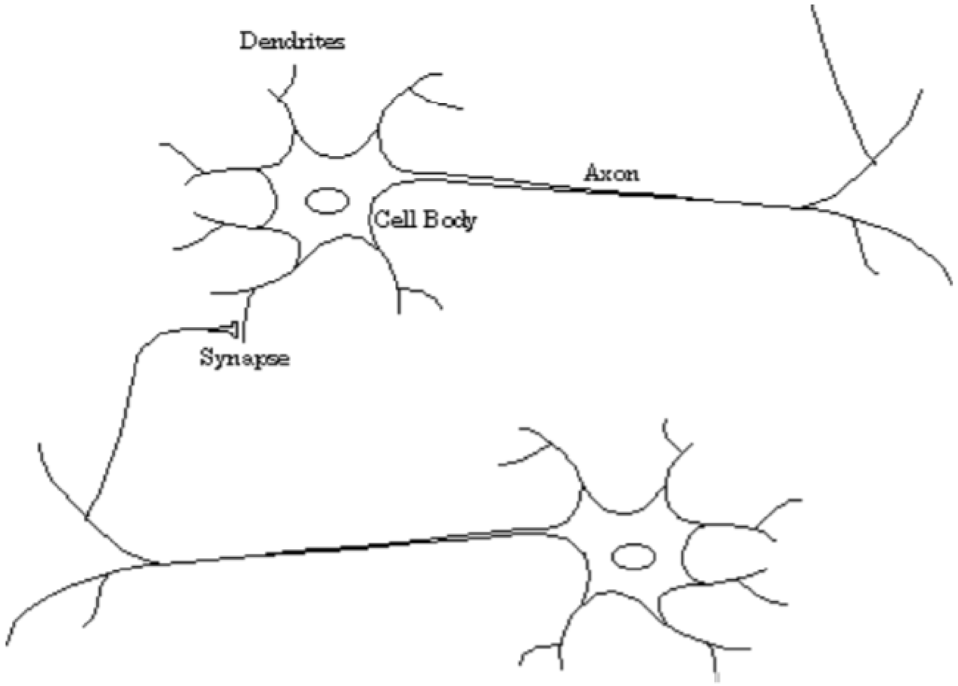
Neural Networks

Biological Inspirations

- Human brain is known to be the most intelligent learning model
- One way to build intelligent machines is to try to imitate the human brain.
- Neural network is a machine learning model which mimics the behavior of human brain

