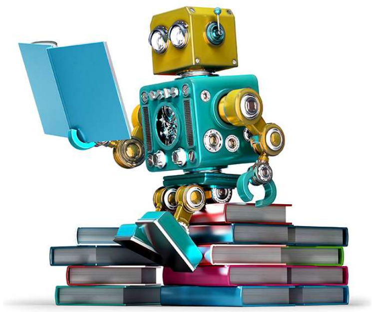
MACHINE LEARNING



MODULE-III
ARTIFICIAL NEURAL NETWORKS

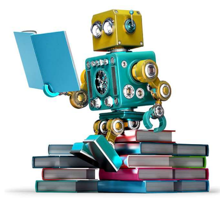
BY
HARIVINOD N
VIVEKANANDA COLLEGE OF ENGINEERING TECHNOLOGY, PUTTUR

Module 3- Outline

MACHINE LEARNING

Artificial Neural Network

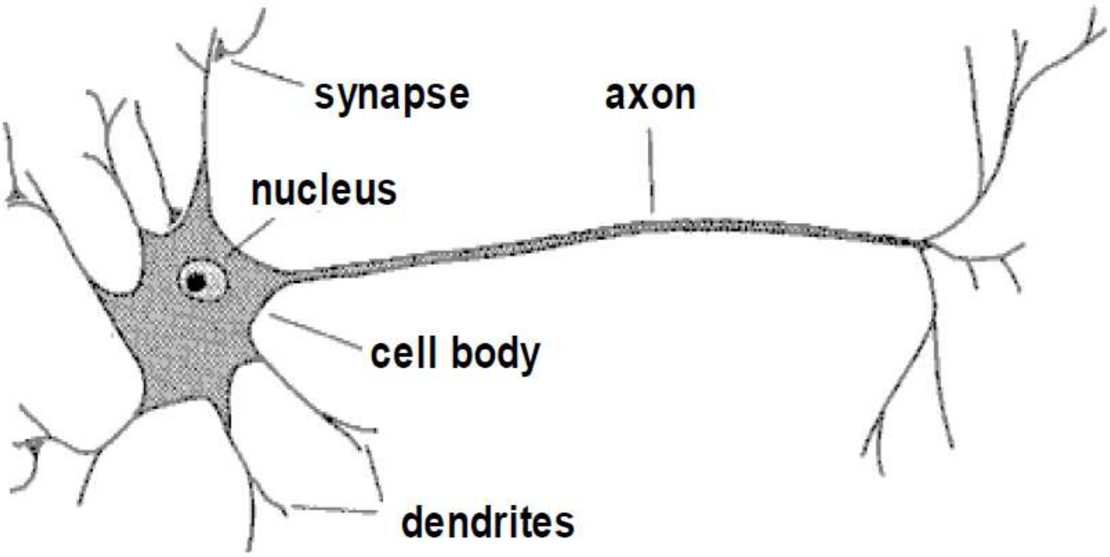
1. **Biological Motivation**
2. Neural Network Representation
3. Appropriate Problems for NN learning
4. Perceptions
5. Multilayer Networks and Backpropagation Algorithm
6. Remarks on Backpropagation Algorithm
7. Summary



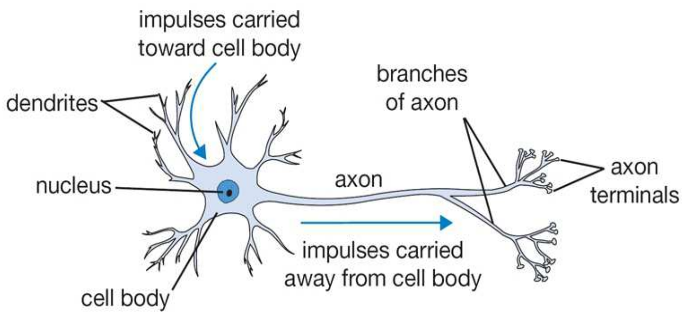
Biological Motivation

- The basic computational unit of the brain is a **neuron**.

Neurons



Biological Motivation



- biological learning systems are built of complex webs of interconnected neurons

- **motivation:**
  - capture kind of highly parallel computation
  - based on distributed representation
- **goal:**
  - obtain highly effective machine learning algorithms, independent of whether these algorithms fit biological processes (*no cognitive modeling!*)

Biological Motivation

	Computer	Brain
computation units	1 CPU ($> 10^7$ Gates)	10^{11} neurons
memory units	512 MB RAM	10^{11} neurons
	500 GB HDD	10^{14} synapses
clock	10^{-8} sec	10^{-3} sec
transmission	$> 10^9$ bits/sec	$> 10^{14}$ bits/sec

