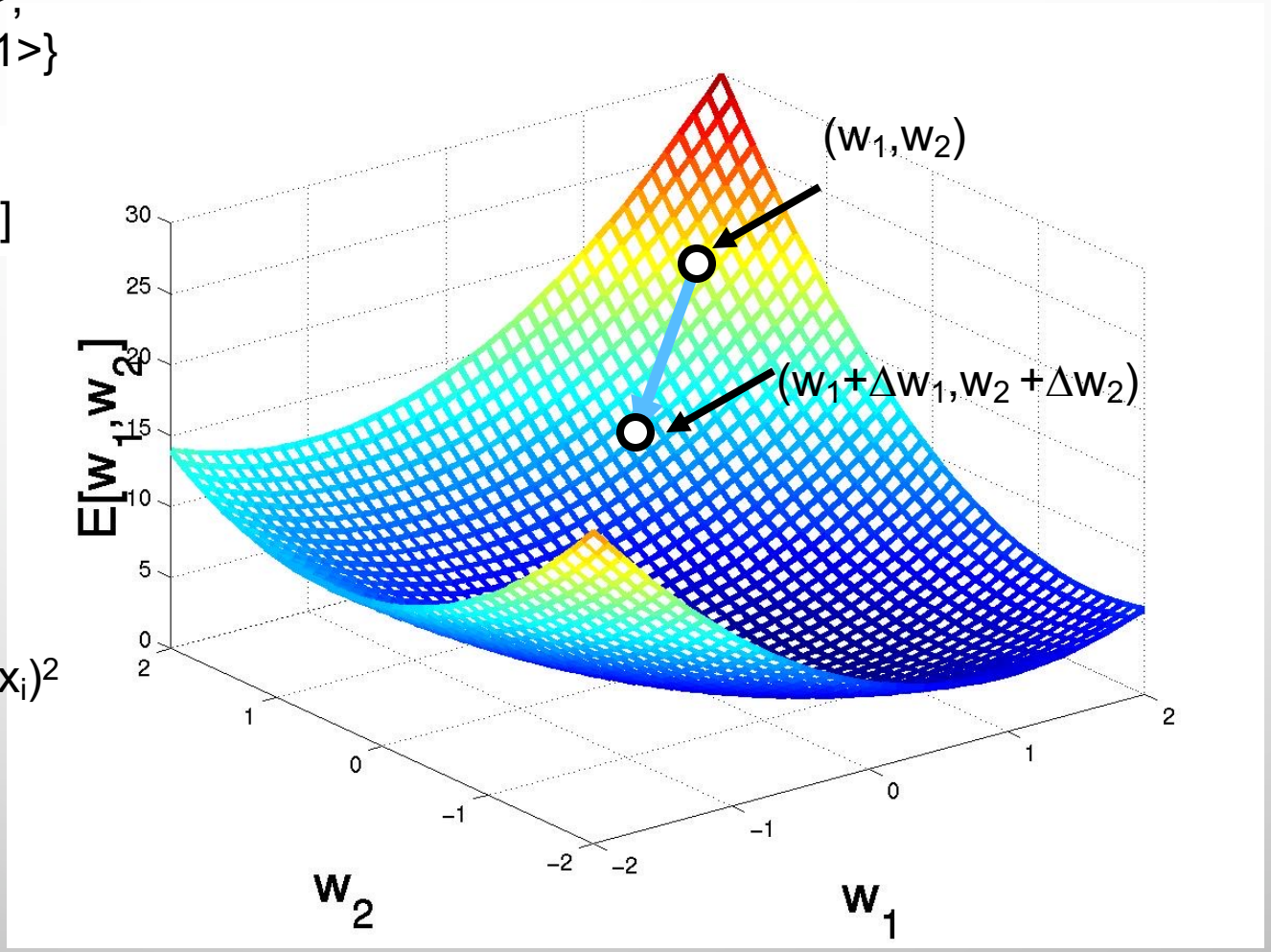
$\Delta w=-\eta\ \nabla E[w]$

$\Delta w_i=-\eta\ \partial E/\partial w_i$

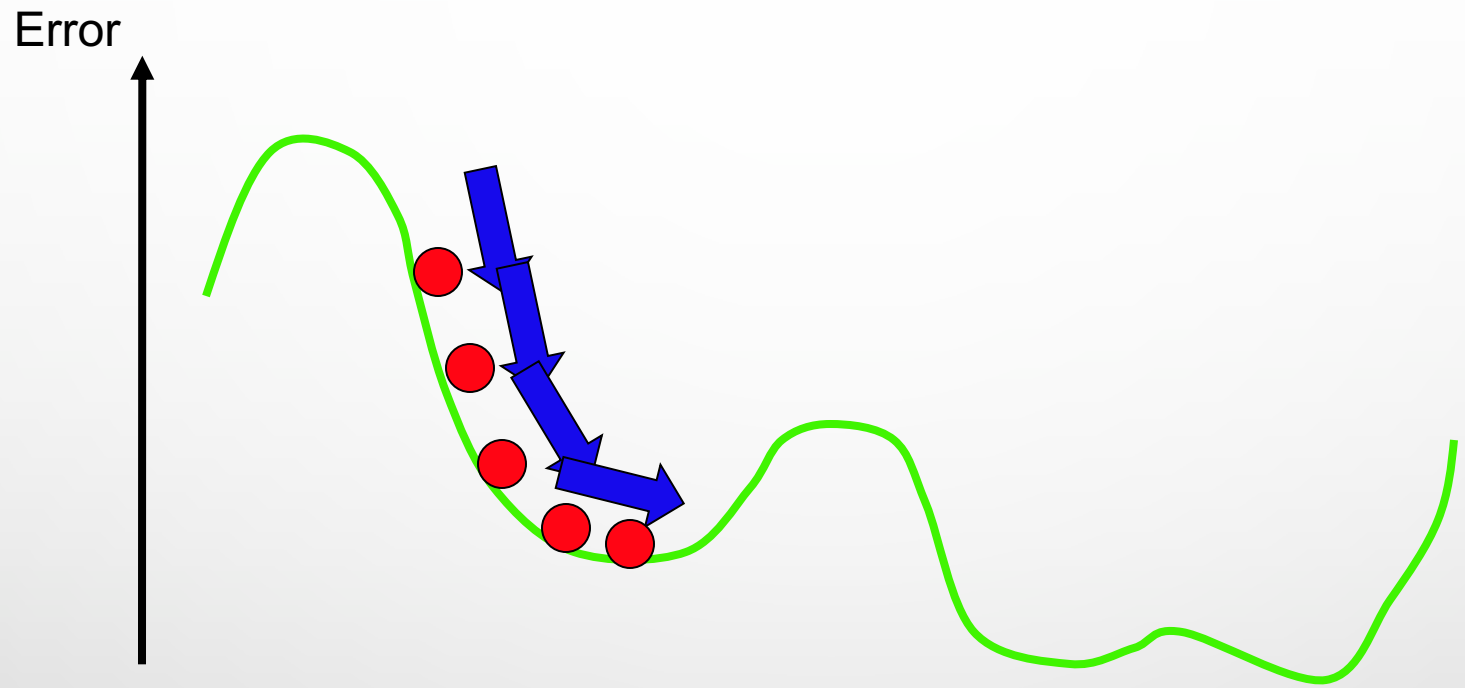
$\partial/\partial w_i\ 1/2\sum_d(t_d-y_d)^2$

