MALINENI LAKSHMAIAH WOMEN'S ENGINEERING COLLEGE

(Approved by AICTE, Affiliated to JNTUK)(An ISO9001:2008 Certified Institution)

IV B.Tech (Common to CSE and IT), IV-I Semester, R19, Machine Learning Notes, UNIT-IV

Prepared by Dr.M.BHEEMALINGAIAH

Artificial Neural Networks and Support Vector Machine (SVM)

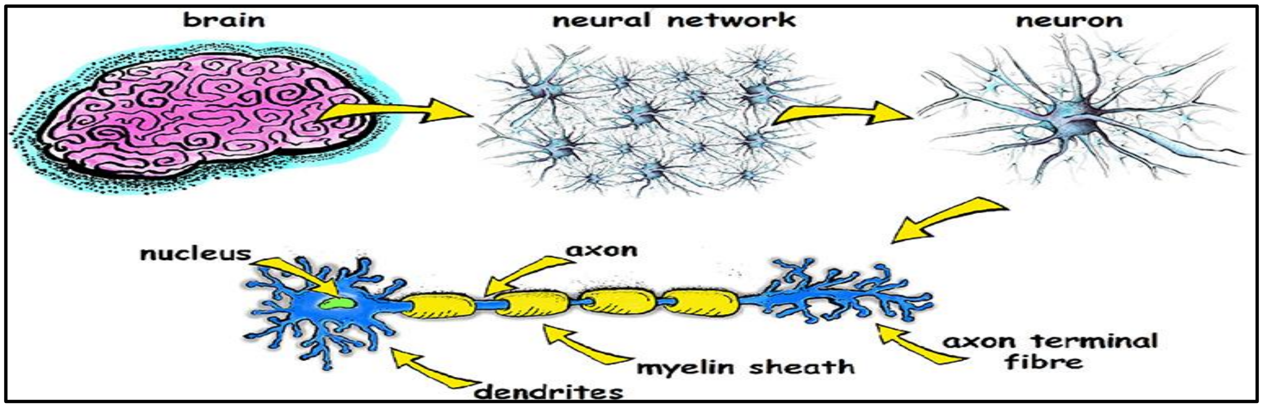
UNIT IV: Artificial Neural Networks: Neurons and biological motivation, Linear threshold units. Perceptrons: representational limitation and gradient descent training, Multilayer networks and backpropagation, Hidden layers and constructing intermediate, distributed representations. Overfitting, learning network structure, recurrent networks.
Support Vector Machines: Maximum margin linear separators. Quadratic programming solution to finding maximum margin separators. Kernels for learning non-linear functions.

Content

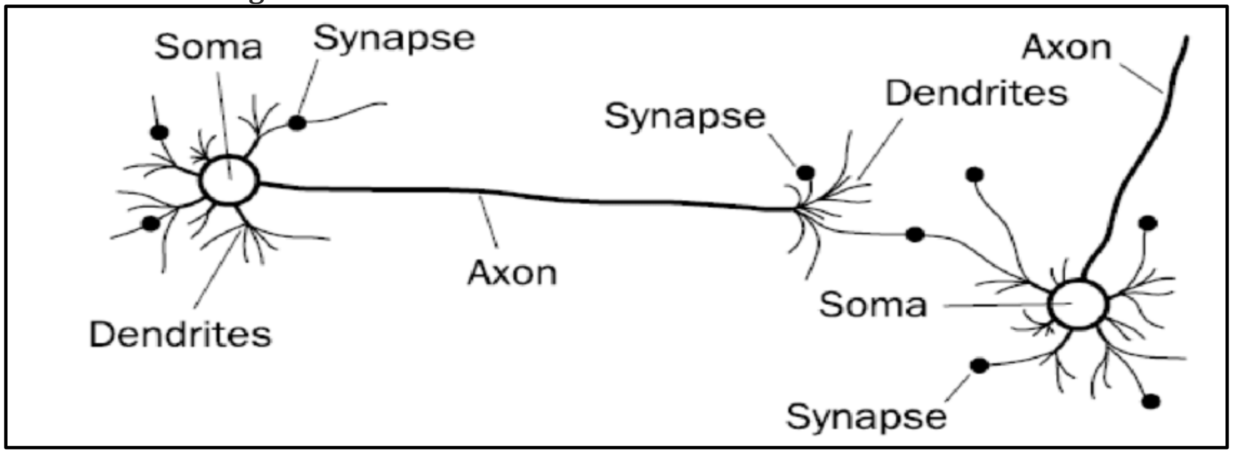
4.1 Introduction of Artificial neural network

What is Artificial Neural Network?

The term "**Artificial Neural Network**" is derived from Biological neural networks that develop the structure of a human brain. Similar to the human brain that has neurons interconnected to one another, artificial neural networks also have neurons that are interconnected to one another in various layers of the networks. These neurons are known as nodes.



Structure of Biological neurons and their functions



- **Dendrites** receive incoming signals.
- **Soma** (cell body) is responsible for processing the input and carries biochemical information.

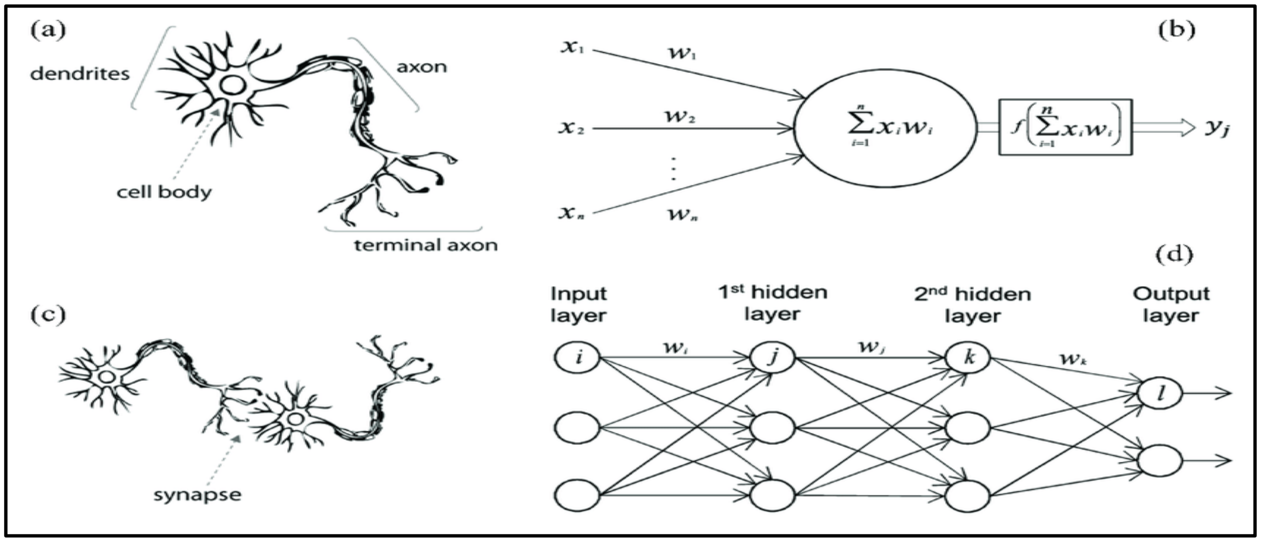
- **Axon** is tubular in structure responsible for the transmission of signals.
- **Synapse** is present at the end of the axon and is responsible for connecting other neurons.

Facts of Human Neurobiology

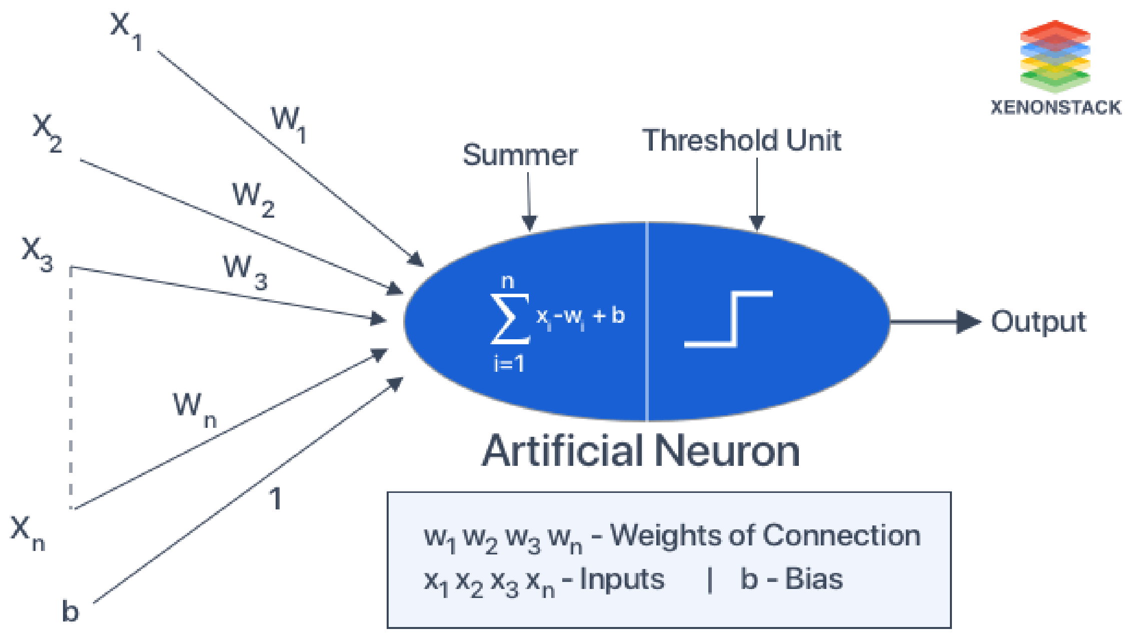
1. Number of neurons ~ 10^{11}
2. Connection per neuron ~ 10^{4-5}
3. Neuron switching time ~ 0.001 second or 10^{-3}
4. Scene recognition time ~ 0.1 second
5. 100 inference steps doesn't seem like enough
6. Highly parallel computation based on distributed representation

Biological Neural Network	**Artificial Neural Network**
Dendrites	Inputs
Cell nucleus	Nodes
Synapse	Weights
Axon	Output

Dendrites from Biological Neural Network represent inputs in Artificial Neural Networks, cell nucleus represents Nodes, synapse represents Weights, and Axon represents Output



How does Artificial Neural Network works?


XENONSTACK
X_1
X_2
X_3
X_n
b
W_1
W_2
W_3
W_n
1
Summer
Threshold Unit
$\sum_{i=1}^{n} x_i \cdot w_i + b$
Output
Artificial Neuron
$w_1\ w_2\ w_3\ w_n$ - Weights of Connection
$x_1\ x_2\ x_3\ x_n$ - Inputs | b - Bias

- Artificial Neural Networks can be viewed as weighted directed graphs in which artificial neurons are nodes, and directed edges with weights are connections between neuron outputs and neuron inputs.
- The Artificial Neural Network receives information from the external world in pattern and image in vector form. These inputs are designated by the notation x(n) for n number of inputs.
- Each input is multiplied by its corresponding weights. Weights are the information used by the neural network to solve a problem. Typically weight represents the strength of the interconnection between neurons inside the Neural Network.
- The weighted inputs are all summed up inside the computing unit (artificial neuron). In case the weighted sum is zero, bias is added to make the output not- zero or to scale up the system response. Bias has the weight and input always equal to '1'.
- The sum corresponds to any numerical value ranging from 0 to infinity. To limit the response to arrive at the desired value, the threshold value is set up. For this, the sum is forward through an activation function.
- The activation function is set to the transfer function to get the desired output. There are linear as well as the nonlinear activation function

4.2 When to consider Artificial Neural Networks?

Properties of Artificial Neural Networks

- 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)

APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN learning is well-suited to problems in which the training data corresponds to noisy,complex sensor data, such as inputs from cameras and microphones.