- Computer: serial, quick
- Brain: parallel, slowly, robust to noisy data

Properties of NNs

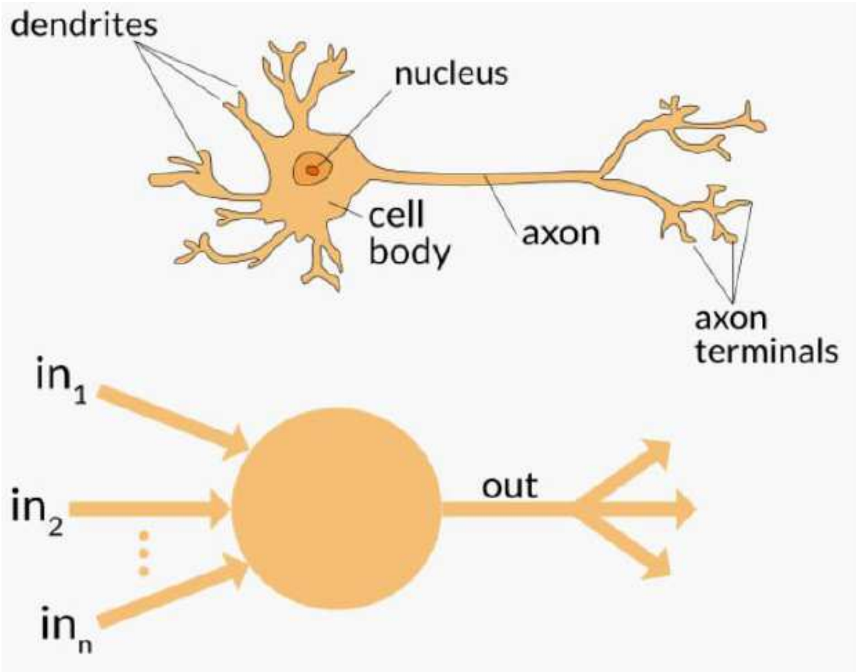
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically
- Input is a high-dimensional discrete or real-valued (e.g, sensor input)

Module 3- Outline

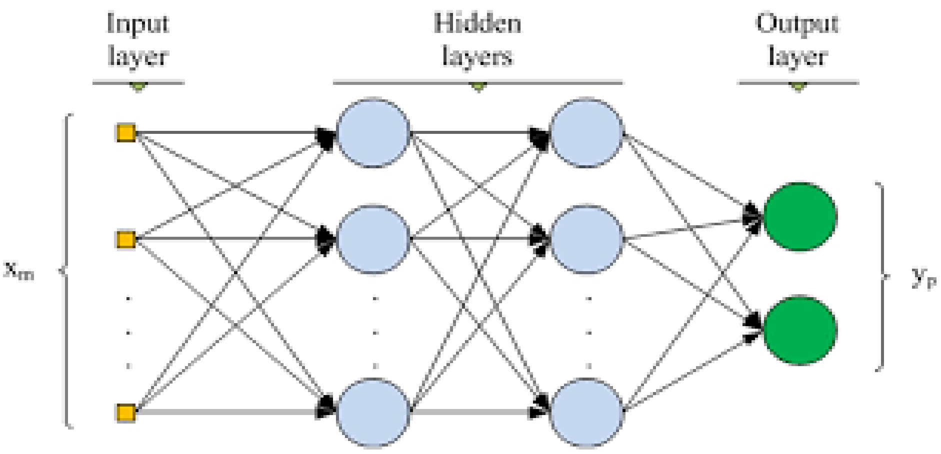
Artificial Neural Network

1. Biological Motivation
2. **Neural Network Representation**
3. Appropriate Problems for NN learning
4. Perceptions
5. Multilayer Networks and Backpropagation Algorithm
6. Remarks on Backpropagation Algorithm
7. Summary

Neuron

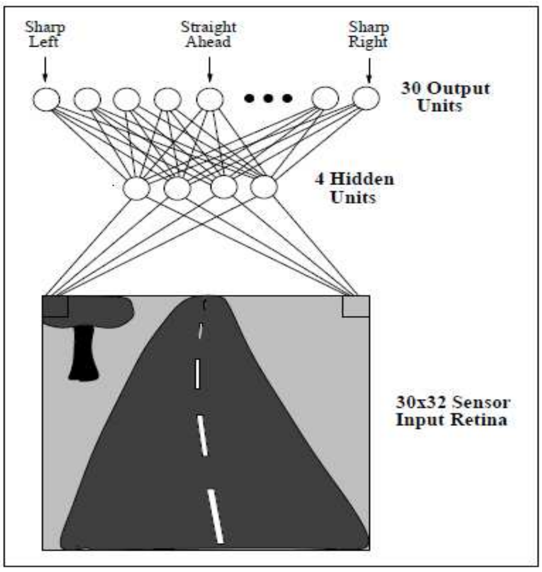


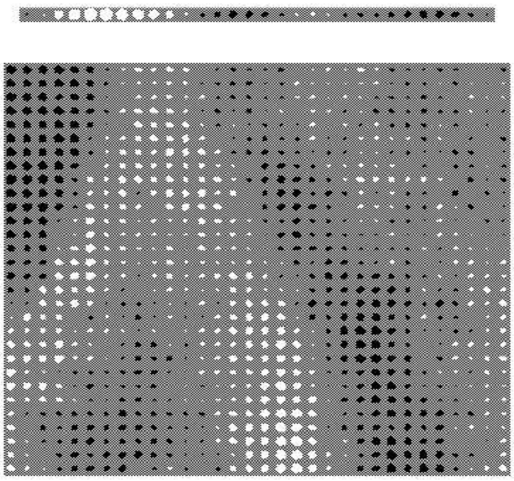
Typical NN



Example: Autonomous Driving

Autonomous Driving

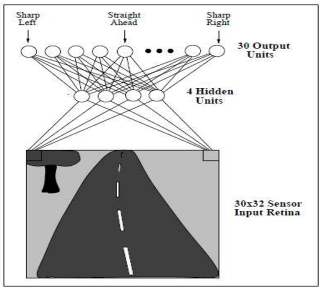


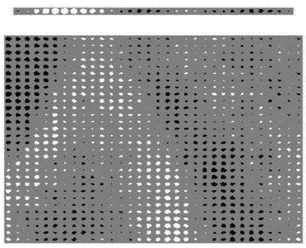


Autonomous Driving

- A prototypical example of ANN learning is provided by Pomerleau's (1993) system ALVINN, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways.
- The ***input*** to the neural network is a 30x32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle.
- The network ***output*** is the direction in which the vehicle is steered.