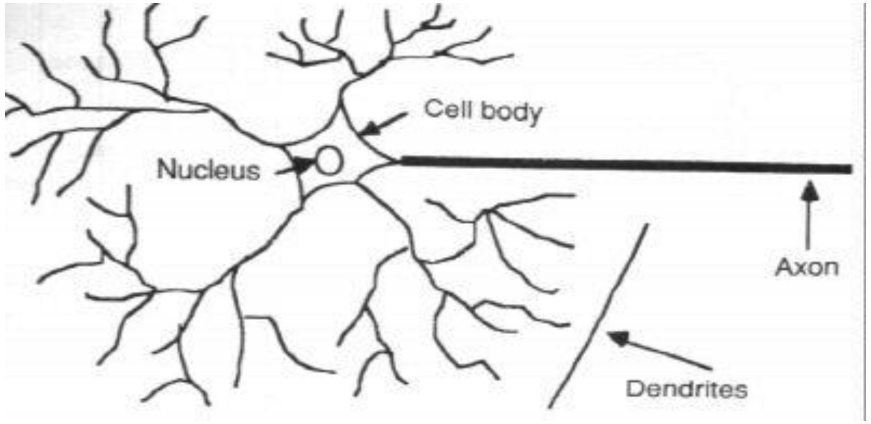
Biological Neuron

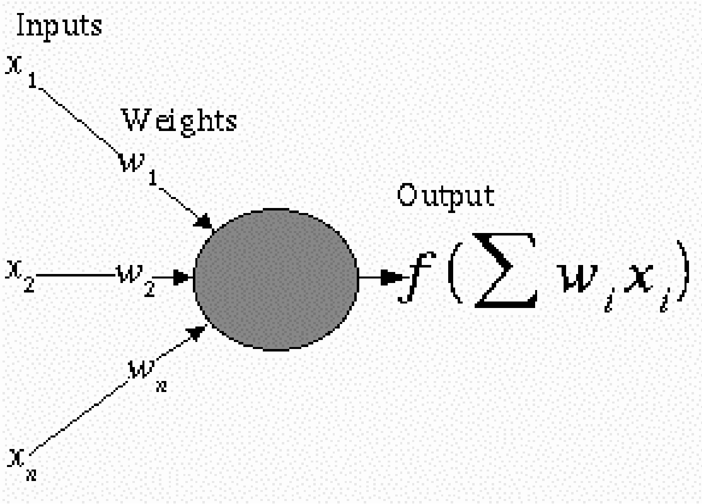
- Dendrites: nerve fibres carrying electrical signals to the cell
- Cell body: computes a non-linear function of its inputs
- Axon: one long fiber that carries the electrical signal from the cell body to other neurons
- Synapse: the point of contact between the axon of one cell and the dendrite of another, regulating a chemical connection whose strength affects the input to the cell.



Neural Network Representation

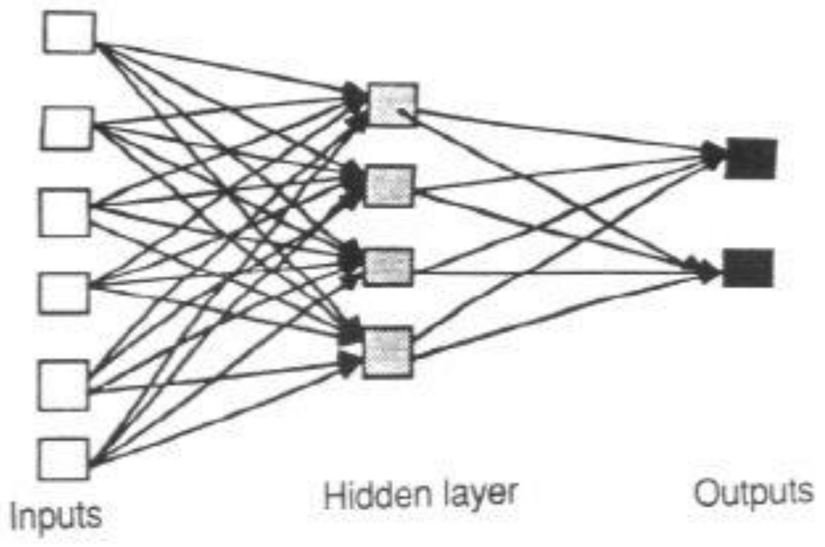
- An ANN is composed of processing elements, organized in different ways to form the network's structure.
- Each element receives inputs, processes inputs and delivers a single output.
- The input can be raw input data or the output of other elements. The output can be the final result (e.g. 1 means yes, 0 means no) or inputs to other elements.





The Network

- Each ANN is composed of a collection of processing elements grouped in layers. A typical structure is shown below
- Note the three layers: input, intermediate (called the ***hidden layer***) and output.
- Several hidden layers can be placed between the input and output layers.