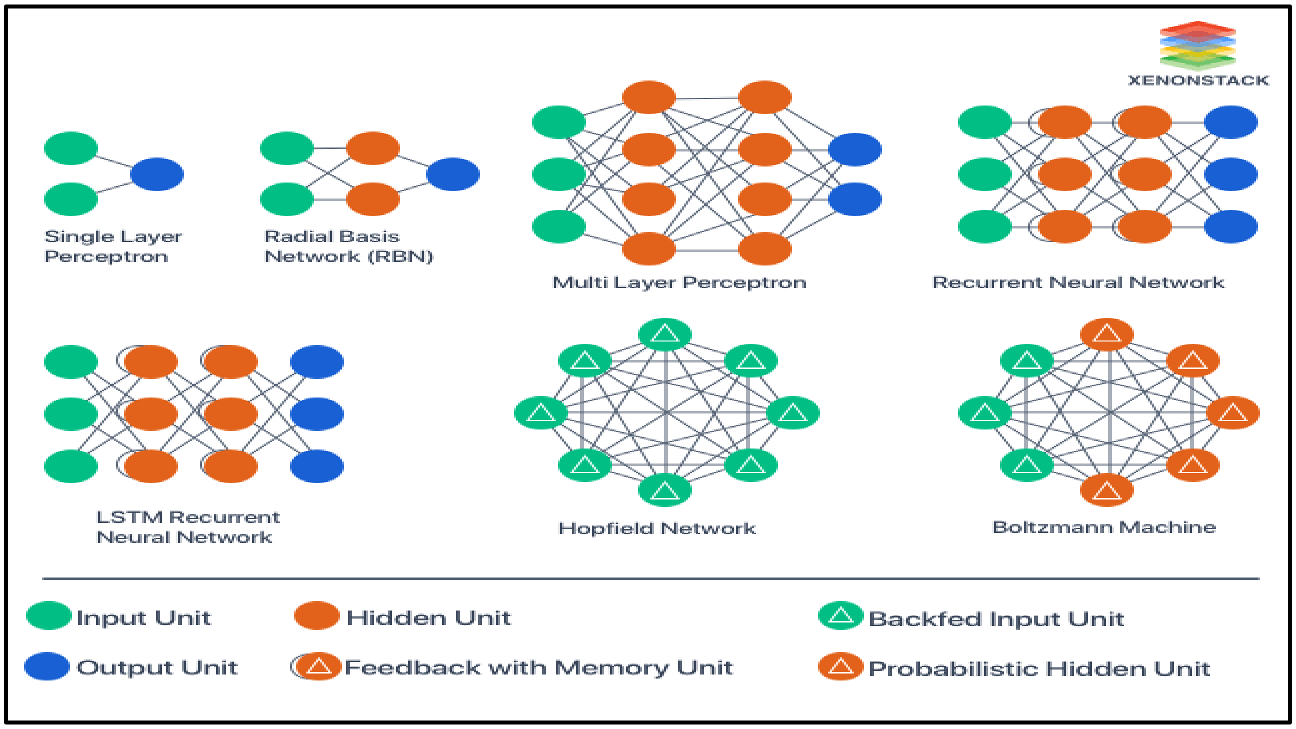
ANN is appropriate for problems with the following characteristics:

1. Instances are represented by many attribute-value pairs.
2. The target function output may be discrete-valued, real-valued, or a vector of severalreal- or discrete-valued attributes.
3. The training examples may contain errors.
4. Long training times are acceptable.
5. Fast evaluation of the learned target function may be required
6. The ability of humans to understand the learned target function is not important

4.3 Basic fundamental of Artificial Neural Networks

4.3.1 Neural Network Architecture Types

- Perceptron Model in Neural Networks
- Radial Basis Function Neural Network
- Multilayer Perceptron Neural Network
- Recurrent Neural Network
- Long Short-Term Memory Neural Network (LSTM)
- Hopfield Network
- Boltzmann Machine Neural Network
- Convolutional Neural Network
- Modular Neural Network
- Physical Neural Network



Perceptron Model in Neural Networks: Neural Network is having two input units and one output unit with no hidden layers. These are also known as ‘single-layer perceptrons.'

Radial Basis Function Neural Network: These networks are similar to the feed-forward Neural Network, except radial basis function is used as these neurons' activation function.

Multilayer Perceptron Neural Network: These networks use more than one hidden layer of neurons, unlike single-layer perceptron. These are also known as Deep Feedforward Neural Networks.

Recurrent Neural Network: Type of Neural Network in which hidden layer neurons have self-connections. Recurrent Neural Networks possess memory. At any instance, the hidden layer neuron receives activation from the lower layer and its previous activation value.

Long Short-Term Memory Neural Network (LSTM): The type of Neural Network in which memory cell is incorporated into hidden layer neurons is called LSTM network.

Hopfield Network: A fully interconnected network of neurons in which each neuron is connected to every other neuron. The network is trained with input patterns by setting a value of neurons to the desired pattern. Then its weights are computed. The weights are not changed. Once trained for one or more patterns, the network will converge to the learned patterns. It is different from other Neural Networks.

Boltzmann Machine Neural Network: These networks are similar to the Hopfield network, except some neurons are input, while others are hidden in nature. The weights are initialized randomly and learn through the backpropagation algorithm.

Convolutional Neural Network: Get a complete overview of Convolutional Neural Networks through our blog Log Analytics with Machine Learning and Deep Learning.

Modular Neural Network: It is the combined structure of different types of neural networks like multilayer perceptron, Hopfield Network, Recurrent Neural Network, etc., which are incorporated as a single module into the network to perform independent subtask of whole complete Neural Networks.

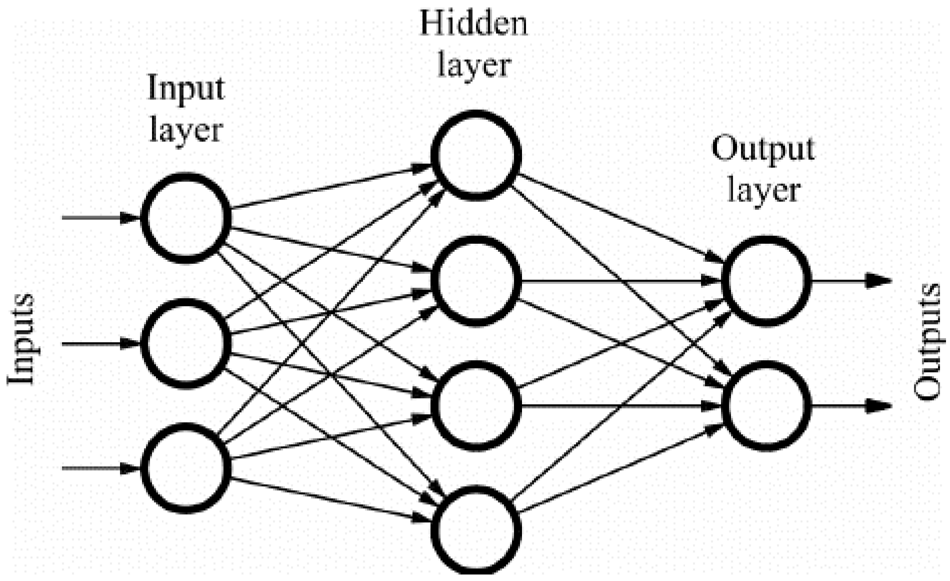
Physical Neural Network: In this type of Artificial Neural Network, electrically adjustable resistance material is used to emulate synapse instead of software simulations performed in the neural network. Artificial Intelligence collects and analyze data using smart sensors or machine learning algorithms and automatically route service requests to reduce the human workload.

4.3.2 What are the Five Algorithms to Train a Neural Network?

- Hebbian Learning Rule
- Self - Organizing Kohonen Rule
- Hopfield Network Law
- LMS algorithm (Least Mean Square)
- Competitive Learning

4.3.3 What is the architecture of Artificial Neural Network?

A typical Neural Network contains a large number of artificial neurons called units arranged in a series of layers. In typical Artificial Neural Network comprise different layers –



- **Input layer** - It contains those units (Artificial Neurons) which receive input from the outside world on which the network will learn, recognize about, or otherwise process.

′

- **Output layer** - It contains units that respond to the information about how it learn any task.

- **Hidden layer** - These units are in between input and output layers. The hidden layer's job is to transform the input into something that the output unit can use somehow.

  Connect Neural Networks, which means say each hidden neuron links completely to every neuron in its previous layer (input) and the next layer (output) layer.

4.3.4 What are the Learning Techniques in Neural Networks?

Here is a list of Learning Techniques

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning
- Offline Learning
- Online Learning

Supervised Learning: In this learning, the training data is input to the network, and the desired output is known weights are adjusted until production yields desired value.

Unsupervised Learning: Use the input data to train the network whose output is known. The network classifies the input data and adjusts the weight by feature extraction in input data.

Reinforcement Learning: Here, the output value is unknown, but the network provides feedback on whether the output is right or wrong. It is Semi-Supervised Learning.

Offline Learning: The weight vector adjustment and threshold adjustment are made only after the training set is shown to the network. It is also called Batch Learning.

Online Learning: The adjustment of the weight and threshold is made after presenting each training sample to the network.

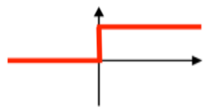
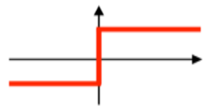
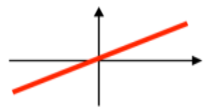
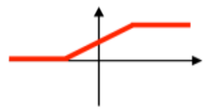
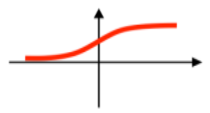
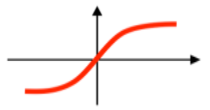
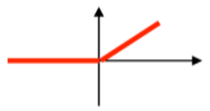
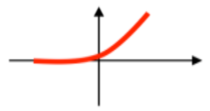
4.3.5 What are the applications of neural networks?

Top applications of neural networks

- Neural Networks for Pattern Recognition
- Neural Network for Machine Learning
- Face Recognition using Artificial Neural Networks
- Neuro-Fuzzy Model and its applications
- Neural Networks for data-intensive applications

4.4 Activation Functions in Artificial Neural Network

Some of well-known activation functions are used in ANN. They are shown in following table

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus Copyright © Sebastian Raschka 2016 (http://sebastianraschka.com)	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

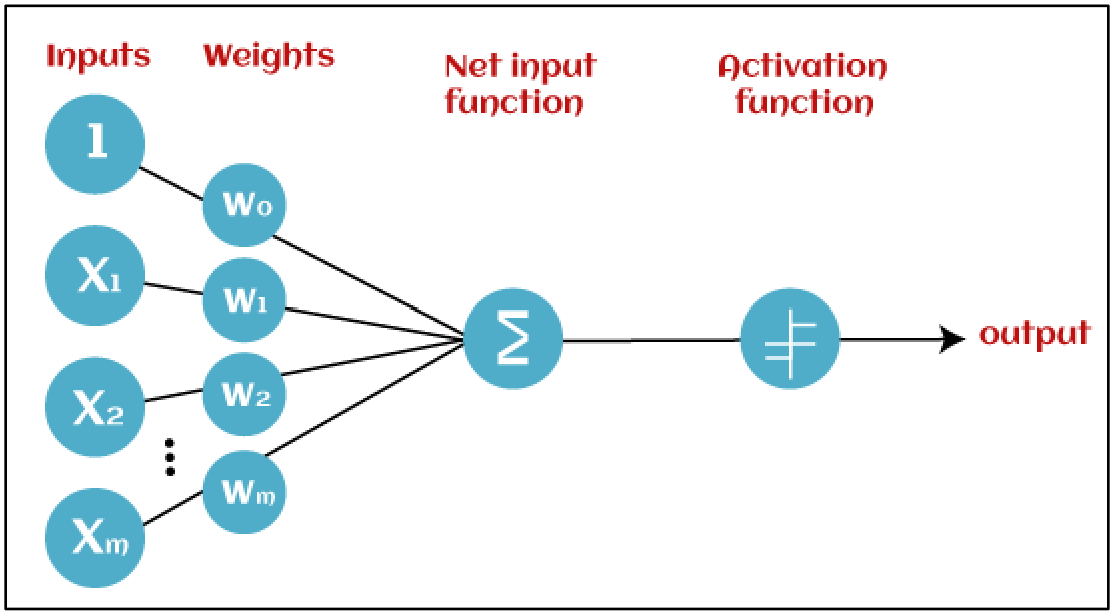
4.5 PERCEPTRON (Single Neuron Network)

- . **Perceptron is a building block of an Artificial Neural Network**. Initially, in the mid of 19th century, **Mr. Frank Rosenblatt** invented the Perceptron for performing certain calculations to detect input data capabilities or business intelligence. Perceptron is a linear Machine Learning algorithm used for supervised learning for various binary classifiers. This algorithm enables neurons to learn elements and processes them one by one during preparation

- One type of ANN system is based on a unit called a perceptron. Perceptron is a singlelayer neural network.