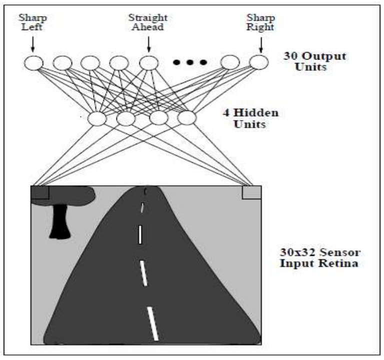
Autonomous driving

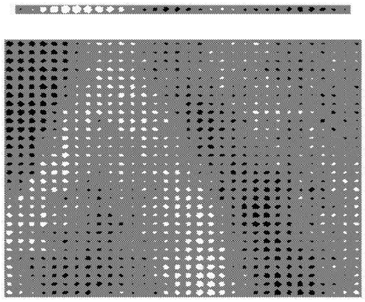




- The network is shown on the left side of the figure, with the input camera image depicted below it.
- Each node (i.e., circle) in the network diagram corresponds to the output of a single network ***unit,*** and the lines entering the node from below are its *inputs*.
- There are four units that receive inputs directly from all of the 30 x 32 pixels in the image. These are called *"hidden" units* because their output is available only within the network and is not available as part of the global network output. Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs
- These hidden unit outputs are then used as inputs to a second layer of 30 "output" units.
- Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.

Autonomous Driving





- The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN.
- The large matrix of black and white boxes on the lower right depicts the weights from the 30 x 32 pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude.
- The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

Example 2: Bank Credit Score

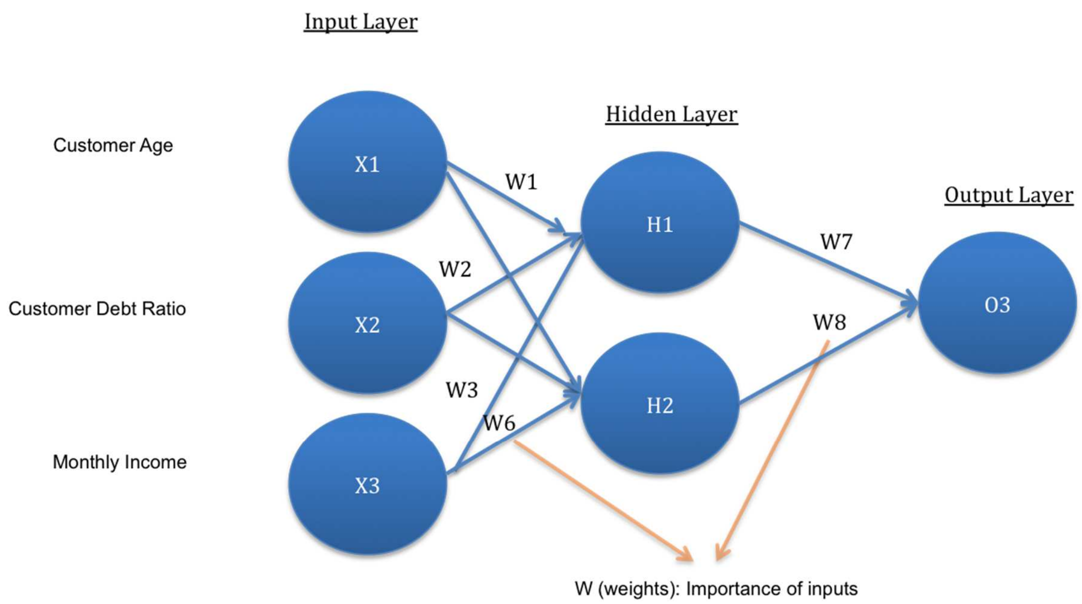
- Credit Scoring
  - Determine whether a load should be approved based on features extracted from applicant's information
- Inputs:
  - Own/Rent your home, Years with Employer, Credit Cards, Store Account, Bank Account, Occupation, Previous Account, Credit Bureau
- Outputs:
  - credit scores: delinquent, charged-off, or paid-off