$=\sum_d \partial/\partial w_i\ 1/2(t_d-\sum_i w_i x_i)^2$

$=\sum_d(t_d- y_d)(-x_i)$



# GRADIENT DESCENT




$$\Delta w_i = -\eta \, \partial E / \partial w_i$$

# INCREMENTAL STOCHASTIC GRADIENT DESCENT

- BATCH MODE : GRADIENT DESCENT

  $W=W - \eta \nabla E_D[W]$ OVER THE ENTIRE DATA D

  $E_D[W]=1/2\Sigma_D(T_D-Y_D)^2$

- INCREMENTAL MODE: GRADIENT DESCENT

  $W=W - \eta \nabla E_D[W]$ OVER INDIVIDUAL TRAINING EXAMPLES D

  $E_D[W]=1/2\ (T_D-Y_D)^2$

INCREMENTAL GRADIENT DESCENT CAN APPROXIMATE BATCH GRADIENT DESCENT ARBITRARILY CLOSELY IF $\eta$ IS SMALL ENOUGH

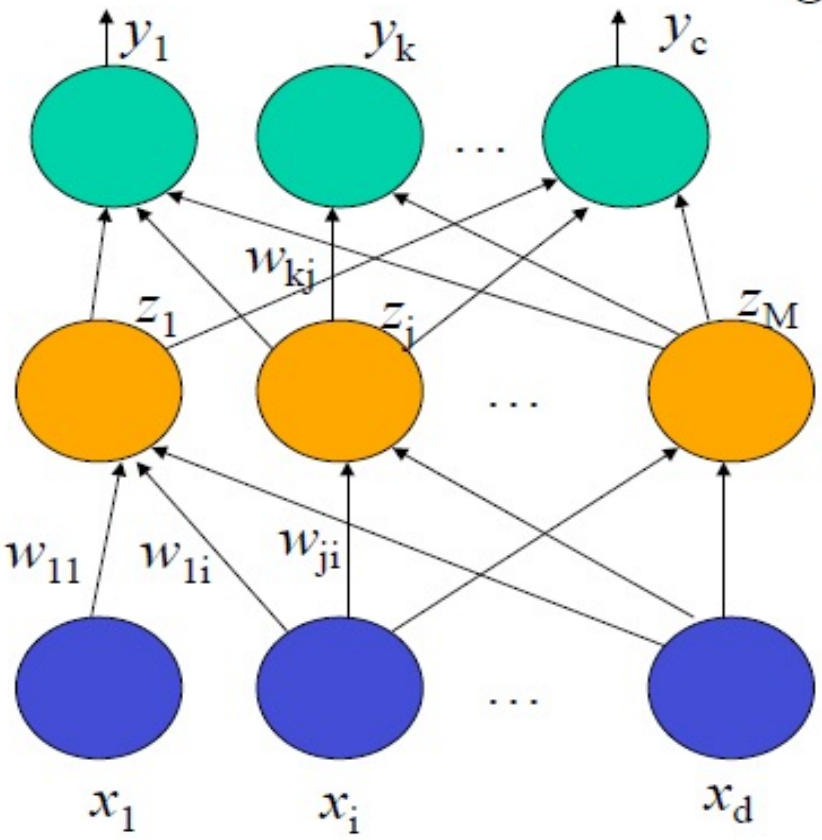
# GRADIENT DESCENT PERCEPTRON LEARNING

GRADIENT-DESCENT(*TRAINING_EXAMPLES*, $\eta$)

EACH TRAINING EXAMPLE IS A PAIR OF THE FORM $<(X_1,\ldots X_N),T>$ WHERE $(X_1,\ldots,X_N)$ IS THE VECTOR OF INPUT VALUES, AND T IS THE TARGET OUTPUT VALUE, $\eta$ IS THE LEARNING RATE (E.G. 0.1)

- INITIALIZE EACH $W_I$ TO SOME SMALL RANDOM VALUE
- UNTIL THE TERMINATION CONDITION IS MET, DO
  - FOR EACH $<(X_1,\ldots X_N),T>$ IN *TRAINING_EXAMPLES* DO
    - INPUT THE INSTANCE $(X_1,\ldots,X_N)$ TO THE LINEAR UNIT AND COMPUTE THE OUTPUT Y
    - FOR EACH LINEAR UNIT WEIGHT $W_I$ DO
      - $\Delta W_I = \eta\ (T-Y)\ X_I$
  - FOR EACH LINEAR UNIT WEIGHT $W_I$ DO
    - $W_I = W_I + \Delta W_I$

# Forward Propagation




Output $y_k$

Activation function: $f$ (linear,sigmoid, softmax)