Basic Components of Perceptron

Mr. Frank Rosenblatt invented the perceptron model as a binary classifier which contains three main components. These are as follows:



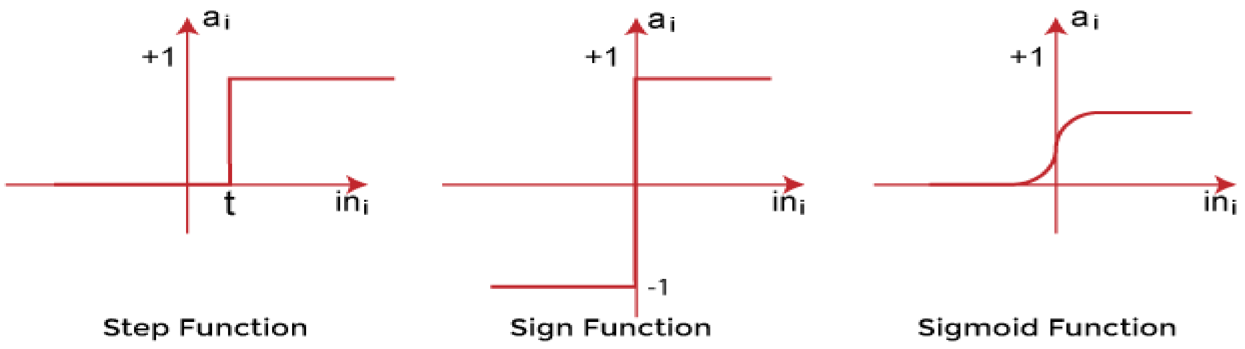
Input Nodes or Input Layer: This is the primary component of Perceptron which accepts the initial data into the system for further processing. Each input node contains a real numerical value.

Wight and Bias: Weight parameter represents the strength of the connection between units. This is another most important parameter of Perceptron components. Weight is directly proportional to the strength of the associated input neuron in deciding the output. Further, Bias can be considered as the line of intercept in a linear equation.

Activation Function: These are the final and important components that help to determine whether the neuron will fire or not. Activation Function can be considered primarily as a step function.

Types of Activation functions:

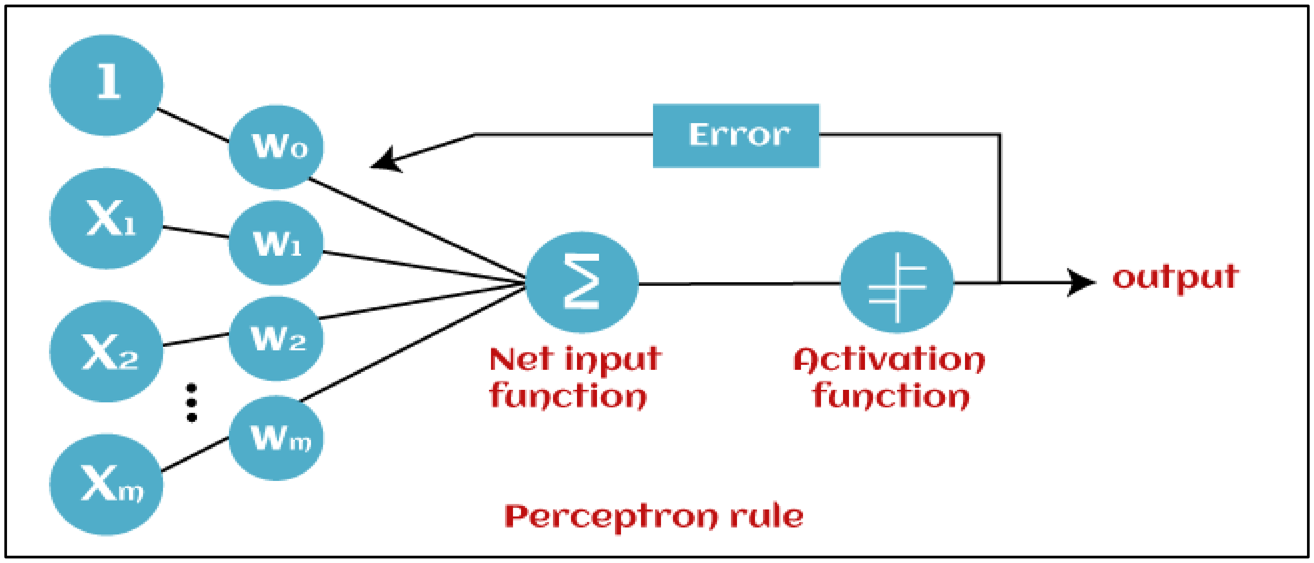
- Sign function
- Step function, and
- Sigmoid function



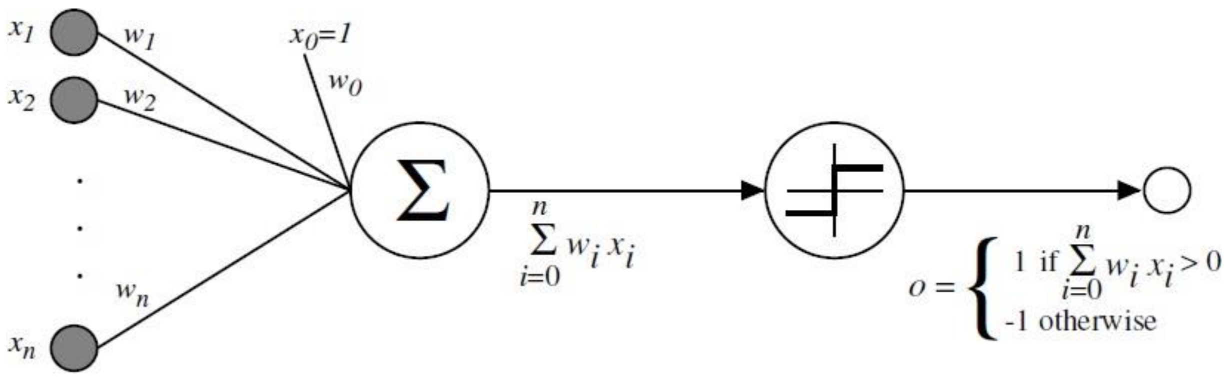
The data scientist uses the activation function to take a subjective decision based on various problem statements and forms the desired outputs. Activation function may differ (e.g., Sign, Step, and Sigmoid) in perceptron models by checking whether the learning process is slow or has vanishing or exploding gradients.

How does Perceptron work?

In Machine Learning, Perceptron is considered as a single-layer neural network that consists of four main parameters named input values (Input nodes), weights and Bias, net sum, and an activation function. The perceptron model begins with the multiplication of all input values and their weights, then adds these values together to create the weighted sum. Then this weighted sum is applied to the activation function 'f' to obtain the desired output. This activation function is also known as the **step function** and is represented by **'f'**.



This step function or Activation function plays a vital role in ensuring that output is mapped between required values (0,1) or (-1,1). It is important to note that the weight of input is indicative of the strength of a node. Similarly, an input's bias value gives the ability to shift the activation function curve up or down.



x_1
w_1
x_2
w_2
$x_0=1$
w_0
w_n
x_n
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

Figure: A perceptron

- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise.
- Given inputs ***x*** through ***x,*** the output **O*(x1,*** . . . , ***xn)*** computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1x_1 + \cdots + w_nx_n > 0 \\ -1 \text{ otherwise.} \end{cases}$$

- Where, each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output.
- w_0 is a threshold that the weighted combination of inputs $w_1 * x_1 + \ldots + w_n * x_n$ must surpass in order for the perceptron to output a 1.

Sometimes, the perceptron function is written as,

$$O(\vec{x}) = \text{sgn}(\vec{w} \cdot \vec{x})$$

Where,

$$\text{sgn}(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise.} \end{cases}$$

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$. Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all

$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

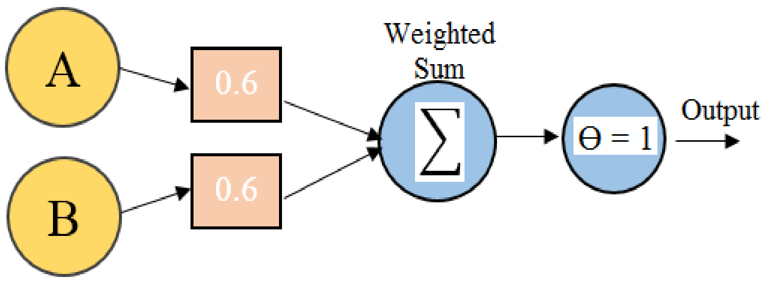
possible real-valued weight vectors

Representational Power of Perceptrons

- The perceptron can be viewed as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points)
- The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputsa -1 for instances lying on the other side, as illustrated in below figure

Example: Representation of AND functions

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.

Drawback of perceptron

- The perceptron rule finds a successful weight vector when the training examples arelinearly separable, it can fail to converge if the examples are not linearly separable

The Perceptron Training Rule

The learning problem is to determine a weight vector that causes the perceptron to produce thecorrect + 1 or - 1 output for each of the given training examples.

To learn an acceptable weight vector

- Begin with random weights, then iteratively apply the perceptron to each trainingexample, modifying the perceptron weights whenever it misclassifies an example.

- This process is repeated, iterating through the training examples as many times asneeded until the perceptron classifies all training examples correctly.
- Weights are modified at each step according to the perceptron training rule, whichrevises the weight wi associated with input xi according to the rule.

$$w_i \leftarrow w_i + \Delta w_i$$

Where,

$$\Delta w_i = \eta(t - o)x_i$$

Here,
t is the target output for the current training example
o is the output generated by the perceptron
η is a positive constant called the ***learning rate***

- The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decayas the number of weight-tuning iterations increases

Drawback:

The perceptron rule finds a successful weight vector when the training examples are linearlyseparable, it can fail to converge if the examples are not linearly separable

4.5.1 Gradient Descent and the Delta Rule

Gradient Descent is an optimization algorithm used for minimizing the cost function in various machine learning algorithms. It is basically used for updating the parameters of the learning model.

Types of gradient Descent:

- **Batch Gradient Descent(Gradient Descent)** : This is a type of gradient descent which processes all the training examples for each iteration of gradient descent. But if the number of training examples is large, then batch gradient descent is computationally very expensive. Hence if the number of training examples is large, then batch gradient descent is not preferred. Instead, we prefer to use stochastic gradient descent or mini-batch gradient descent.
- **Stochastic Gradient Descent**: This is a type of gradient descent which processes 1 training example per iteration. Hence, the parameters are being updated even after one

iteration in which only a single example has been processed. Hence this is quite faster than batch gradient descent. But again, when the number of training examples is large, even then it processes only one example which can be additional overhead for the system as the number of iterations will be quite large.

- **Mini Batch gradient descent**: This is a type of gradient descent which works faster than both batch gradient descent and stochastic gradient descent. Here *b* examples where *b<m* are processed per iteration. So even if the number of training examples is large, it is processed in batches of b training examples in one go. Thus, it works for larger training examples and that too with lesser number of iterations.