Example 2: Bank Credit Score

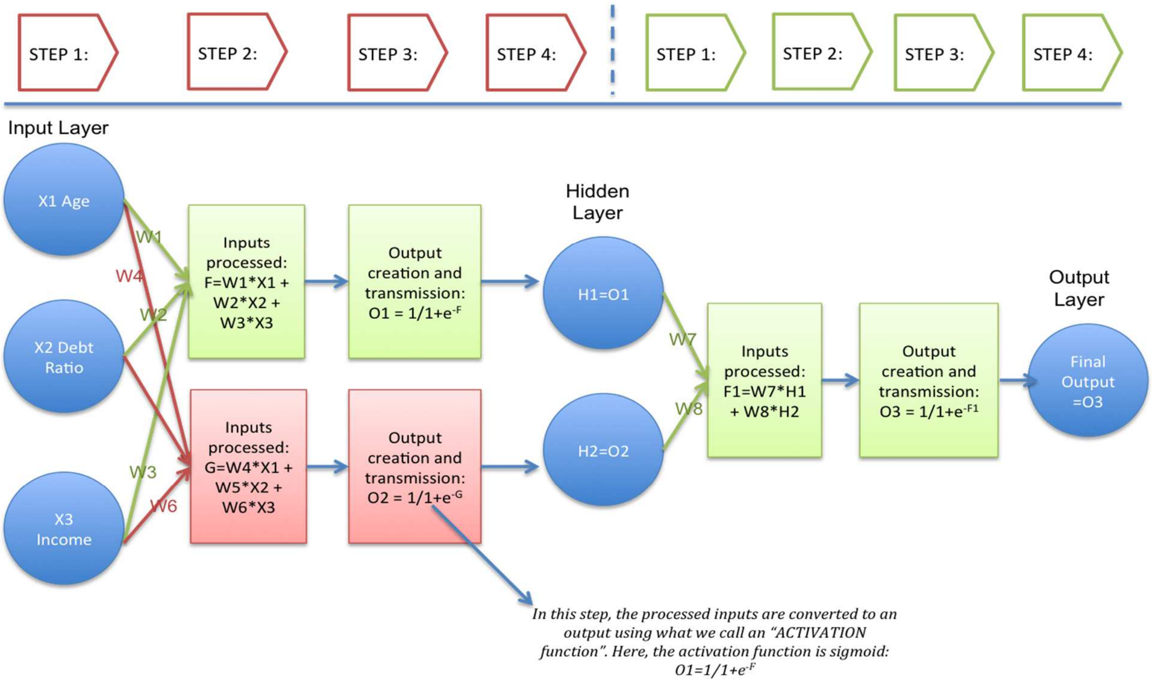
- To make things clearer, lets understand ANN using a simple example: A bank wants to assess whether to approve a loan application to a customer, so, it wants to predict whether a customer is likely to default on the loan. It has data like below:

Customer ID	Customer Age	Debt Ratio (% of Income)	Monthly Income ($)	Loan Defaulter Yes:1 No:0 (Column W)	Default Prediction (Column X)
1	45	0.80	9120	1	0.76
2	40	0.12	2000	1	0.66
3	38	0.08	3042	0	0.34
4	25	0.03	3300	0	0.55
5	49	0.02	63588	0	0.15
6	74	0.37	3500	0	0.72

Example 2: Bank Credit Score



Example 2: Bank Credit Score



Example 2: Bank Credit Score

Customer ID	Customer Age	Debt Ratio (% of Income)	Monthly Income ($)	Loan Defaulter Yes:1 No:0 (Column W)	Default Prediction (Column X)	Prediction Error
1	45	0.80	9120	1	0.76	0.24
2	40	0.12	2000	1	0.66	0.34
3	38	0.08	3042	0	0.34	-0.34
4	25	0.03	3300	0	0.55	-0.55
5	49	0.02	63588	0	0.15	-0.15
6	74	0.37	3500	0	0.72	-0.72

Some Applications

- Autonomous Driving
- Speech Phenome Recognition
- Image Classification
- Financial Prediction

Module 3- Outline

Artificial Neural Network

1. Biological Motivation
2. Neural Network Representation
3. **Appropriate Problems for NN learning**
4. Perceptions
5. Multilayer Networks and Backpropagation Algorithm
6. Remarks on Backpropagation Algorithm
7. Summary

Appropriate problems for ANN

BACKPROPAGATION algorithm is the most commonly used ANN learning technique with the following characteristics:

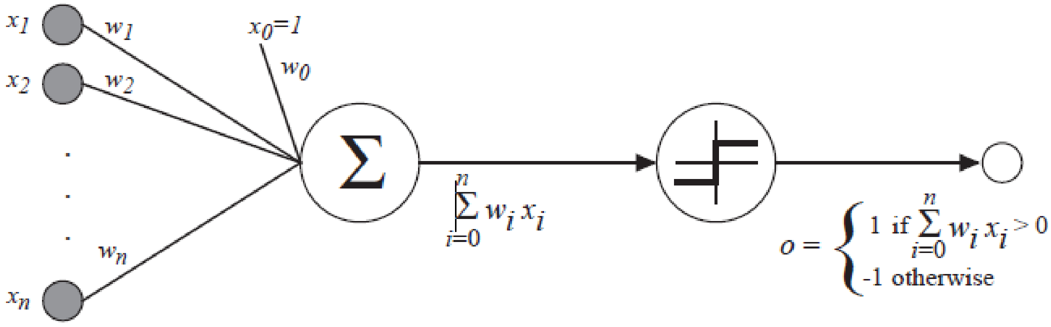
- instances are representated as many attribute-value pairs
  - input values can be any real values
- target function output may be **discrete-, real- or vector-valued**
- training examples **may contain errors**
- long training times are acceptable
- fast evaluation of the learned target function may be required
  - many iterations may be neccessary to converge to a good approximation
- ability of humans to understand the learned target function is not important
  - learned weights are not intuitively understandable

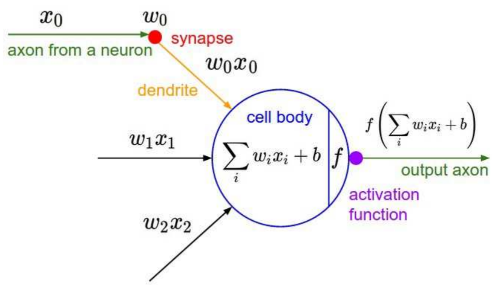
Module 3- Outline

Artificial Neural Network

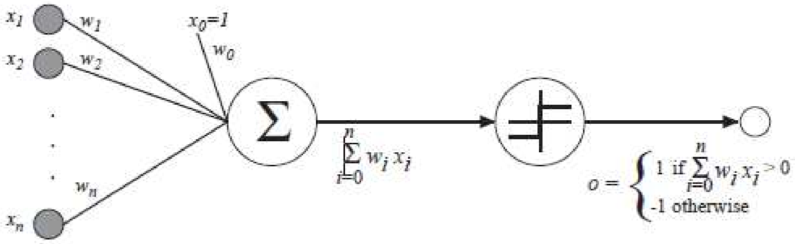
1. Biological Motivation
2. Neural Network Representation
3. Appropriate Problems for NN learning
4. **Perceptions**
5. Multilayer Networks and Backpropagation Algorithm
6. Remarks on Backpropagation Algorithm
7. Summary