Activation of unit $a_k$: $\sum_{j=1}^{M} w_{kj} z_j$

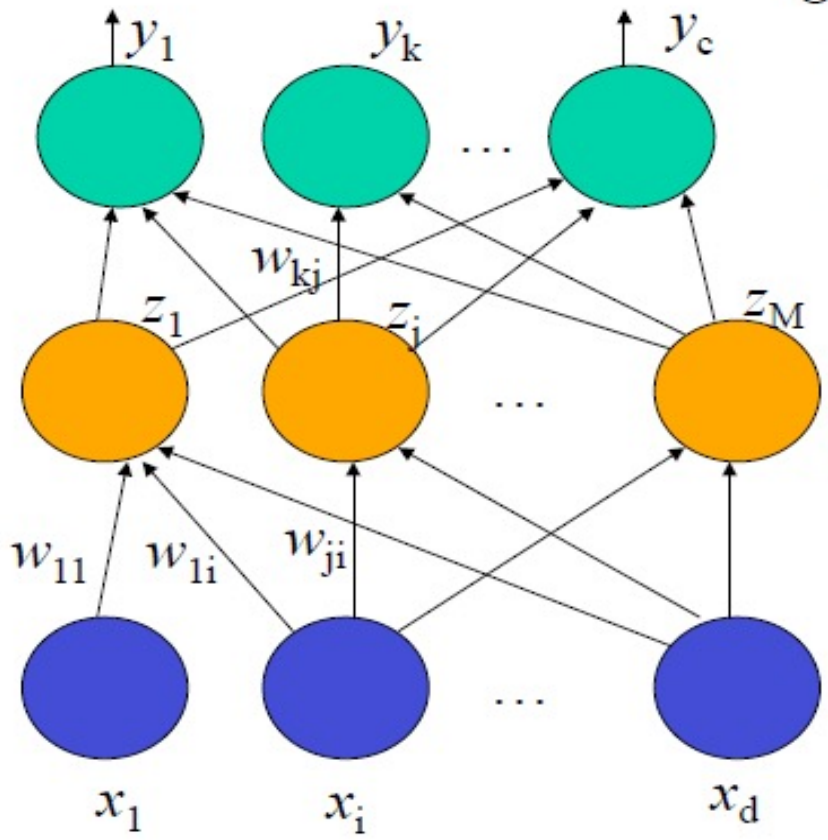
$z_j$

Activation function: $g$ (linear, tanh, sigmoid)

Activation of unit $a_j$: $\sum_{i=1}^{d} w_{ji} x_i$

**Time complexity?**

# Forward Propagation




Output $y_k$

Activation function: $f$ (linear,sigmoid, softmax)

Activation of unit $a_k$: $\sum_{j=1}^{M} w_{kj} z_j$

$z_j$

Activation function: $g$ (linear, tanh, sigmoid)

Activation of unit $a_j$: $\sum_{i=1}^{d} w_{ji} x_i$

**Time complexity?**
**O(dM + MC) = O(W)**

# MATRIX VIEW OF PROPAGATION

| $w_{11}$ | $w_{12}$ | $w_{13}$ | … | $w_{1d}$ |
|---|---|---|---|---|
| $W_{21}$ | $W_{22}$ | … | …. | $w_{2d}$ |
| …. | | | | |
| $W_{m1}$ | $W_{m2}$ | | | $w_{md}$ |

$*\ (X_1, x_2, \ldots, X_d) \qquad = (a_1, a_2, \ldots, a_m)$

**Apply Transfer function**

| $w_{11}$ | $w_{12}$ | $w_{13}$ | … | $w_{1m}$ |
|---|---|---|---|---|
| $W_{21}$ | $W_{22}$ | … | …. | $w_{2c}$ |
| …. | | | | |
| $W_{c1}$ | $W_{m2}$ | | | $w_{cm}$ |

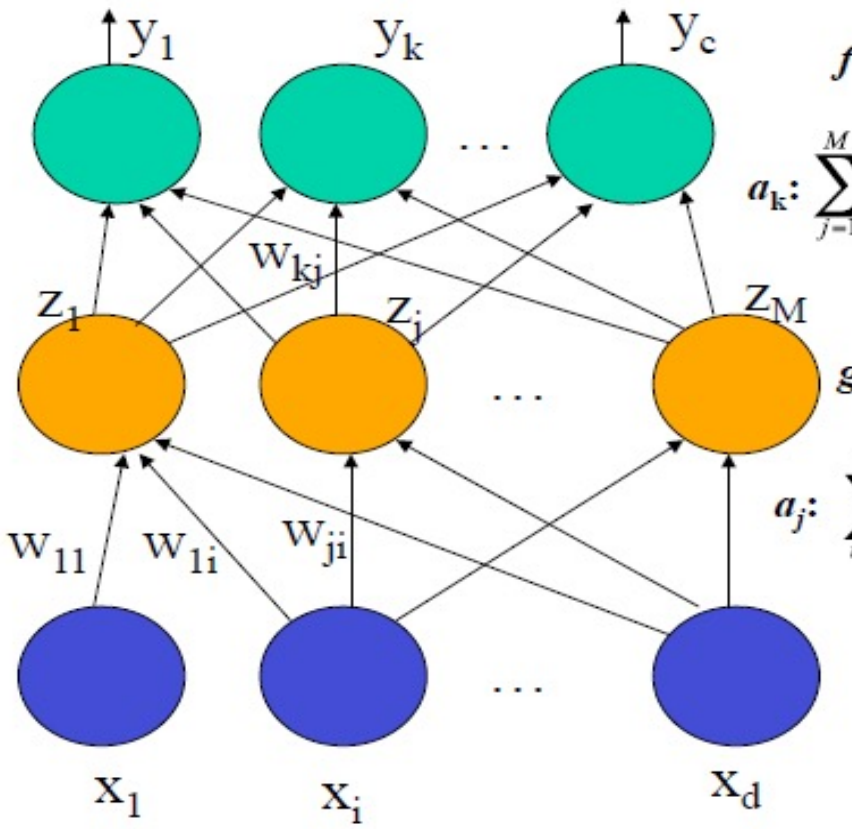
$*\ (z_1, z_2, \ldots, z_m)$

$(b_1, b_2, \ldots, b_c) =$

Apply transfer function

$(O_1, O_2, \ldots, O_c)$

# Backward Propagation




$$a_k: \sum_{j=1}^{M} w_{kj} z_j \qquad a_j: \sum_{i=1}^{d} w_{ji} x_i$$

$$E = \frac{1}{2}\sum_{k=1}^{c}(y_k - t_k)^2$$

$$\frac{\partial E}{\partial y_k} = y_k - t_k$$

$$\frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k} = (y_k - t_k) f'(a_k) = \delta_k$$