Batch Gradient Descent(Gradient Descent)

- If the training examples are not linearly separable, the delta rule converges toward abest-fit approximation to the target concept.
- The key idea behind the delta rule is to use **gradient descent** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

To understand the delta training rule, consider the task of training an unthresholded perceptron. That is, a linear unit for which the output ***O*** is given by

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

$$O(\vec{x}) = (\vec{w} . \vec{x}) \qquad \text{equ. (1)}$$

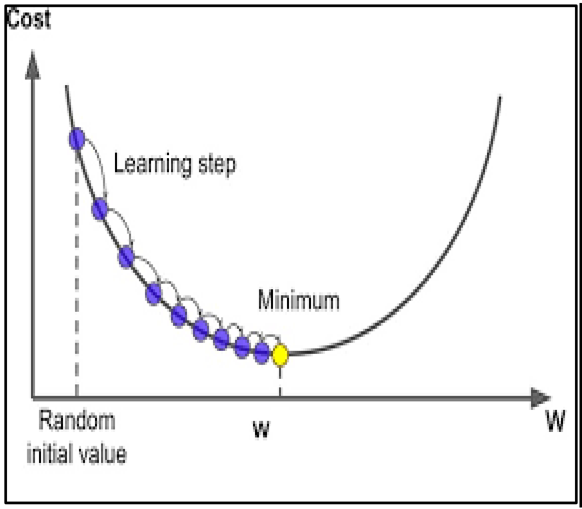
To derive a weight learning rule for linear units, specify a measure for the ***training error*** of ahypothesis (weight vector), relative to the training examples.

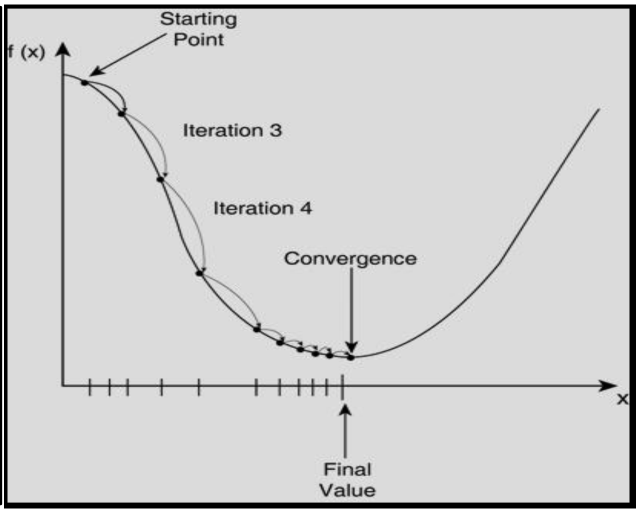
$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \qquad \text{equ. (2)}$$

Where,

- D is the set of training examples,
- t_d is the target output for training example d,
- o_d is the output of the linear unit for training example d
- E ($\vec{w}$) simply half the squared difference between the target output t_d and the linear unit output o_d, summed over all training examples.

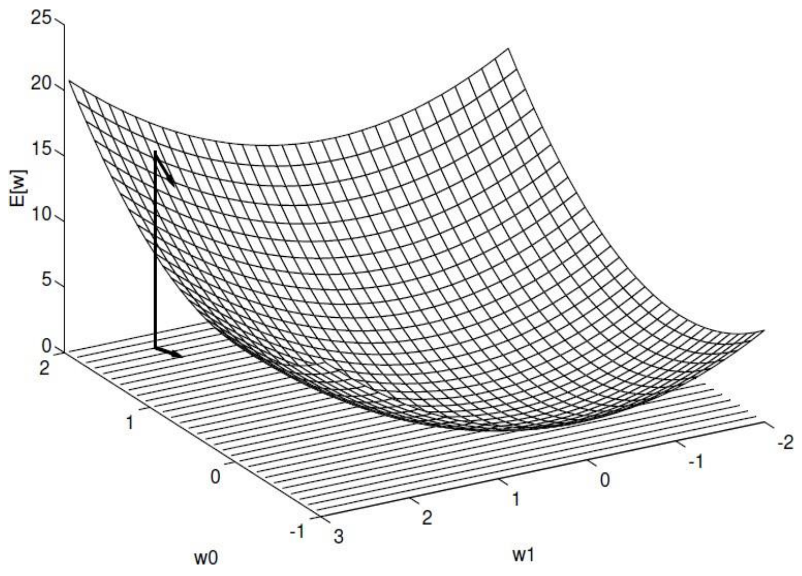
Visualizing the Hypothesis Space





- To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values as shown in below figure.
- Here the axes w0 and wl represent possible values for the two weights of a simple linearunit. The w0, wl plane therefore represents the entire hypothesis space.
- The vertical axis indicates the error E relative to some fixed set of training examples.
- The arrow shows the negated gradient at one particular point, indicating the direction inthe w0, wl plane producing steepest descent along the error surface.

- The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space



.

- Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a single global minimum.

Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps.

At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in above figure. This process continues until the global minimum error is reached.

4.5.2 Derivation of the Gradient Descent Rule

How to calculate the direction of steepest descent along the error surface?

The direction of steepest can be found by computing the derivative of E with respect to each component of the vector $\overset{u}{w}$. This vector derivative is called the gradient of E with respect to $\overset{u}{w}$, is written as

$$\boxed{\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right] \qquad \text{equ. (3)}}$$

The gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

Where,

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \qquad \text{equ. (4)}$$

- Here η is a positive constant called the learning rate, which determines the stepsize in the gradient descent search.
- The negative sign is present because we want to move the weight vector in thedirection that decreases E.

This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$

Where,

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} \qquad \text{equ. (5)}$$

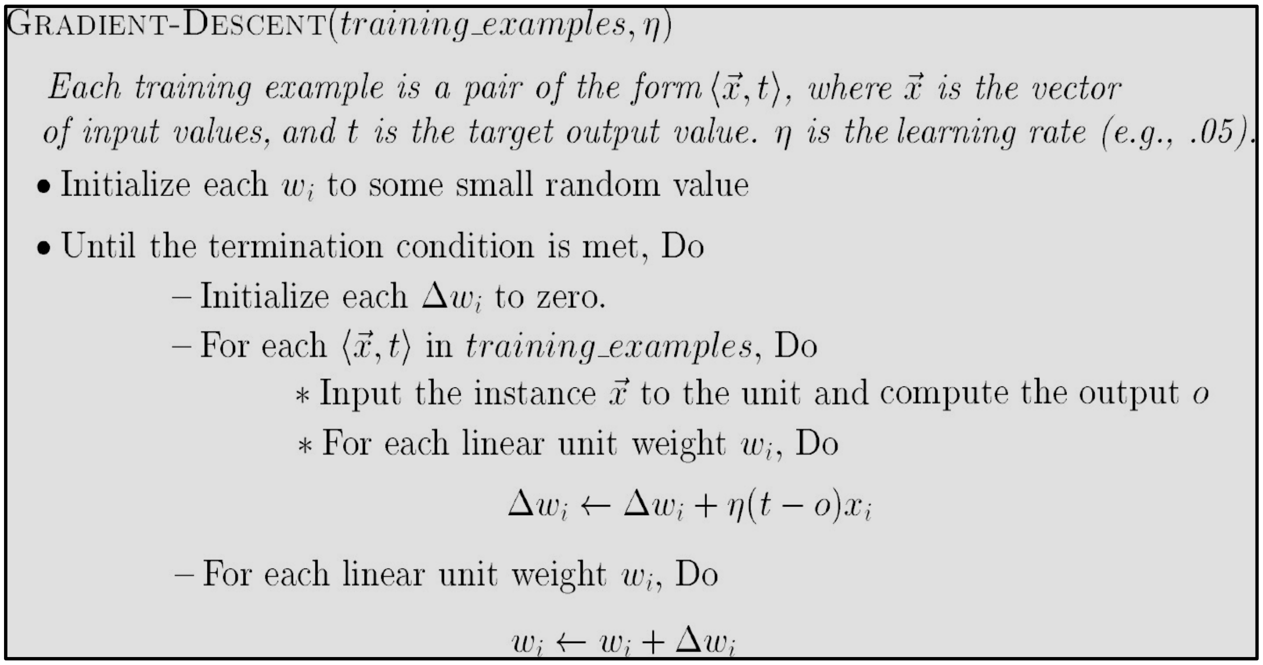
Calculate the gradient at each step. The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating E from Equation (2), as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d}) \qquad \text{equ. (6)}
\end{aligned}$$

Substituting Equation (6) into Equation (5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id} \qquad \text{equ. (7)}$$

4.5.3 Gradient Descent Algorithm



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

  Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector
  of input values, and $t$ is the target output value. $\eta$ is the learning rate (e.g., .05).
  • Initialize each $w_i$ to some small random value
  • Until the termination condition is met, Do
        – Initialize each $\Delta w_i$ to zero.
        – For each $\langle \vec{x}, t \rangle$ 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
                    $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
        – For each linear unit weight $w_i$, Do
                    $w_i \leftarrow w_i + \Delta w_i$
```

Issues in Gradient Descent Algorithm

Gradient descent is an important general paradigm for learning. It is a strategy for searchingthrough a large or infinite hypothesis space that can be applied whenever

1. The hypothesis space contains continuously parameterized hypotheses
2. The error can be differentiated with respect to these hypothesis parameters

The key practical difficulties in applying gradient descent are

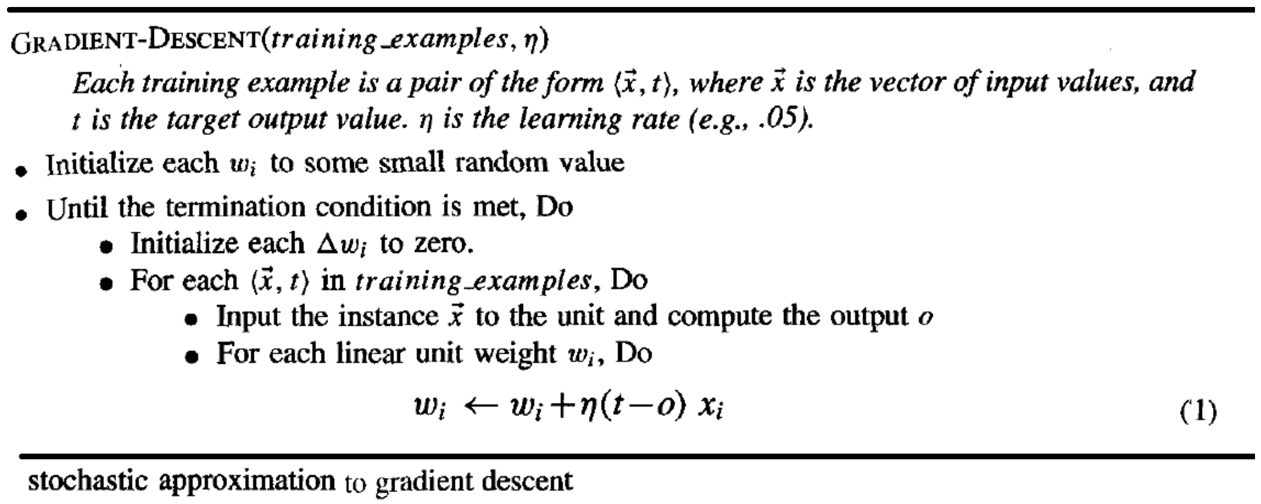
1. Converging to a local minimum can sometimes be quite slow
2. If there are multiple local minima in the error surface, then there is no guarantee thatthe procedure will find the global minimum

4.5.4 Stochastic Approximation to Gradient Descent

- The gradient descent training rule presented in Equation (7) computes weight updates after summing over all the training examples in D
- The idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for each individual example

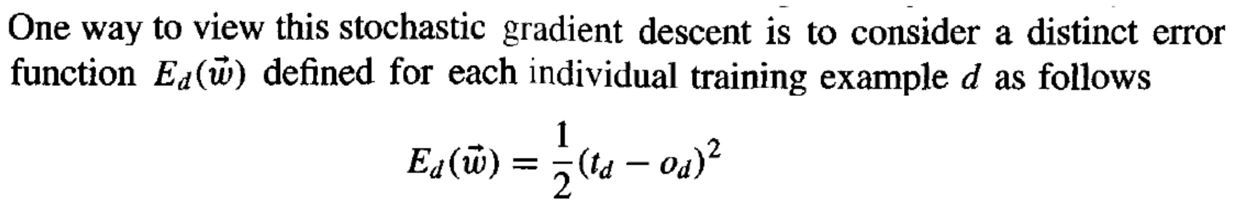
$$\Delta wi = \eta\ (t - o)\ xi$$

- where t, o, and x_i are the target value, unit output, and ith input for the training examplein question