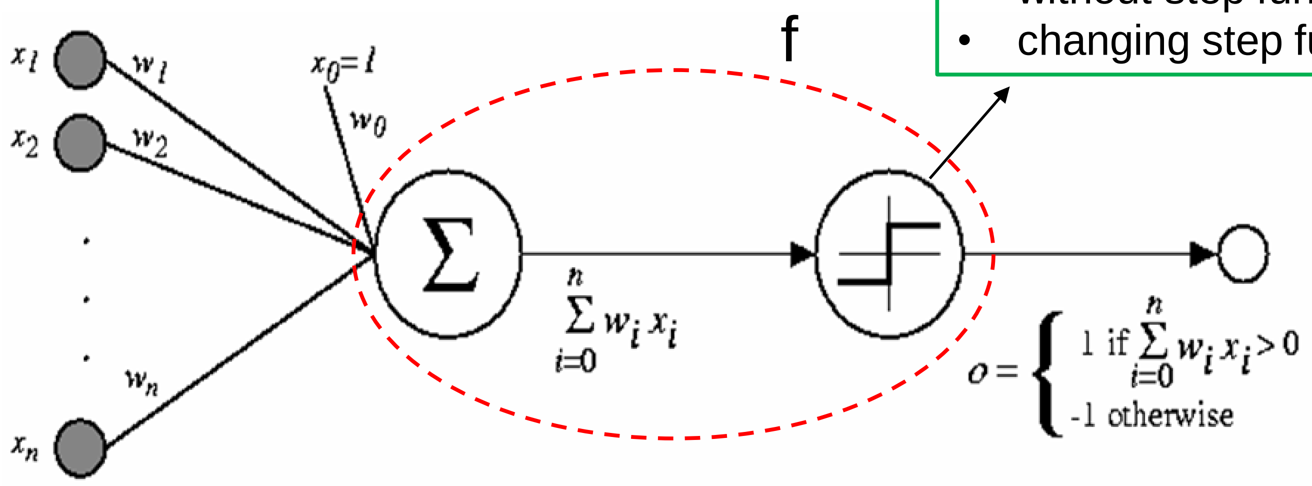


Perceptrons

- The first neural network model
  - Input layer & output layer only (no hidden layer)
- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs using a step function
- Given real-valued inputs x_1 through x_n, the output(activation function) $o(x_1, \ldots, x_n)$ is

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

where w_i is a real-valued constant, or *weight*.



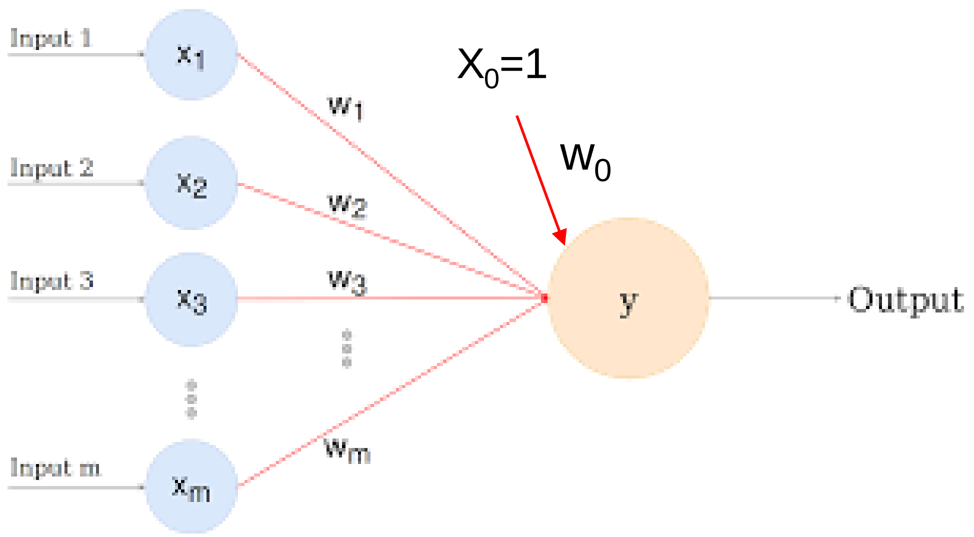
- without step function: perceptron linear regression.
- changing step function to sigmoid -> ?

Perceptrons

- To simplify notation, we imagine an additional dummy input $x_0 = 1$, allowing us to write the above inequality as

$$\sum_{i=0}^{n} w_i x_i > 0 \quad \text{instead of} \quad w_0 + w_1x_1 + \ldots + w_nx_n > 0$$

- Learning a perceptron involves choosing values for the weights $w_0, w_1, \ldots, w_n$.

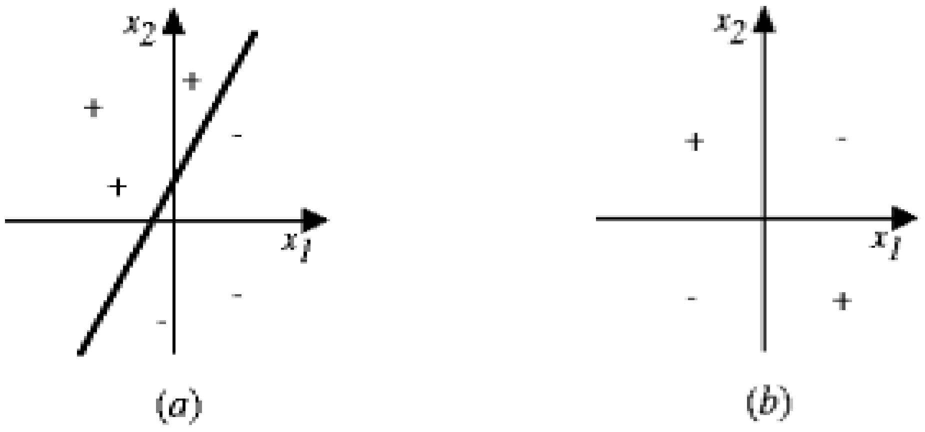


Representation Power of Perceptrons

- We can view the perceptron as representing a linear **decision surface** in the *n*-dimensional space (linear classifier)