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} = \delta_k z_j$$

$$\frac{\partial E}{\partial a_j} = \sum_{k=1}^{c}\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k}\frac{\partial a_k}{\partial z_j}\frac{\partial z_j}{\partial a_j} = \sum_{k=1}^{C}\delta_k w_{kj} g'(a_j) = \delta_j$$

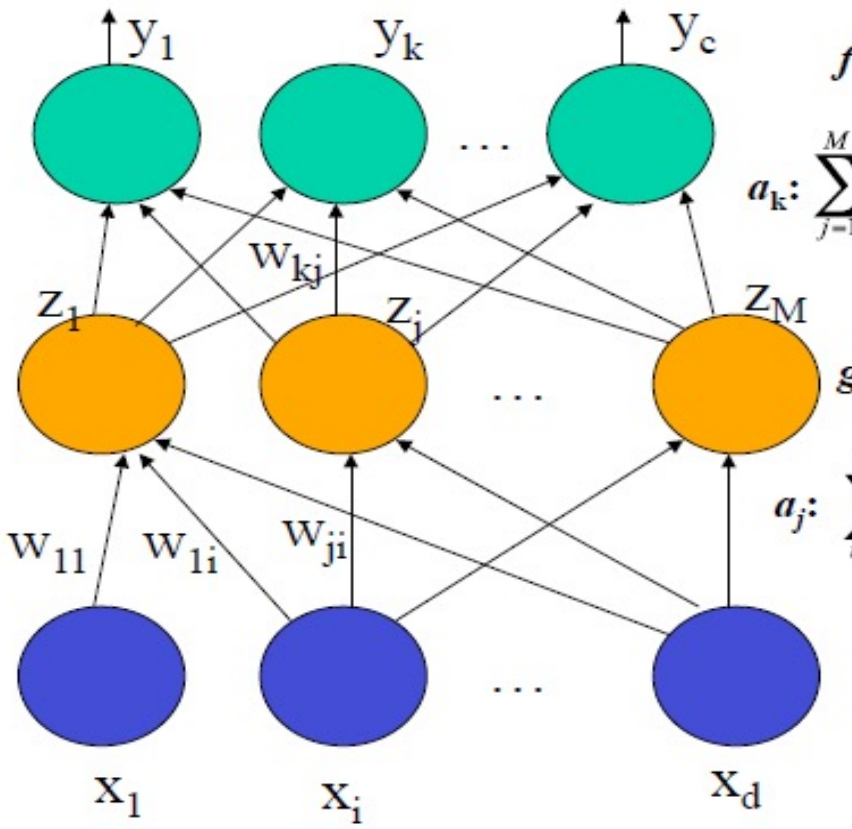
$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial wj\,i} = \delta_j x_i$$

**Chain Rule of Calculating Derivatives**

**Time complexity?**

If no back-propagation, time complexity is: (MdC+CM)

# Backward Propagation




$$E = \frac{1}{2}\sum_{k=1}^{c}(y_k - t_k)^2$$

$$\frac{\partial E}{\partial y_k} = y_k - t_k$$

$$\frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k} = (y_k - t_k) f'(a_k) = \delta_k$$

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} = \delta_k z_j$$

$$\frac{\partial E}{\partial a_j} = \sum_{k=1}^{c}\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k}\frac{\partial a_k}{\partial z_j}\frac{\partial z_j}{\partial a_j} = \sum_{k=1}^{C}\delta_k w_{kj} g'(a_j) = \delta_j$$