```
GRADIENT-DESCENT(training_examples, $\eta$)
    Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and
    $t$ is the target output value. $\eta$ is the learning rate (e.g., .05).
• Initialize each $w_i$ to some small random value
• Until the termination condition is met, Do
    • Initialize each $\Delta w_i$ to zero.
    • For each $\langle \vec{x}, t \rangle$ 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
                $w_i \leftarrow w_i + \eta(t - o)\, x_i$        (1)
```

stochastic approximation to gradient descent



One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ defined for each individual training example d as follows

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

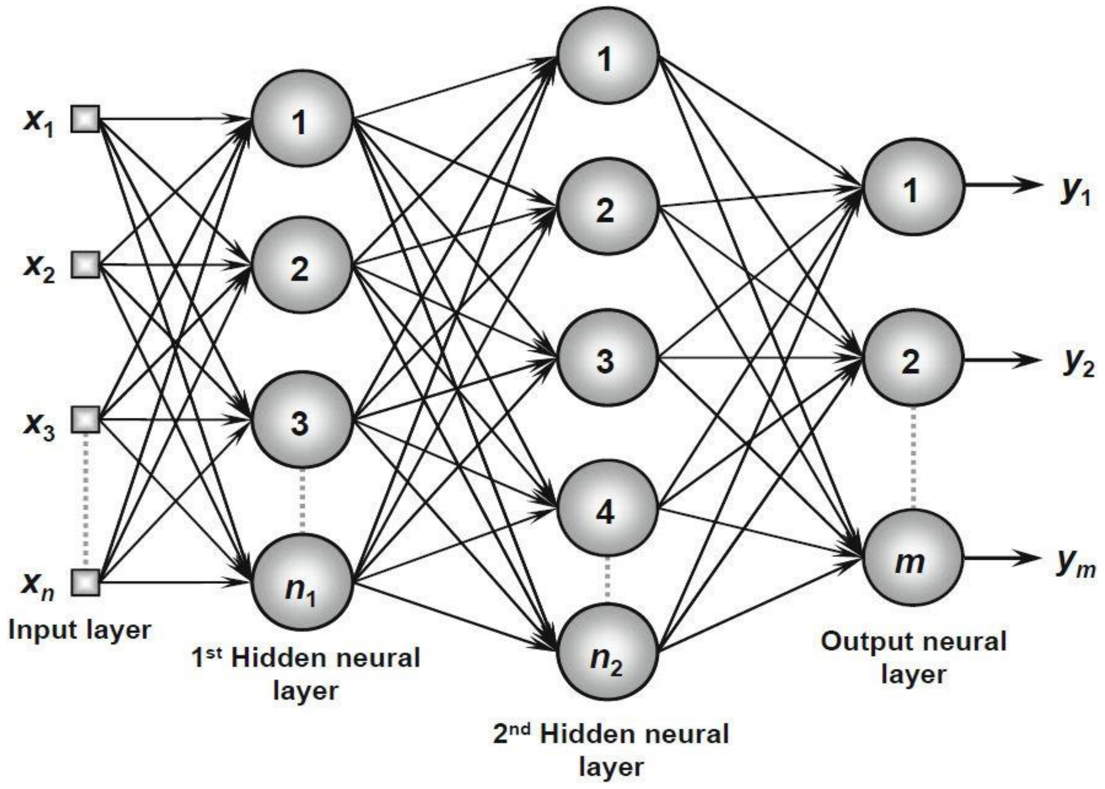
- One way to view this stochastic gradient descent is to consider a distinct error function for each individual training example d **as** follows
- Where, t_d and o_d are the target value and the unit output value for training example d.
- Stochastic gradient descent iterates over the training examples ***d*** in D, at each iteration altering the weights according to the gradient with respect to
- The sequence of these weight updates, when iterated over all training examples, provides a reasonable approximation to descending the gradient with respect to our original error function
- By making the value of **η** sufficiently small, stochastic gradient descent can be made to approximate true gradient descent arbitrarily closely

The key differences between standard gradient descent and stochastic gradient descent are

- In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.
- In cases where there are multiple local minima with respect to stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $E_d(\vec{w})$ rather than $E(\vec{w})$ to guide its search.

4.6 Multilayer Networks (Multi-Layer Perceptron Network)

The main shortcoming of the Feed Forward networks was its inability to learn with backpropagation (Unsupervised Learning). Multi-layer Perceptrons are the neural networks which incorporate multiple hidden layers and activation functions. The learning takes place in a Supervised manner where the weights are updated by the means of Gradient Descent. Multi-layer Perceptron is bi-directional, i.e., Forward propagation of the inputs, and the backward propagation of the weight updates. The activation functions can be changes with respect to the type of target. Softmax is usually used for multi-class classification, Sigmoid for binary classification and so on. These are also called dense networks because all the neurons in a layer are connected to all the neurons in the next layer as shown in figure



Multilayer networks learned by the **BACKPROPACATION** algorithm are capable of expressing a rich variety of nonlinear decision surfaces.

A Differentiable Threshold Unit (Sigmoid unit)

Sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiablethreshold function.

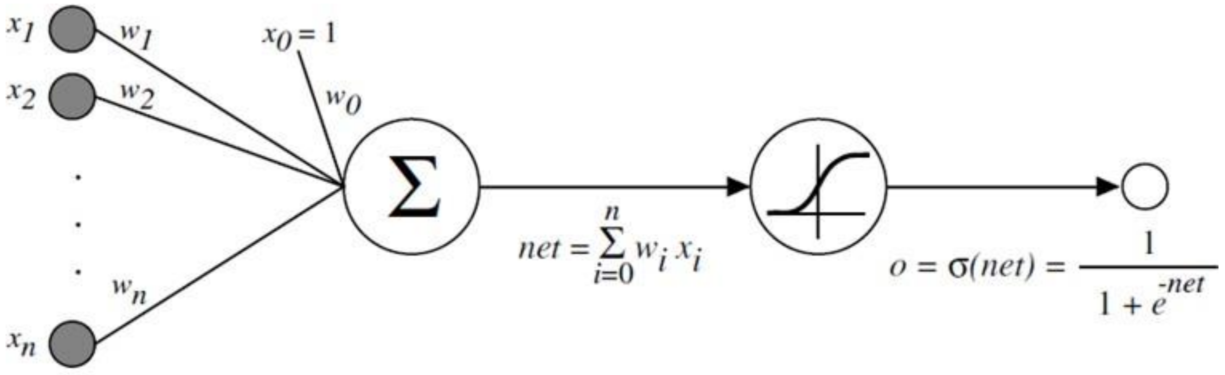

x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1+e^{-net}}$

Figure: A Sigmoid Threshold Unit

- The sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result and the threshold output is a continuous function of its input.
- More precisely, the sigmoid unit computes its output O as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

Where,

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

σ is the sigmoid function

4. 6 .1 Backpropagation Algorithm

Backpropagation is an algorithm that back propagates the errors from output nodes to the input nodes. Therefore, it is simply referred to as backward propagation of errors. It uses in the vast applications of neural networks like Character recognition, Signature verification, etc.

Backpropagation is a widely used algorithm for training feedforward neural networks. It computes the gradient of the loss function with respect to the network weights and is very efficient, rather than naively directly computing the gradient with respect to each individual weight. This efficiency makes it possible to use gradient methods to train multi-layer networks and update weights to minimize loss; variants such as gradient descent or stochastic gradient descent are often used.

The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight via the chain rule, computing the gradient layer by layer, and iterating backward from the last layer to avoid redundant computation of intermediate terms in the chain rule

Working of Backpropagation:
Neural networks use supervised learning to generate output vectors from input vectors that the network operates on. It Compares generated output to the desired output and generates an error report if the result does not match the generated output vector. Then it adjusts the weights according to the bug report to get your desired output.

Backpropagation Algorithm:

Step 1: Inputs X, arrive through the preconnected path.

Step 2: The input is modeled using true weights W. Weights are usually chosen randomly.

Step 3: Calculate the output of each neuron from the input layer to the hidden layer to the output layer.

Step 4: Calculate the error in the outputs

Backpropagation Error= Actual Output – Desired Output

Step 5: From the output layer, go back to the hidden layer to adjust the weights to reduce the error.

Step 6: Repeat the process until the desired output is achieved.

Need for Backpropagation: Backpropagation is “backpropagation of errors” and is very useful for training neural networks. It’s fast, easy to implement, and simple. Backpropagation does not require any parameters to be set, except the number of inputs. Backpropagation is a flexible method because no prior knowledge of the network is required.

Types of Backpropagation: There are two types of backpropagation networks.

- **Static backpropagation:** Static backpropagation is a network designed to map static inputs for static outputs. These types of networks are capable of solving static classification problems such as OCR (Optical Character Recognition).
-
- **Recurrent backpropagation:** Recursive backpropagation is another network used for fixed-point learning. Activation in recurrent backpropagation is feed-forward until a fixed

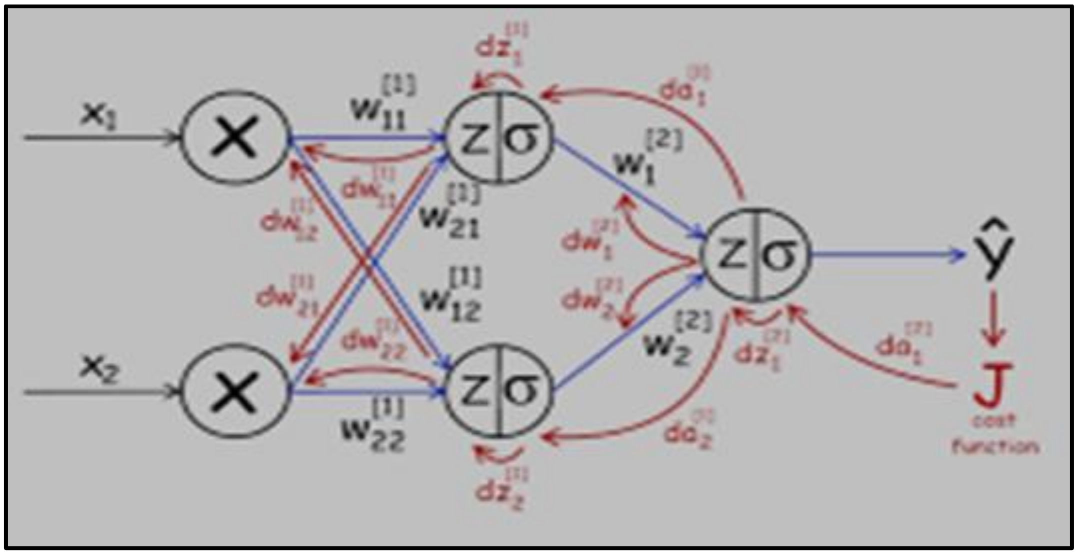
value is reached. Static backpropagation provides an instant mapping, while recurrent backpropagation does not provide an instant mapping.

Advantages:

- It is simple, fast, and easy to program.
- Only numbers of the input are tuned, not any other parameter.
- It is Flexible and efficient.
- No need for users to learn any special functions.

Disadvantages:

- It is sensitive to noisy data and irregularities. Noisy data can lead to inaccurate results.
- Performance is highly dependent on input data.
- Spending too much time training.