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

- The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a –1 for instances lying on the other side



Representation Power of Perceptrons

A single perceptron can be used to represent many Boolean functions

AND function :

<Training examples>

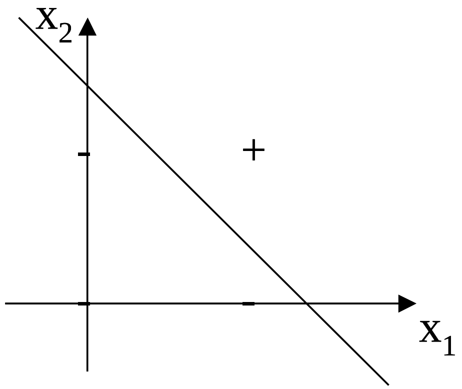
x_1	x_2	output
0	0	−1
0	1	−1
1	0	−1
1	1	1

Decision hyperplane :

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$
$$-0.8 + 0.5 x_1 + 0.5 x_2 = 0$$

<Test Results>

x_1	x_2	$\Sigma w_i x_i$	output
0	0	−0.8	−1
0	1	−0.3	−1
1	0	−0.3	−1
1	1	0.2	1



$-0.8 + 0.5 x_1 + 0.5 x_2 = 0$

Representation Power of Perceptrons

OR function：

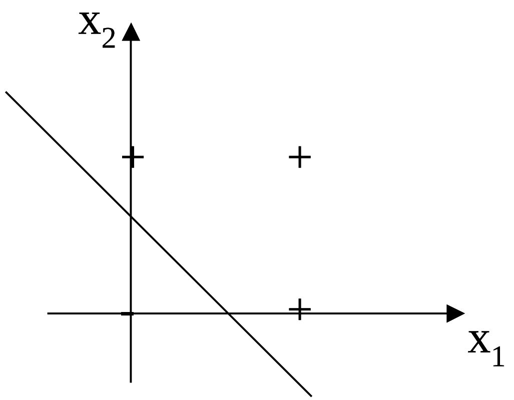
- The two-input perceptron can implement the OR function when we set the weights: $w_0 = -0.3$, $w_1 = w_2 = 0.5$

<Training examples>		
x_1	x_2	output
0	0	–1
0	1	1
1	0	1
1	1	1

Decision hyperplane :

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$
$$-0.3 + 0.5 x_1 + 0.5 x_2 = 0$$

<Test Results>			
x_1	x_2	$\Sigma w_i x_i$	output
0	0	–0.3	–1
0	1	0.2	1
1	0	0.2	1
1	1	0.7	1



$-0.3 + 0.5 x_1 + 0.5 x_2 = 0$

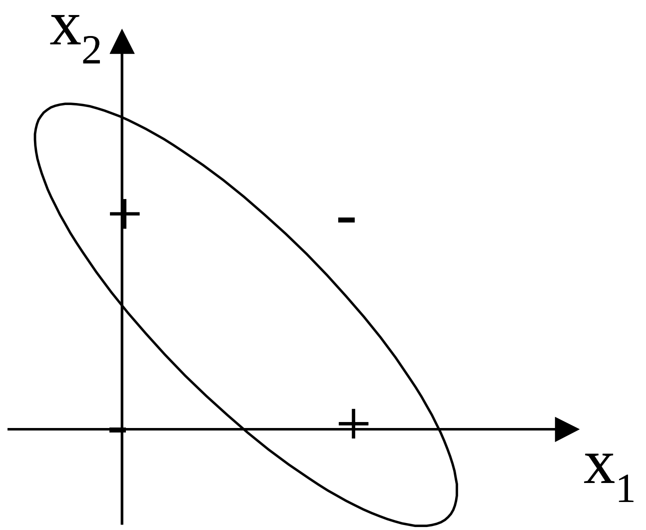
Representation Power of Perceptrons

XOR function :

- It's impossible to implement the XOR function by a single perception.