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial wj\,i} = \delta_j x_i$$

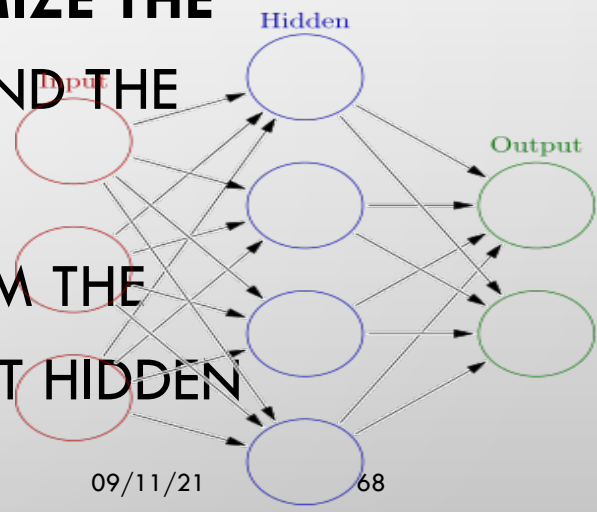
**Chain Rule of Calculating Derivatives**

If no back-propagation, time complexity is: (MdC+CM)

**Time complexity?**
**O(CM+Md) = O(W)**

# BACK PROPAGATION

- **BACK PROPAGATION**: RESET WEIGHTS ON THE "FRONT" NEURAL UNITS AND THIS IS SOMETIMES DONE IN COMBINATION WITH TRAINING WHERE THE CORRECT RESULT IS KNOWN
- ITERATIVELY PROCESS A SET OF TRAINING TUPLES & COMPARE THE NETWORK'S PREDICTION WITH THE ACTUAL KNOWN TARGET VALUE
- FOR EACH TRAINING TUPLE, THE WEIGHTS ARE MODIFIED TO **MINIMIZE THE MEAN SQUARED ERROR** BETWEEN THE NETWORK'S PREDICTION AND THE ACTUAL TARGET VALUE
- MODIFICATIONS ARE MADE IN THE "**BACKWARDS**" DIRECTION: FROM THE OUTPUT LAYER, THROUGH EACH HIDDEN LAYER DOWN TO THE FIRST HIDDEN LAYER, HENCE "**BACKPROPAGATION**"
- STEPS



## The Multi-layer Perceptron Algorithm

- **Initialisation**
  - initialise all weights to small (positive and negative) random values
- **Training**
  - repeat:
    - for each input vector:

      **Forwards phase:**

      - compute the activation of each neuron $j$ in the hidden layer(s) using:

$$h_\zeta = \sum_{i=0}^{L} x_i v_{i\zeta} \tag{4.4}$$

$$a_\zeta = g(h_\zeta) = \frac{1}{1+\exp(-\beta h_\zeta)} \tag{4.5}$$

      - work through the network until you get to the output layer neurons, which have activations (although see also Section 4.2.3):

$$h_\kappa = \sum_j a_j w_{j\kappa} \tag{4.6}$$

$$y_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)} \tag{4.7}$$

      **Backwards phase:**

      - compute the error at the output using:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)\, y_\kappa (1 - y_\kappa) \tag{4.8}$$

      - compute the error in the hidden layer(s) using:

$$\delta_h(\zeta) = a_\zeta (1 - a_\zeta) \sum_{k=1}^{N} w_\zeta \delta_o(k) \tag{4.9}$$

      - update the output layer weights using:

$$w_{\zeta\kappa} \leftarrow w_{\zeta\kappa} - \eta \delta_o(\kappa) a_\zeta^{\text{hidden}} \tag{4.10}$$

      - update the hidden layer weights using:

$$v_\iota \leftarrow v_\iota - \eta \delta_h(\kappa) x_\iota \tag{4.11}$$

    - (if using sequential updating) randomise the order of the input vectors so that you don't train in exactly the same order each iteration
  - until learning stops (see Section 4.3.3)
- **Recall**
  - use the Forwards phase in the training section above

# SEQUENTIAL AND BATCH TRAINING

The MLP is designed to be a **batch algorithm.**

- All the training examples are presented to the neural network, **the average sum-of-squares error** is then computed, and this is used to update the weights. Thus, there is **only one set of weight updates for each epoch** (pass through all the training examples).
- Only update the weights once for each iteration of the algorithm
- The weights are moved in the direction that most of the inputs want them to move, rather than being pulled around by each input
- The batch method performs a more accurate estimate of the error gradient and will thus converge to the local minimum more quickly.

# SEQUENTIAL AND BATCH TRAINING

The algorithm that was described in the previous slide was the **sequential version**, where the errors are computed, and the weights updated after each input.

- This is not guaranteed to be as efficient in learning, but it is simpler to program when using loops,
- Therefore, **much more common**. Since it does not converge as well, it can also sometimes avoid local minima, thus potentially reaching better solutions.