A Perceptron

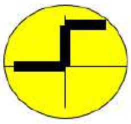




Perceptron



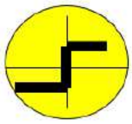
- takes a vector of real-valued inputs $(x_1, \ldots, x_n)$ weighted with $(w_1, \ldots, w_n)$
- calculates the linear combination of these inputs
  - $\sum_{i=0}^{n} w_i x_i = w_0 x_0 + w_1 x_1 + \ldots + w_n x_n$
  - w_0 denotes a threshold value
  - x_0 is always 1
- outputs 1 if the result is greater than 1, otherwise -1



Step function

(Linear Threshold Unit)

step(x) = 1, if x >= threshold
0, if x < threshold

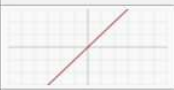
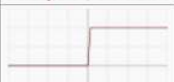
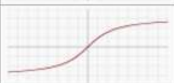
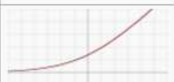


Sign function

sign(x) = +1, if x >= 0
-1, if x < 0

Sigmoid function

sigmoid(x) = $1/(1+e^{-x})$

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Architectures

MACHINE LEARNING

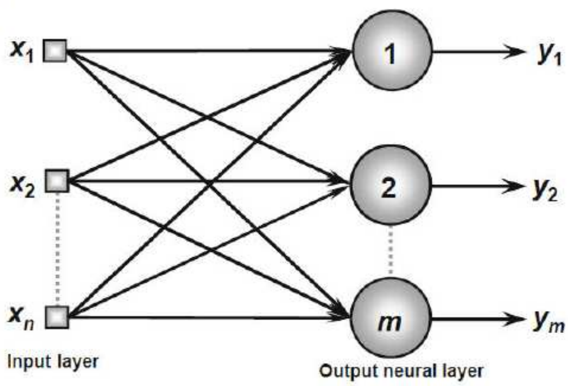
The main architectures of artificial neural networks, considering the neuron disposition, how they are interconnected and how its layers are composed, can be divided as follows:

1. Single-layer feedforward network
2. Multi-layer feedforward networks
3. Recurrent or Feedback networks
4. Mesh networks

MACHINE LEARNING

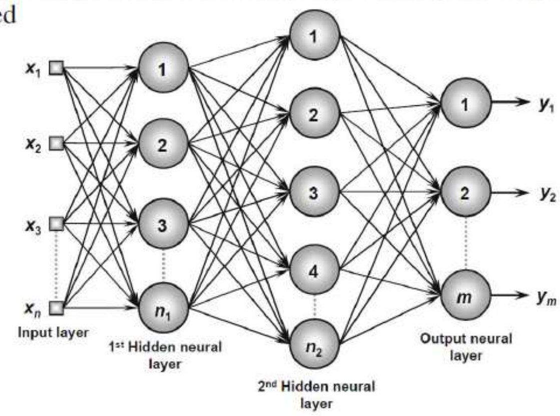
Single-Layer Feedforward Architecture

- This artificial neural network has just one input layer and a single neural layer, which is also the output layer.
- Figure illustrates a simple-layer feedforward network composed of n inputs and m outputs.
- The information always flows in a single direction (thus, unidirectional), which is from the input layer to the output layer



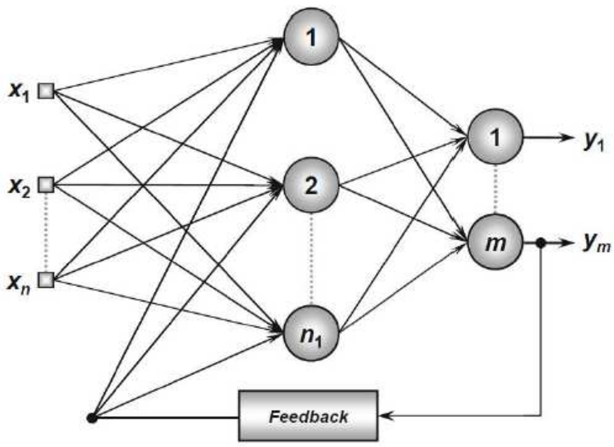
Multi-Layer Feedforward Architecture

- This artificial neural feedforward networks with multiple layers are composed of one or more hidden neural layers.
- Figure shows a feedforward network with multiple layers composed of one input layer with n sample signals, two hidden neural layers consisting of n_1 and n_2 neurons respectively, and, finally, one output neural layer composed of m neurons representing the respective output values of the problem being analyzed



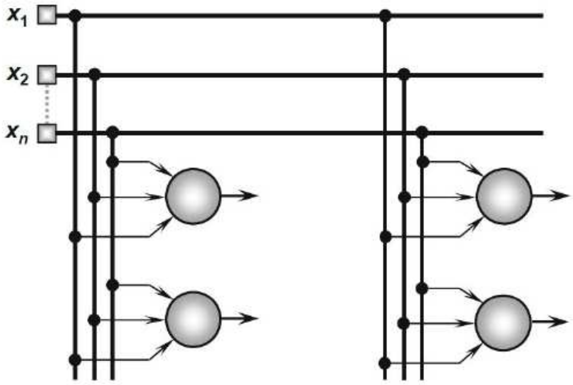
Recurrent or Feedback Architecture

- In these networks, the outputs of the neurons are used as feedback inputs for other neurons.
- Figure illustrates an example of a Perceptron network with feedback, where one of its output signals is fed back to the middle layer.