- The matrix-based approach is preferred over a mini-batch



- The BACKPROPAGATION Algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.
- In BACKPROPAGATION algorithm, we consider networks with multiple output unitsrather than single units as before, so we redefine E to sum the errors over all of the network output units.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \quad \text{........equ. (1)}$$

where,

- **outputs** - is the set of output units in the network

- **tkd** and **Okd** - the target and output values associated with the **kth** output unit
- **d** - training example

BACKPROPAGATION Algorithm:

BACKPROPAGATION (*training_example, η, n_{in}, n_{out}, n_{hidden}*)

- *Each training example is a pair of the form* $\left(\vec{x}, \vec{t}\right)$ *where*
- $\left(\vec{x}\right)$ *is the vector of network input values,*
- $\left(\vec{t}\right)$ *and is the vector of target network output values.*
- *η is the learning rate (e.g., .05).*
- *n_i, is the number of network inputs,*
- *n_{hidden} the numberof units in the hidden layer, and n_{out} the number of output units.*
- *The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is*
- *denoted by w_{ji}*

- Create a feed-forward network with n_i inputs, n_{hidden} hidden units, and n_{out} outputunits.
- Initialize all network weights to small random numbers
- Until the termination condition is met, Do
  - For each $\left(\vec{x}, \vec{t}\right)$ in training examples, Do

    Propagate the input forward through the network:

    1. Input the instance $\left(\vec{x}\right)$, to the network and compute the output Ou of every unit u in the network.

    Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

4. Update each network weight w_{ji}

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

Where

$$\Delta w_{ji} = \eta \delta_j x_{i,j}$$

Adding Momentum

Because BACKPROPAGATION is such a widely used algorithm, many variations have beendeveloped. The most common is to alter the weight-update rule the equation below

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

by making the weight update on the nth iteration depend partially on the update that occurredduring the (n - 1)th iteration, as follows:

$$\Delta w_{ji}(n) = \eta \, \delta_j \, x_{ji} + \alpha \Delta w_{ji}(n-1)$$

4.6.2 Derivation of the Backpropagation Rule

- Deriving the stochastic gradient descent rule: Stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error E_d with respect to this single example
- For each training example d every weight w_{ji} is updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \qquad \text{.........equ. (1)}$$

where, E_d is the error on training example d, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in output} (t_k - o_k)^2$$

Here outputs is the set of output units in the network, ***tk*** is the target value of unit ***k*** for trainingexample ***d***, and ***ok*** is the output of unit ***k*** given training example ***d***.

The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \Sigma_i w_{ji}x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- outputs = the set of units in the final layer of the network
- Downstream(j) = the set of units whose immediate inputs include the output of unit j

derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the stochastic gradient descent rule

seen in Equation $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$

notice that weight w_{ji} can influence the rest of the network only through net_j.

Use chain rule to write

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

$$= \frac{\partial E_d}{\partial net_j} x_{ji} \qquad \text{......equ(2)}$$

Derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$

<u>Consider two cases:</u> The case where unit j is an output unit for the network, and the case wherej is an internal unit (hidden unit).

Case 1: Training Rule for Output Unit Weights.

wji can influence the rest of the network only through netj , netj can influence the network only through oj. Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j} \qquad \text{.....equ}(3)$$

To begin, consider just the first term in Equation (3)

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k\in outputs}(t_k - o_k)^2$$

The derivatives $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all output units k except when $k = j$. We therefore drop the summation over output units and simply set $k = j$.

$$\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2\\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j}\\ &= -(t_j - o_j) \qquad \text{......equ}(4)\end{aligned}$$

Next consider the second term in Equation (3). Since $o_j = \sigma(net_j)$, the derivative $\frac{\partial o_j}{\partial net_j}$ is just the derivative of the sigmoid function, which we have already noted is equal to $\sigma(net_j)(1 - \sigma(net_j))$. Therefore,

$$\begin{aligned}\frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j}\\ &= o_j(1 - o_j) \qquad \text{.......equ}(5)\end{aligned}$$

Substituting expressions (4) and (5) into (3), we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\ o_j(1 - o_j) \qquad \text{.......equ}(6)$$

and combining this with Equations (1) and (2), we have the stochastic gradient descent rule for output units

$$\Delta w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1 - o_j)x_{ji} \qquad \text{.......equ}(7)$$

Case 2: Training Rule for Hidden Unit Weights.

- In the case where j is an internal, or hidden unit in the network, the derivation of the training rule for wji must take into account the indirect ways in which wji can influence the network outputs and hence Ed.
- For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network and denoted this set of units by Downstream(j).

● netj can influence the network outputs only through the units in Downstream(j). Therefore, we can write

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k\in Downstream(j)} \frac{\partial E_d}{\partial net_k}\frac{\partial net_k}{\partial net_j} \\
&= \sum_{k\in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k\in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j}\frac{\partial o_j}{\partial net_j} \\
&= \sum_{k\in Downstream(j)} -\delta_k \; w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k\in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1-o_j) \qquad \text{...........equ (8)}
\end{aligned}$$

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1-o_j) \sum_{k\in Downstream(j)} \delta_k \; w_{kj}$$

and

$$\Delta w_{ji} = \eta \; \delta_j \; x_{ji}$$

REMARKS ON THE BACKPROPAGATION ALGORITHM

1. Convergence and Local Minima

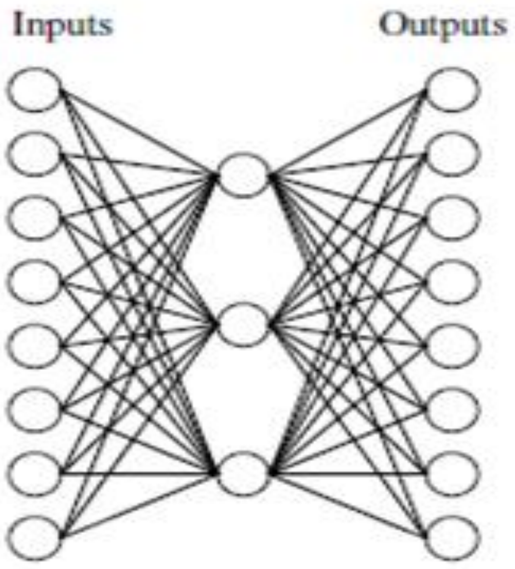
- The BACKPROPAGATION multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.