# DEFINING A NETWORK TOPOLOGY

- DECIDE THE **NETWORK TOPOLOGY**
  - SPECIFY # OF UNITS IN THE *INPUT LAYER*, # OF *HIDDEN LAYERS* (IF > 1), # OF UNITS IN *EACH HIDDEN LAYER*, AND # OF UNITS IN THE *OUTPUT LAYER*
- NORMALIZE THE INPUT VALUES FOR EACH ATTRIBUTE MEASURED IN THE TRAINING TUPLES TO [0.0—1.0]
- ONE **INPUT** UNIT PER DOMAIN VALUE, EACH INITIALIZED TO 0
- **OUTPUT**, IF FOR CLASSIFICATION AND MORE THAN TWO CLASSES, ONE OUTPUT UNIT PER CLASS IS USED
- ONCE A NETWORK HAS BEEN TRAINED AND ITS ACCURACY IS **UNACCEPTABLE**, REPEAT THE TRAINING PROCESS WITH A *DIFFERENT NETWORK TOPOLOGY* OR A *DIFFERENT SET OF INITIAL WEIGHTS*

# Test Phase and Prediction

- Weights are known
- Given an input x, neural network generates an output O
- Binary classification: O > 0.5 → positive, O <= 0.5 → negative
- Multi-classification: Choose the class of the output node with highest value
- Regression: O is the predicted value