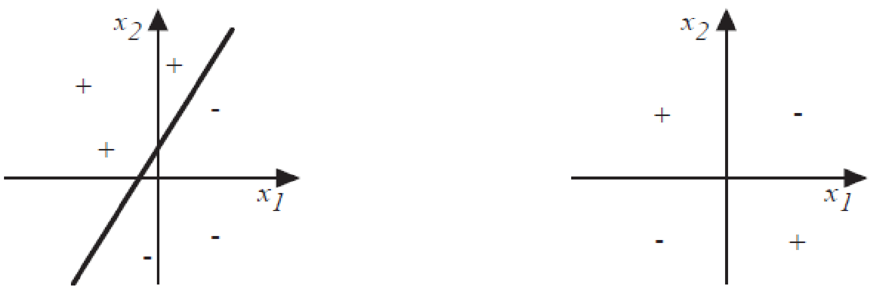
Mesh Architectures

- The main features of networks with mesh structures reside in considering the spatial arrangement of neurons for pattern extraction purposes, that is, the spatial localization of the neurons is directly related to the process of adjusting their synaptic weights and thresholds.
- Figure illustrates an example of the Kohonen network where its neurons are arranged within a two-dimensional space



Representation power

- a perceptron represents a **hyperplane decision surface** in the n-dimensional space of instances
- some sets of examples cannot be separated by any hyperplane, those that can be separated are called **linearly separable**
- many boolean functions can be representated by a perceptron: AND, OR, NAND, NOR

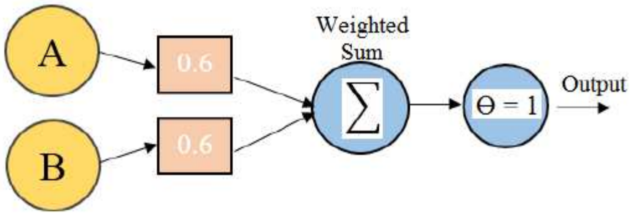


(a) (b)

AND function representation

MACHINE LEARNING

A	B	A ^ B
0	0	0
0	1	0
1	0	0
1	1	1



- If A=0 & B=0 → 0*0.6 + 0*0.6 = 0.
  This is not greater than the threshold of 1, so the output = 0.
- If A=0 & B=1 → 0*0.6 + 1*0.6 = 0.6.
  This is not greater than the threshold, so the output = 0.
- If A=1 & B=0 → 1*0.6 + 0*0.6 = 0.6.
  This is not greater than the threshold, so the output = 0.
- If A=1 & B=1 → 1*0.6 + 1*0.6 = 1.2.
  This exceeds the threshold, so the output = 1.

Key Terms

MACHINE LEARNING

- **Input Nodes (input layer)**
  - Just pass the information to the next layer
  - A block of nodes is also called **layer**.
- **Hidden nodes (hidden layer)**
  - In Hidden layers is where intermediate processing or computation is done,
  - they perform computations and then transfer the weights (signals or information) from the input layer to the next layers
  - It is possible to have a neural network without a hidden layer also.
- **Output Nodes (output layer)**
  - Here we finally use an activation function that maps to the desired output format (e.g. softmax for classification).

Key Terms

- **Connections and weights**
  - The *network* consists of connections, each connection transferring the output of a neuron i to the input of a neuron *j*.
  - In this sense *i* is the predecessor of *j* and *j* is the successor of *i*, Each connection is assigned a weight *Wij.*
- **Activation function**
  - the **activation function** of a node defines the output of that node given an input or set of inputs.
  - A standard computer chip circuit can be seen as a digital network of activation functions that can be "ON" (1) or "OFF" (0), depending on input.
  - In artificial neural networks this function is also called the transfer function.

Key Terms

- **Learning rule**
  - The *learning rule* is a rule or an algorithm which modifies the parameters of the neural network, in order for a given input to the network to produce a favored output.
  - This *learning* process typically amounts to modifying the weights and thresholds.

Perceptron Training Rule

- **problem:** determine a weight vector $\vec{w}$ that causes the perceptron to produce the correct output for each training example
- **perceptron training rule:**
  - $w_i = w_i + \Delta w_i$ where $\Delta w_i = \eta(t - o)x_i$

    t target output

    o perceptron output

    η learning rate (usually some small value, e.g. 0.1)
- **algorithm:**
  1. initialize $\vec{w}$ to random weights
  2. repeat, until each training example is classified correctly
     (a) apply perceptron training rule to each training example
- convergence guaranteed provided linearly separable training examples and sufficiently small η

Illustration – Perceptron learning

Consider following training set

i/p: $x^1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}$ $x^2 = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}$ $x^3 = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$

Desired o/p: $t_1 = -1$ $t_2 = -1$ $t_3 = 1$

Assume learning constant $\eta = 0.1$

Weights are initialised to $w = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}$

Illustration – Perceptron learning

Solution

Algorithm

Step 1: Compute $net = \sum w_i x_i$ [a/w-o/p].