- Despite the lack of assured convergence to the global minimum error, BACKPROPAGATION is a highly effective function approximation method in practice.
- Local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases.

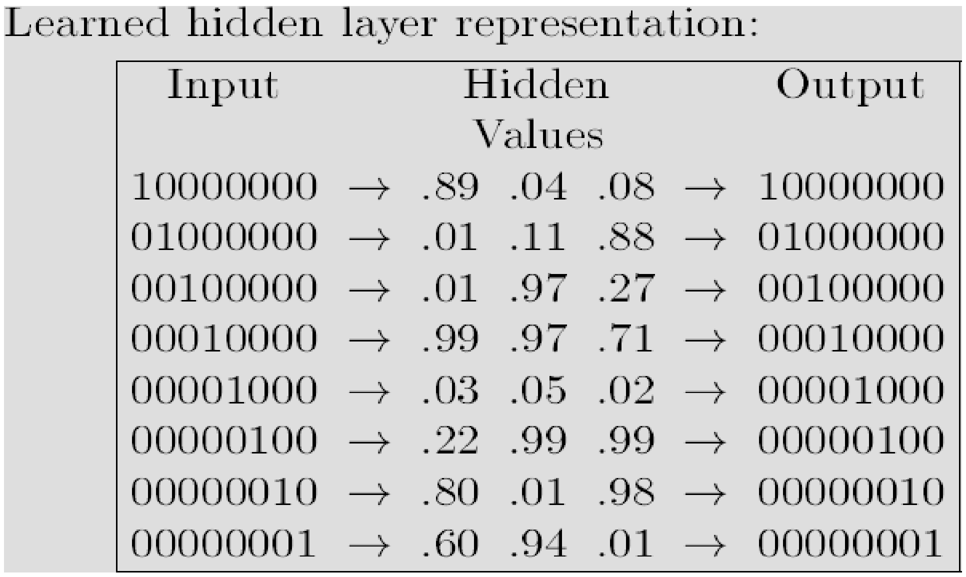
Common heuristics to attempt to alleviate the problem of local minima include:

1. Add a momentum term to the weight-update rule. Momentum can sometimes carry the gradient descent procedure through narrow local minima
2. Use stochastic gradient descent rather than true gradient descent
3. Train multiple networks using the same data, but initializing each network with different random weights

4.6.3 Hidden Layer Representations

BACKPROPAGATION can define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function. Consider example, the network shown in below Figure




Learned hidden layer representation:

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

- Consider training the network shown in Figure to learn the simple target function f (x)

= x, where x is a vector containing seven 0's and a single 1.

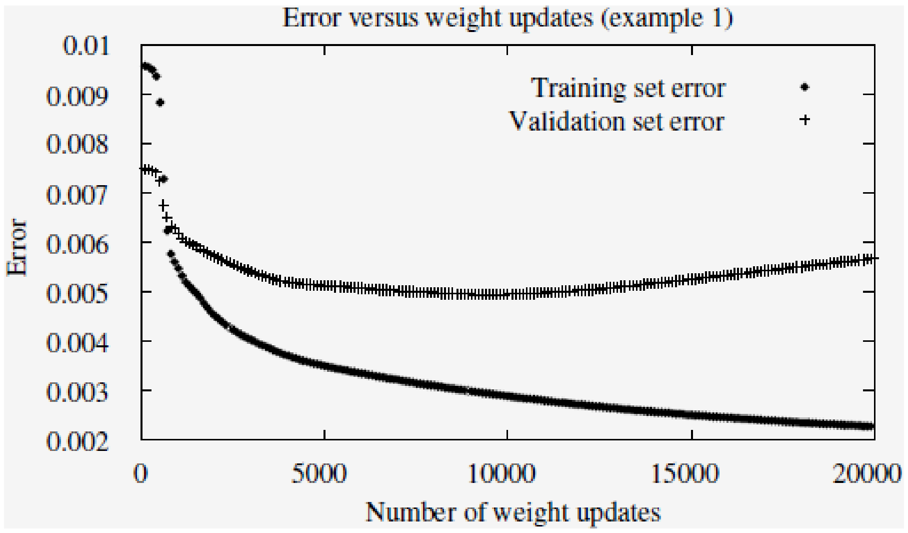
- The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network in this case is constrained to use only three hidden units. Therefore, the essential information from all eight input units must be captured by the three learned hidden units.

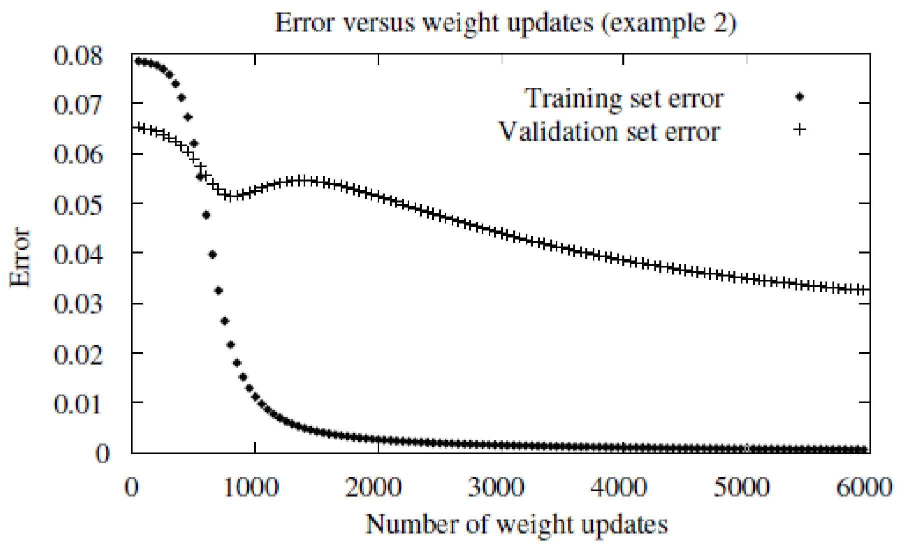
- When BACKPROPAGATION applied to this task, using each of the eight possible vectors as training examples, it successfully learns the target function. By examining the hidden unit values generated by the learned network for each of the eight possible input vectors, it is easy to see that the learned encoding is similar to the familiar standardbinary encoding of eight values using three bits (e.g., 000,001,010,. . . , 111). The exactvalues of the hidden units for one typical run of shown in Figure.

- This ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning

4.6.4 Generalization, Overfitting, and Stopping Criterion

What is an appropriate condition for terminating the weight update loop? One choice is to continue training until the error E on the training examples falls below some predeterminedthreshold. To see the dangers of minimizing the error over the training data, consider how the error Evaries with the number of weight iterations





- Consider first the top plot in this figure. The lower of the two lines shows the monotonically decreasing error E over the training set, as the number of gradient descent iterations grows. The upper line shows the error E measured over a different validation set of examples, distinct from the training examples. This line measures the generalization accuracy of the network-the accuracy with which it fits examples beyond the training data.

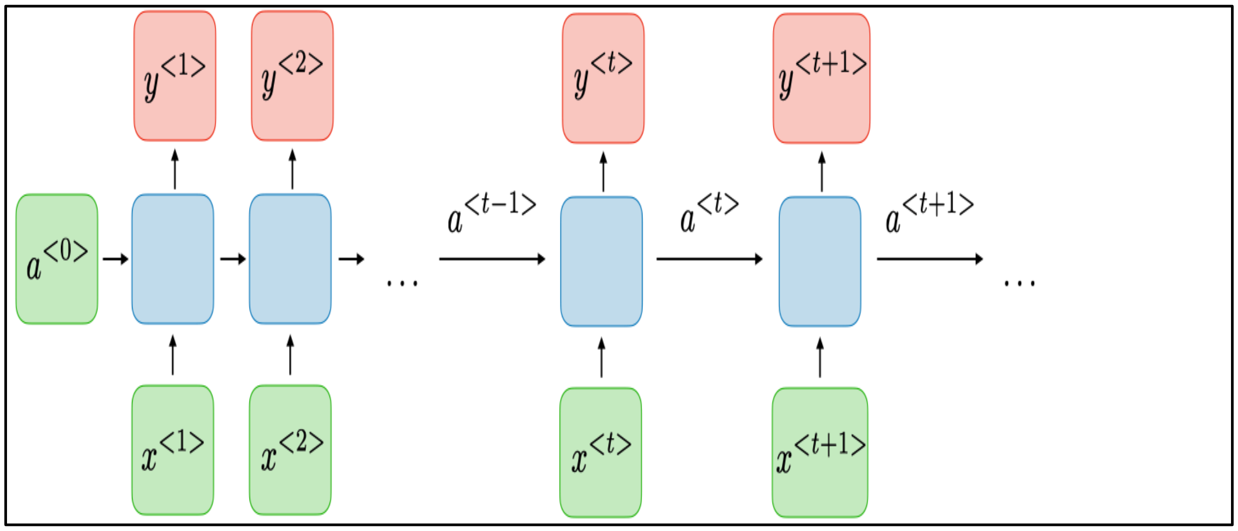
- The generalization accuracy measured over the validation examples first decreases, then increases, even as the error over the training examples continues to decrease. How can this occur? This occurs because the weights are being tuned to fit idiosyncrasies of the training examples that are not representative of the general distribution of examples. The large number of weight parameters in ANNs provides many degrees of freedom for fitting such idiosyncrasies

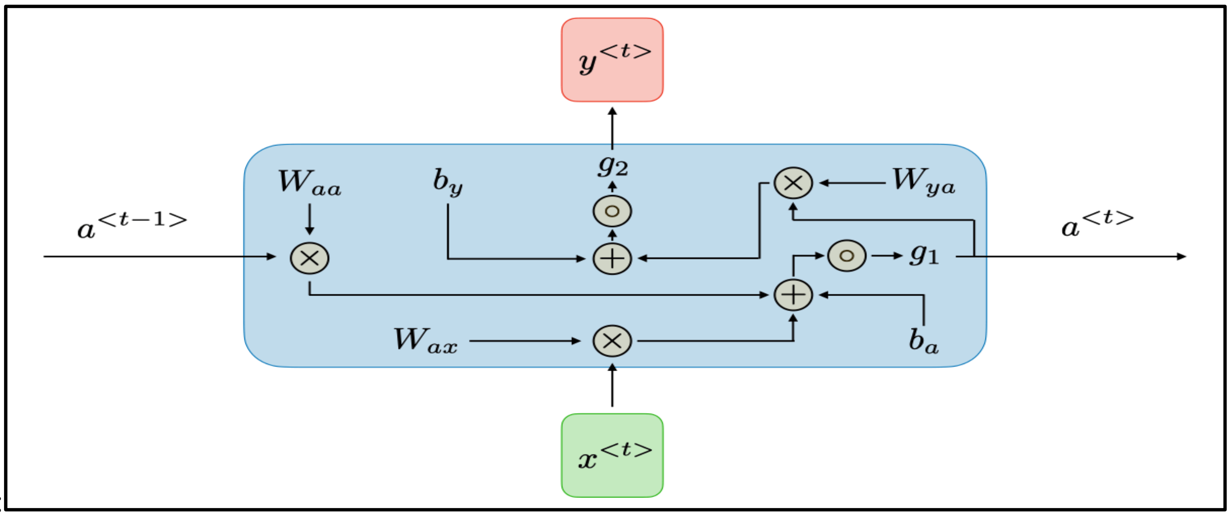
- Why does overfitting tend to occur during later iterations, but not during earlier iterations?

By giving enough weight-tuning iterations, BACKPROPAGATION will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresentative characteristics of the particular training sample.

4.7 Recurrent networks

Architecture of a traditional RNN Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. They are typically as follows:





:

Applications of Recurrent Neural Networks (RNNs)

- Prediction problems.
- Language Modelling and Generating Text.
- Machine Translation.
- Speech Recognition.
- Generating Image Descriptions.
- Video Tagging.
- Text Summarization.
- Call Center Analysis

4.8 Support Vector Machine

Support Vector Machine Algorithm

- Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.
- The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.
- SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane:

Advantages of Support Vector algorithm

- Support vector machine is very effective even with high dimensional data.
- When you have a data set where number of features is more than the number of rows of data, SVM can perform in that case as well.
- When classes in the data are points are well separated SVM works really well.
- SVM can be used for both regression and classification problem.
- And last but not the least SVM can work well with image data as well.

Disadvantages of Support Vector algorithm

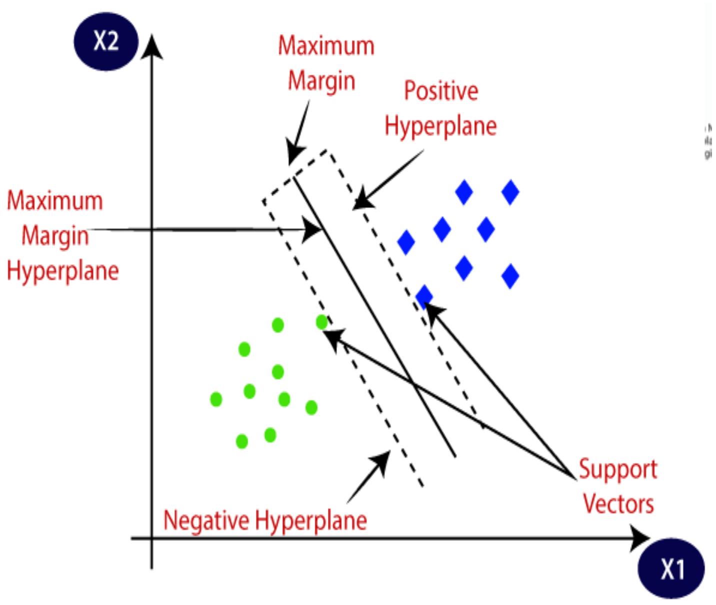
- When classes in the data are points are not well separated, which means overlapping classes are there, SVM does not perform well.
- We need to choose an optimal kernel for SVM and this task is difficult.
- SVM on large data set comparatively takes more time to train.
- SVM or Support vector machine is not a probabilistic model so we can not explanation the classification in terms of probability.
- It is difficult to understand and interpret the SVM model compared to Decision tree as SVM is more complex.

Applications of support vector machine:

- **Face observation:** It is used for detecting the face according to the classifier and model.
- **Text and hypertext arrangement: In** this, the categorization technique is used to find important information or you can say required information for arranging text.
- **Grouping of portrayals :**It is also used in the Grouping of portrayals for grouping or you can say by comparing the piece of information and take an action accordingly.
- **Bioinformatics :**In is also used for medical science as well like in laboratory, DNA, research, etc.
- **Handwriting remembrance:** In this, it is used for handwriting reorganization.
- **Protein fold and remote homology spotting:** It is used for spotting or you can say the classification class into functional and structural classes given their amino acid sequences. It is one of the problems in bioinformatics.
- **Generalized predictive control(GPC) :**It is also used for Generalized predictive control(GPC) for predicting and it relies on predictive control using a multilayer feed-forward network as the plants linear model is presented

SVM can be of two types:

- **Linear SVM:** Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
- **Non-linear SVM:** Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.



4.8.1 Linear Discriminant Functions for Binary Classification

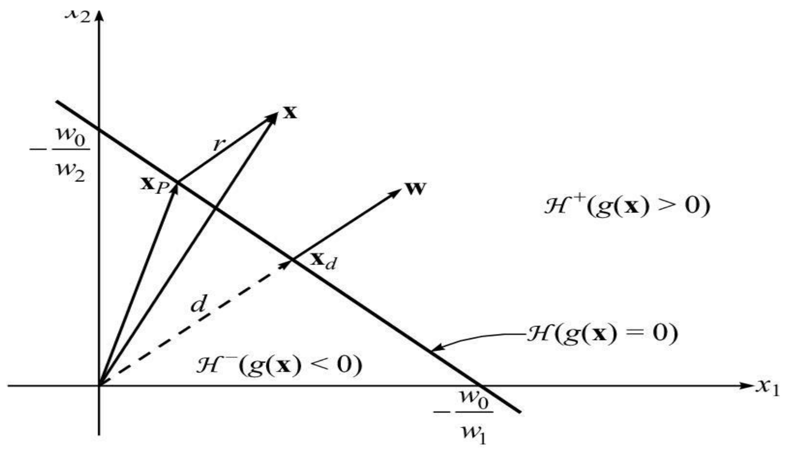
Linear discriminant function for n-dimensional feature space in $\Re^n$:

$$g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0 = 0$$

$\mathbf{x} = [x_1\ x_2\ x_3 \dots x_n]^T$ - The feature vector

$\mathbf{w} = [w_1\ w_2\ w_3 \dots w_n]^T$ - The weight vector

$\mathrm{w}_\mathbf{0}$ - bias parameter
Discriminant function *hyperplane* $\mathcal{H}$. For discriminant function $g(\mathbf{x})$, two-category classifier has the decision rule:
Decide Class 1 if $g(\mathbf{x})>0$ and Class 2 if $g(\mathbf{x})<0$
Location of any point $\mathbf{x}$ may be considered relative to $\mathcal{H}$. Defining $\boldsymbol{x}_p$as the normal projection projection of $\boldsymbol{x}_p$as the normal projection of $\mathbf{x}$ onto $\mathcal{H}$


x_2
x
r
$-\frac{w_0}{w_2}$
x_P
w
$\mathcal{H}^+(g(\mathbf{x}) > 0)$
x_d
d
$\mathcal{H}(g(\mathbf{x}) = 0)$
$\mathcal{H}^-(g(\mathbf{x}) < 0)$
x_1
$-\frac{w_0}{w_1}$

$\mathcal{H}_1$

$$\mathbf{x} = \mathbf{x}_P + r\frac{\mathbf{w}}{\|\mathbf{w}\|}$$

where$\|\mathbf{w}\|$is the Euclidean norm of $\mathbf{w}$ and $\frac{\mathbf{w}}{\|\mathbf{w}\|}$ is a unit vector.

It can be shown that

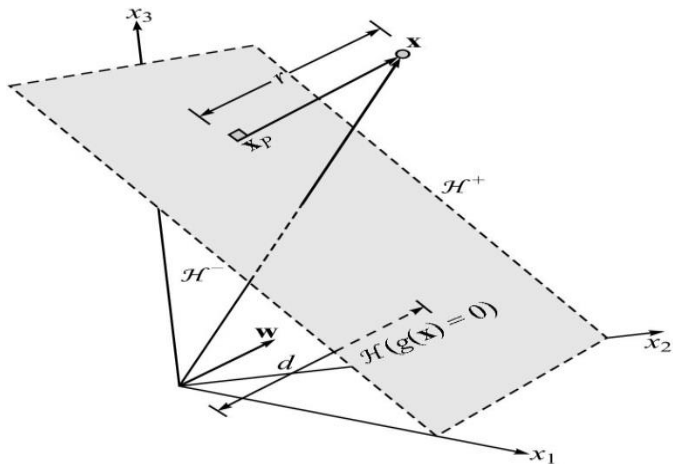
$$r = \frac{g(\mathbf{x})}{\|\mathbf{w}\|}$$

$|g(\mathbf{x})|$ is a measure of the Euclidean distance of the point $\mathbf{x}$ from the decision hyperplane $\mathcal{H}$.

$$g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0 \begin{cases} > 0 \text{ if } \mathbf{x} \in \mathcal{H}^+ \\ = 0 \text{ if } \mathbf{x} \in \mathcal{H} \\ < 0 \text{ if } \mathbf{x} \in \mathcal{H}^- \end{cases}$$

Perpendicular distance d from coordinate origin to $\mathcal{H} = w_0/\|\mathbf{w}\|$

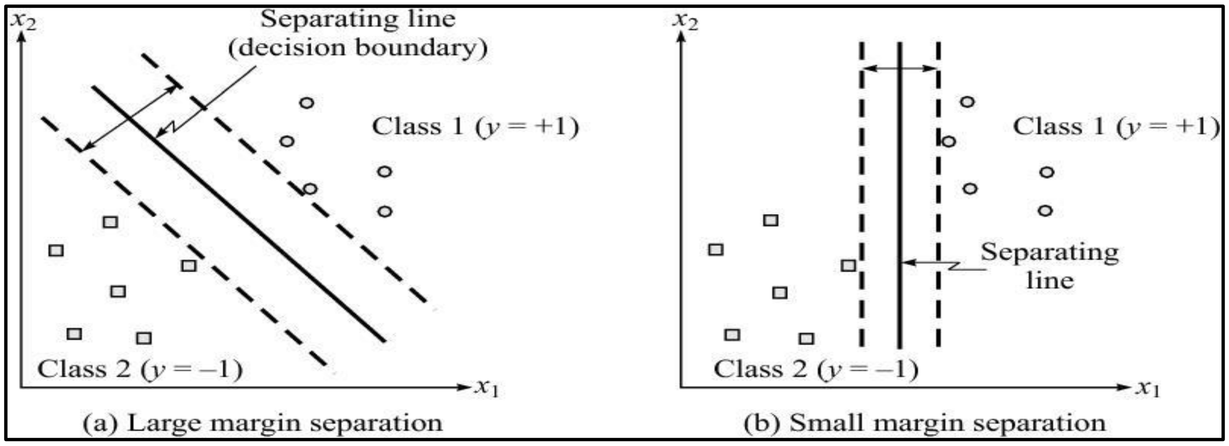
Geometry for 3-dimensions (n=3)



Hyperplane $\mathcal{H}$ separates the feature space into two half space $\mathcal{H}^+$ and $\mathcal{H}^-$

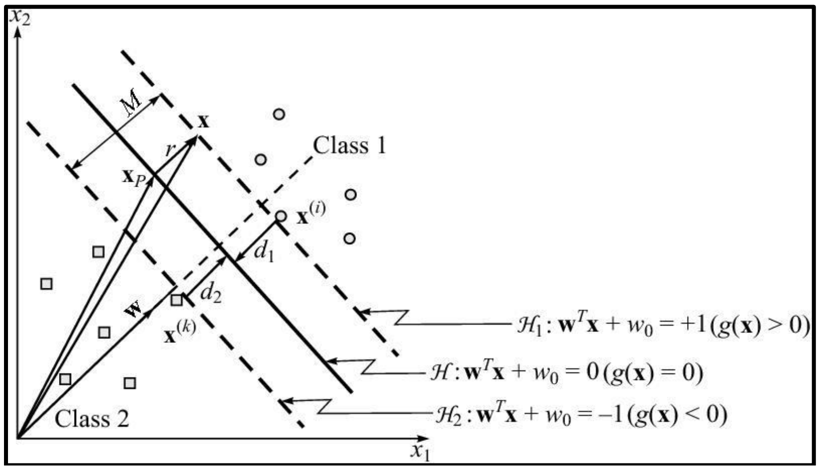
- For linearly separable, many hyperplanes exist to perform. Separation.

- SVM framework tells which hyperplane is best.
- Hyperplane with the largest margin which minimizes training error.
- Select the decision boundary that is far away from both the classes. Large margin separation is expected to yield good generalization.



x_2
Separating line (decision boundary)
Class 1 ($y = +1$)
Class 2 ($y = -1$)
x_1
(a) Large margin separation
Separating line
(b) Small margin separation

Two parallel hyperplanes $\mathcal{H}_1$ and $\mathcal{H}_2$ that pass through $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(k)}$ respectively



x_2
M
$\mathbf{x}$
r
$\mathbf{x}_P$
Class 1
$\mathbf{x}^{(i)}$
d_1
d_2
$\mathbf{w}$
$\mathbf{x}^{(k)}$
$\mathcal{H}_1: \mathbf{w}^T\mathbf{x} + w_0 = +1\,(g(\mathbf{x}) > 0)$
$\mathcal{H}: \mathbf{w}^T\mathbf{x} + w_0 = 0\,(g(\mathbf{x}) = 0)$
$\mathcal{H}_2: \mathbf{w}^T\mathbf{x} + w_0 = -1\,(g(\mathbf{x}) < 0)$
Class 2
x_1

Geometric interpretation of algebraic distances of points to a hyperplane for two-dimensional case

4.8.2 Estimation of Margin

$\mathcal{H}_{\mathcal{H}_1}$ and $\mathcal{H}_2$ are parallel to the hyperplane $\mathbf{w}^T\mathbf{x} + w_0 = 0$.

$$\mathcal{H}_1\text{: } \mathbf{w}^T\mathbf{x} + w_0 = +1$$
$$\mathcal{H}_2\text{: } \mathbf{w}^T\mathbf{x} + w_0 = -1$$

such that

$$\mathbf{w}^T\mathbf{x}^{(i)} + w_0 \geq 1 \text{ if } y^{(i)} = +1$$
$$\mathbf{w}^T\mathbf{x}^{(i)} + w_0 \leq -1 \text{ if } y^{(i)} = -1$$

or equivalently,

$$y^{(i)}\left(\mathbf{w}^T\mathbf{x}^{(i)} + w_0\right) \geq 1$$

distance between the two hyperplanes = margin M

$$M = \frac{2}{||\mathbf{w}||}$$

Linearly separable training examples,

$$\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(N)}, y^{(N)})\}$$

4.8.3 Quadratic programming solution to finding maximum margin separators

Problem: Solve the following constrained minimization problem:

$$minimize\ f(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w}$$

$$\text{subject to } y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + w_0) \geq 1; i = 1, \dots, N$$

This is the formulation of hard-margin SVM.

Dual formulation of constrained optimization problem:

Lagrangian is constructed:

$$L(\mathbf{w}, w_{\mathbf{0}}, \boldsymbol{\lambda}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{i=1}^{N} \lambda_i \left[y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + w_0) - 1\right]$$

The KKT conditions are as follows:

i. $\frac{\partial L}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_{i=1}^{N} \lambda_i y^{(i)}\mathbf{x}^{(i)}$

$\frac{\partial L}{\partial w_0} = 0 \Rightarrow \sum_{i=1}^{N} \lambda_i y^{(i)} = 0$

ii. $y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + w_0) - 1 \geq 0; i = 1, \dots, N$

iii. $\lambda_i \geq 0; i = 1, \dots, N$

iv. $\lambda_i(y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + w_0) - 1) = 0; i = 1, \dots, N$

After solving the dual problem numerically, the resulting optimum λ_i values are used to compute $\mathbf{w}$ and w_0 using the KKT conditions.

$\mathbf{w}$ is computed using condition (i) of KKT conditions

$$\mathbf{w} = \sum_{i=1}^{N} \lambda_i y^{(i)}\mathbf{x}^{(i)}$$

- Very small percentage have $\lambda_i > 0$.

$\mathbf{x}^{(i)}$ whose $\lambda_i > 0$ – *support vectors*

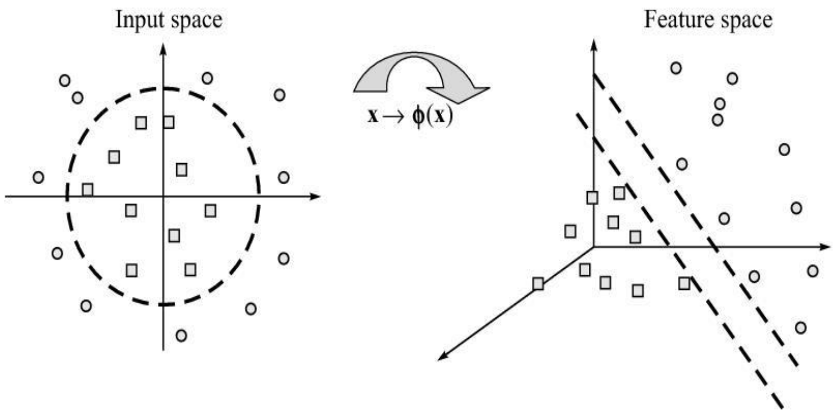
$$\mathbf{w} = \sum_{i \in svindex} \lambda_i y^{(i)}\mathbf{x}^{(i)}$$

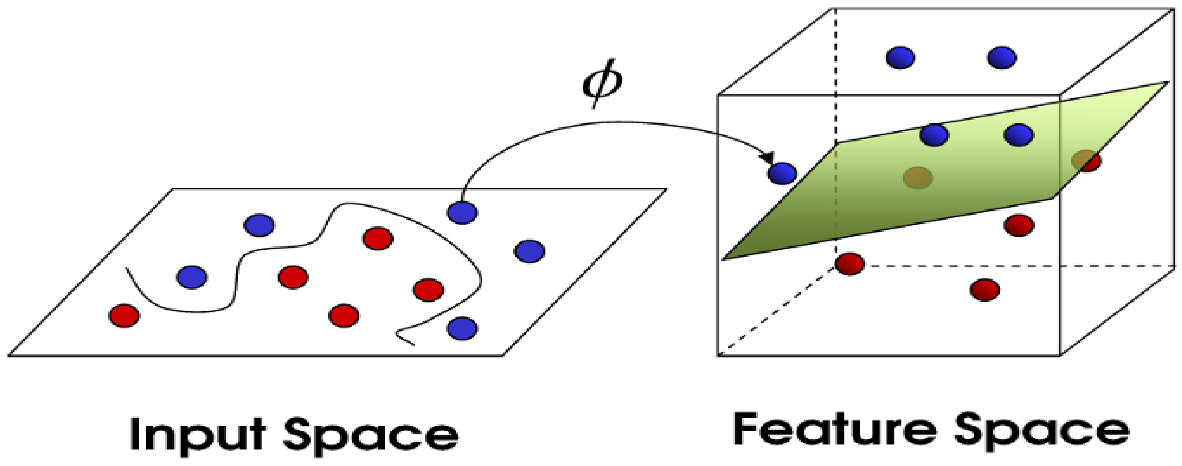
where $svindex$ is the set of indices of support vectors

$$w_0 = \frac{1}{|svindex|} \sum_{i \in svindex} (y^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)})$$

where $|svindex|$ is the total number of indices in $svindex$

4.8.4 . Kernels for learning non-linear functions : For training examples which cannot be linearly separated. In the feature space, they can be separated linearly with some transformations





Examples of SVM Kernels

Let us see some common kernels used with SVMs and their uses:

Polynomial kernel: It is popular in image processing. Equation is:

$$k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j} + 1)^d$$

Gaussian kernel: It is a general-purpose kernel; used when there is no prior knowledge about the data. Equation is:

$$k(x, y) = \exp\left(-\frac{||x-y||^2}{2\sigma^2}\right)$$

Gaussian radial basis function (RBF): It is a general-purpose kernel; used when there is no prior knowledge about the data. Equation is:

$k(\mathbf{x_i}, \mathbf{x_j}) = \exp(-\gamma \|\mathbf{x_i} - \mathbf{x_j}\|^2)$, for $\gamma > 0$

Laplace RBF kernel: It is general-purpose kernel; used when there is no prior knowledge about the data. Equation is:

$$k(x, y) = \exp\left(-\frac{\|x - y\|}{\sigma}\right)$$

Hyperbolic tangent kernel: We can use it in neural networks. Equation is:

$$k(\mathbf{x_i}, \mathbf{x_j}) = \tanh(\kappa \mathbf{x_i} \cdot \mathbf{x_j} + c)$$

for some (not every) k>0 and c<0.

Sigmoid kernel: We can use it as the proxy for neural networks. Equation is

$k(x, y) = \tanh(\alpha x^T y + c)$