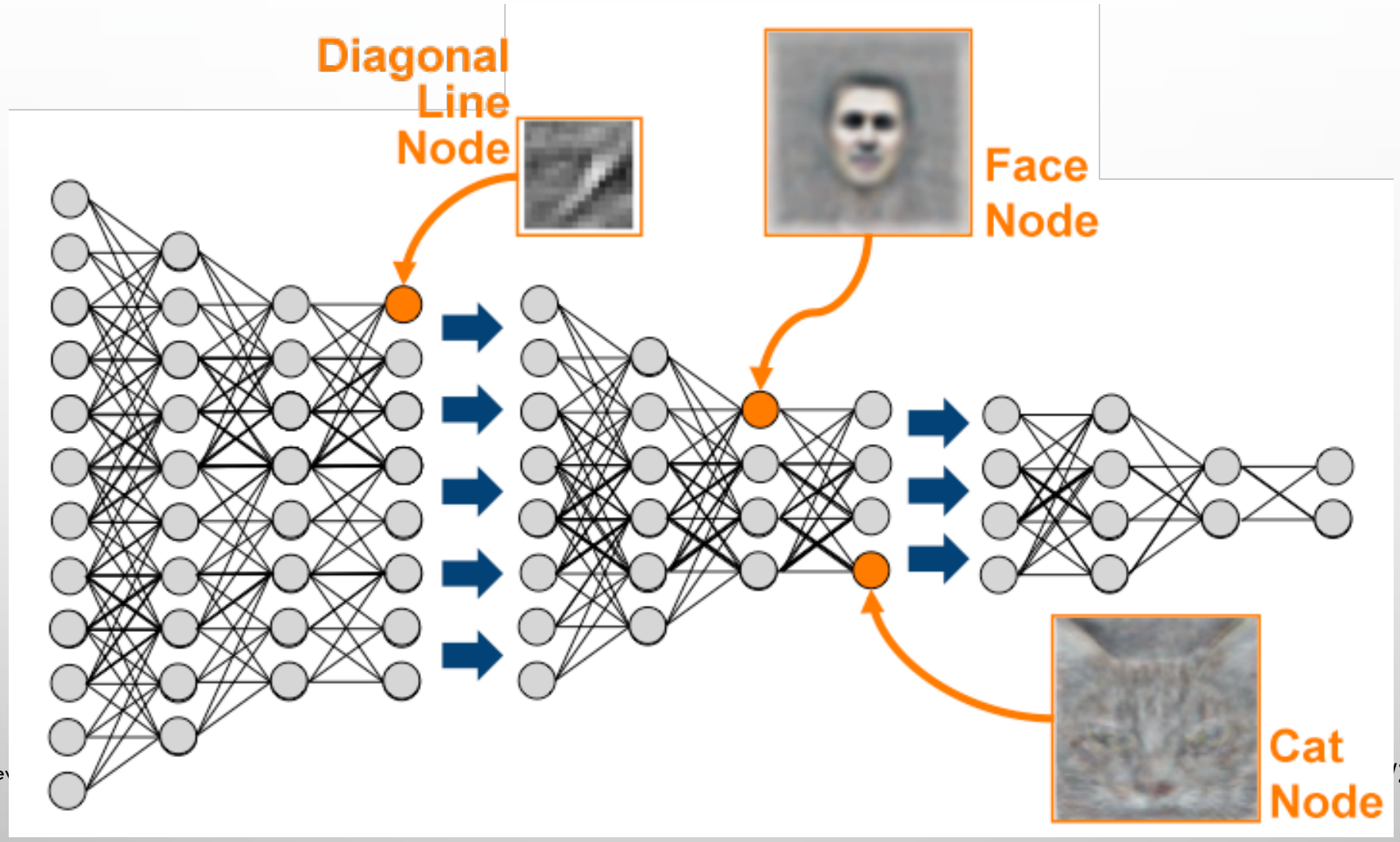
# NEURAL NETWORK AS A CLASSIFIER

- WEAKNESS
  - LONG TRAINING TIME
  - REQUIRE A NUMBER OF PARAMETERS TYPICALLY BEST DETERMINED EMPIRICALLY, E.G., THE NETWORK TOPOLOGY
  - POOR INTERPRETABILITY: DIFFICULT TO INTERPRET THE SYMBOLIC MEANING BEHIND THE LEARNED WEIGHTS AND OF ``HIDDEN UNITS" IN THE NETWORK
- STRENGTH
  - HIGH TOLERANCE TO NOISY DATA
  - ABILITY TO CLASSIFY UNTRAINED PATTERNS
  - WELL-SUITED FOR CONTINUOUS-VALUED INPUTS AND OUTPUTS
  - SUCCESSFUL ON A WIDE ARRAY OF REAL-WORLD DATA
  - ALGORITHMS ARE INHERENTLY PARALLEL
  - TECHNIQUES HAVE BEEN DEVELOPED FOR THE EXTRACTION OF RULES FROM TRAINED NEURAL NETWORKS

# FROM NEURAL NETWORKS TO DEEP LEARNING

- TRAIN NETWORKS WITH MANY LAYERS (VS. SHALLOW NETS WITH JUST A COUPLE OF LAYERS)
- MULTIPLE LAYERS WORK TO BUILD AN IMPROVED FEATURE SPACE
  - FIRST LAYER LEARNS 1ST ORDER FEATURES (E.G., EDGES, …)
  - 2ND LAYER LEARNS HIGHER ORDER FEATURES (COMBINATIONS OF FIRST LAYER FEATURES, COMBINATIONS OF EDGES, ETC.)
  - IN CURRENT MODELS, LAYERS OFTEN LEARN IN AN UNSUPERVISED MODE AND DISCOVER GENERAL FEATURES OF THE INPUT SPACE—SERVING MULTIPLE TASKS RELATED TO THE UNSUPERVISED INSTANCES (IMAGE RECOGNITION, ETC.)
  - THEN FINAL LAYER FEATURES ARE FED INTO SUPERVISED LAYER(S)