Step 2: $O_i = sign(net)$ [Activation fn]

i.e. $O_i = \begin{cases} +1 & \text{if } x >= 0 \\ -1 & \text{if } x < 0 \end{cases}$

Step 3: Compute a) $\Delta w_i = \eta (t - o) x_i$

b) $w_i \leftarrow w_i + \Delta w_i$

Illustration – Perceptron learning

- We do only **ONE epoch**
- **Consider example-1**

1. $net = w^T x$

$= [1, -1, 0, 0.5] \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = [1 + 2 + 0 - 0.5]$

$= 2.5$

2. $o_1 = sign(2.5) = +1$

3. a) $\Delta w = \eta (t_1 - o_1) x_1$

$\Delta w = 0.1(-1-1) \cdot \begin{bmatrix} +1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = -0.2 \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}$

$\Delta w = \begin{bmatrix} -0.2 \\ 0.4 \\ 0 \\ 0.2 \end{bmatrix}$

b) $w \leftarrow w + \Delta w$

$= \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} -0.2 \\ 0.4 \\ 0 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$

Illustration – Perceptron learning

- **Consider example-2**

We have $w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$ $x^{(2)} = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}$ $t_2 = -1$

1. $net = w^T x$

$$= [0.8, -0.6, 0, 0.7] \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}$$

$$= [0 + (-0.9) + 0 - 0.7] = \underline{\underline{-1.6}}$$

2. $o_2 = sign(-1.6) = -1$

3. $\Delta w = \eta \underbrace{(t_2 - o_2)}_{0} x^{(2)} = \underline{\underline{0}}$

$\therefore$ No change in w.

$$w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}.$$

Illustration – Perceptron learning

- Consider Example-3

$w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$ $x^{(3)} = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$ $t_3 = 1$

Step 1: $net = w^T x$

$$= [0.8, -0.6, 0, 0.7] \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$$

$$= [-0.8 - 0.6 + 0 - 0.7]$$

$$= -2.1$$

Step 2: $o_3 = sign(net)$

$$= sign(-2.1) = \underline{\underline{-1}}$$

Step 3: a) $\Delta w = \eta (t_3 - o_3) x^{(3)}$

$$= 0.1(1-(-1)) \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$$

$$= 0.2 \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} = \begin{bmatrix} -0.2 \\ 0.2 \\ 0.1 \\ -0.2 \end{bmatrix}$$

b) $w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix} + \begin{bmatrix} -0.2 \\ 0.2 \\ 0.1 \\ -0.2 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.4 \\ 0.1 \\ 0.5 \end{bmatrix}$ Final updated weights after one epoch

Delta Rule

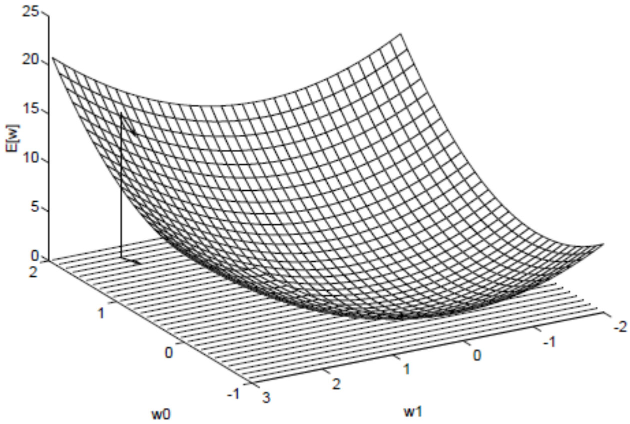
- The Delta Rule employs
  - the error function for what is known as Gradient Descent learning,
  - which involves the 'modification of weights along the most direct path in weight-space to minimize error'
  - so change applied to a given weight is proportional to the negative of the derivative of the error with respect to that weight

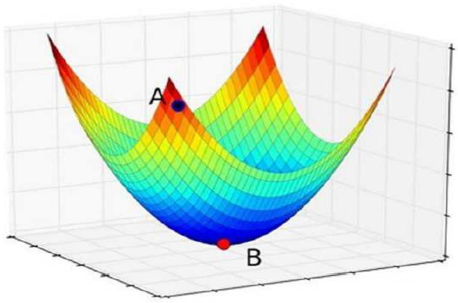
Delta Rule

- perceptron rule fails if data is not linearly separable
- delta rule converges toward a **best-fit approximation**
- uses **gradient descent** to search the hypothesis space
  - perceptron cannot be used, because it is not differentiable
  - hence, a **unthresholded linear unit** is appropriate
  - error measure: $E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$
- to understand gradient descent, it is helpful to visualize the entire hypothesis space with
  - all possible weight vectors and
  - associated E values