    - AND ENTIRE NETWORK IS OFTEN SUBSEQUENTLY TUNED USING SUPERVISED TRAINING OF THE ENTIRE NET, USING THE INITIAL WEIGHTS LEARNED IN THE UNSUPERVISED PHASE
  - COULD ALSO DO FULLY SUPERVISED VERSIONS (BACK-PROPAGATION)

# MOST RECENT DEVELOPMENT – DEEP LEARNING



# BUILDING MLP IN WEKA FOR WEATHER NUMERIC DATASET

THE UNIVERSITY OF WAIKATO
*Te Whare Wānanga o Waikato*

## *More Data Mining with Weka*

Class 5 – Lesson 2

*Multilayer Perceptrons*

Ian H. Witten

Department of Computer Science
University of Waikato
New Zealand

# NN WITH R

- SINGLE-HIDDEN-LAYER NEURAL NETWORK ARE IMPLEMENTED IN PACKAGE **NNET (SHIPPED WITH BASE R).**
- PACKAGE RSNNS OFFERS AN INTERFACE TO THE STUTTGART NEURAL NETWORK SIMULATOR (SNNS).
- PACKAGES IMPLEMENTING DEEP LEARNING FLAVOURS OF NEURAL NETWORKS INCLUDE DEEPNET

# NN WITH R

HTTPS://WWW.RDOCUMENTATION.ORG/PACKAGES/NNET/VERSIONS/7.3-12/TOPICS/NNET

# REFERENCE


1. STEPHAN MARSLAND, **MACHINE LEARNING, AN ALGORITHMIC PERSPECTIVE**, CRC PRESS SECOND EDITION, 2015.
2. J. HAN, M. KAMBER, **DATA MINING: CONCEPTS AND TECHNIQUES**, ELSEVIER INC. (2006).
3. I. H. WITTEN AND E. FRANK, **DATA MINING: PRACTICAL MACHINE LEARNING TOOLS AND TECHNIQUES**, 2ND EDITION, ELSEVIER INC., 2005.