Error Surface

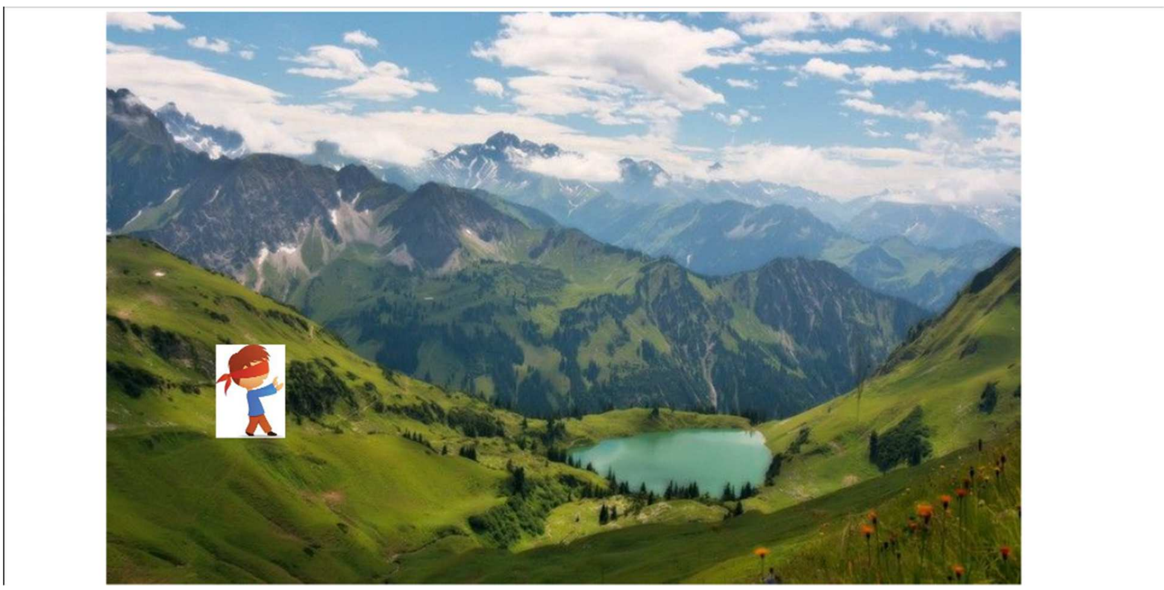
- the axes w_0, w_1 represent possible values for the two weights of a simple linear unit





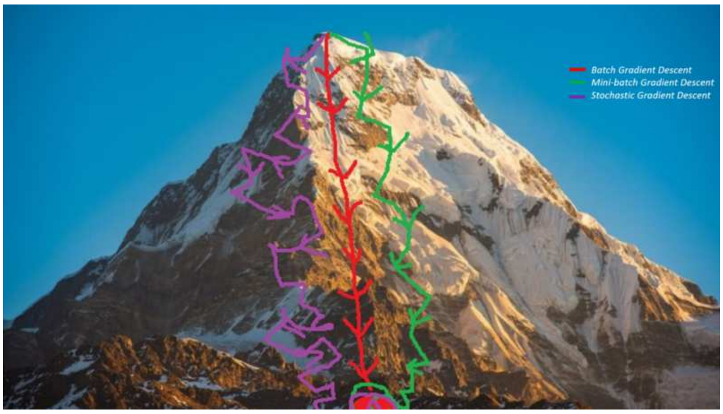
$\Rightarrow$ error surface must be **parabolic** with a **single global minimum**

Gradient Descent



Gradient Descent

MACHINE LEARNING



Gradient descent is an iterative optimization algorithm for finding the minimum of a function; in our case we want to minimize the error function.

To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient of the function at the current point.

Derivation of Gradient Descent

MACHINE LEARNING

- **problem:** How calculate the steepest descent along the error surface?
- derivation of E with respect to each component of $\vec{w}$
- this vector derivate is called *gradient* of E, written $\nabla E(\vec{w})$

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right]$$

- $\nabla E(\vec{w})$ specifies the steepest ascent, so $-\nabla E(\vec{w})$ specifies the steepest descent
- **training rule:** $w_i \leftarrow w_i + \Delta w_i$

Where,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Derivation of Gradient Descent

MACHINE LEARNING

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}$$

Therefore weight update rule for gradient descent is

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\ x_{id}$$

Gradient Descent

MACHINE LEARNING

- application difficulties of gradient descent
  - convergence may be quite slow
  - in case of many local minima, the global minimum may not be found
- **idea:** approximate gradient descent search by updating weights incrementally, following the calculation of the error for *each* individual example
- $\Delta w_i = \eta(t - o)x_i$ where $E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$
- **key differences:**
  - weights are not summed up over all examples before updating
  - requires less computation
  - better for avoidance of local minima

Gradient Descent Algorithm

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair of the form $<\vec{x}, t>$, where $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η is the learning rate.

- Initialize each w_i to some small random value
- Until the **termination condition** is met, Do
  - Initialize each Δw_i to zero
  - For each $<\vec{x}, t>$ in $training\_examples$, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do $\Delta w_i = \Delta w_i + \eta(t - o)x_i^*$
  - For each linear unit weight w_i, Do $w_i \leftarrow w_i + \Delta w_i^{**}$

To implement incremental approximation, equation ** is deleted and equation * is replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

Perceptron vs Delta rule

- **perceptron training rule:**
  - uses thresholded unit
  - converges after a finite number of iterations
  - output hypothesis classifies training data perfectly
  - linearly separability neccessary
- **delta rule:**
  - uses unthresholded linear unit
  - converges asymptotically toward a minimum error hypothesis
  - termination is not guaranteed
  - linear separability not neccessary

Perceptron vs Delta rule

- There are two differences between the perceptron and the delta rule.

1. The perceptron is based on an output from a step function, whereas the delta rule uses the linear combination of inputs directly.
2. The perceptron is guaranteed to converge to a consistent hypothesis assuming the data is linearly separable.

   The delta rules converges in the limit but it does not need the condition of linearly separable data.

Module 3- Outline

Artificial Neural Network

1. Biological Motivation
2. Neural Network Representation
3. Appropriate Problems for NN learning
4. Perceptions
5. **Multilayer Networks and Backpropagation Algorithm**
6. Remarks on Backpropagation Algorithm
7. Summary