<Training examples>

x_1	x_2	output
0	0	−1
0	1	1
1	0	1
1	1	−1

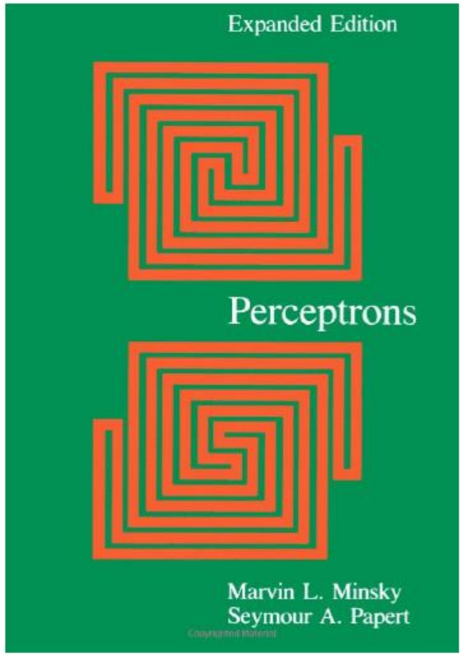


- A multi-layer network of perceptron can represent XOR function.

Criticism and Downfall

- Some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called *linearly separated set of examples.* [Seymour Papert and Marvin Minsky, 1968]
- Perceptrons are painfully limited. Perceptrons are linear classifiers and thus they can not even learn a simple XOR function
- Interest in NN close to fully disappeared
- Beginning of the

  Dark age of Artificial Intelligence



How to Train Perceptrons?

- Let us begin by understanding how to learn the weights for a single perceptron.

- We consider two learning methods in perceptron:
  - **Perceptron training rule(PTR) : step activation function**
  - **Delta rule : linear activation function**

- Similar update rules with different activation functions

Perceptron training rule

- Classification(Supervised learning): Network is provided with a set of examples of correct network behavior(inputs/targets)

$$(x_1,t_1),\ (x_2,t_2),\ \ldots\ (x_n,t_n)$$

- Output of perceptron(o_i) is computed using input(x_i) & weight vector(***W***)
  - output is a step function

$$o_i = f(x_i, \boldsymbol{W})$$

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } \sum_{i=0} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$$

- Training in neural networks (in perceptron) is to determine a weight vector(***W***) that causes the neural network(perceptron) to produce the correct output(o_i) (+1/-1 in Perceptron) for each of the given training examples(x_i)

$$\boxed{\textit{compute } \boldsymbol{W} \ \textit{ such that } \ t_i = o_i \ \ \forall i}$$

Perceptron training rule

- **Perceptron training rule:**
  1) Begin with random weights,
  2) Compute the output of perceptron from training examples
  3) Modifying the perceptron weights whenever it misclassifies an example.
  4) This process is repeated until the perceptron *classifies* all training examples correctly.

- Weights are modified at each step according to the perceptron training rule, which revises the weight w_i associated with input x_i according to the rule.

$$w_i \leftarrow w_i + \Delta w_i \qquad \text{where} \quad \Delta w_i = \eta(t_i - o_i)x_i$$

- Here:
  - t_i is target output value for the *i*-th training example
  - o_i is perceptron output (1 or -1) of the *i*-th training example
  - η is small constant (e.g., 0.1) called *learning rate*

Perceptron training rule

- The role of the **learning rate(η)** 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)

- We can prove that the algorithm will converge if
  1) training data is linearly separable and
  2) η is sufficiently small.

- If the data is not linearly separable, convergence is not guaranteed.

- **Problems of Perceptron Rule**: Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it fails to converge if the examples are *not linearly separable*.

Delta Rule

- **Delta training rule**: the output *o* is given by

$$o = w_0 + w_1x_1 + \cdots + w_nx_n$$

  - *Remember: In Perceptron Training Rule, o is step function*

- Delta rule (DR) is similar to the Perceptron Training Rule (PTR), with some differences:

1. DR can be derived for any continuous differentiable output/activation function o, whereas in PTR only works for threshold/step output function
2. Error in DR is not restricted to having values of 0, 1, or - 1 (as in PTR), but may have any value
3. DR is based on gradient descent method

Basics of Gradient Descent Method

- Suppose we have a loss(error) function E(w)

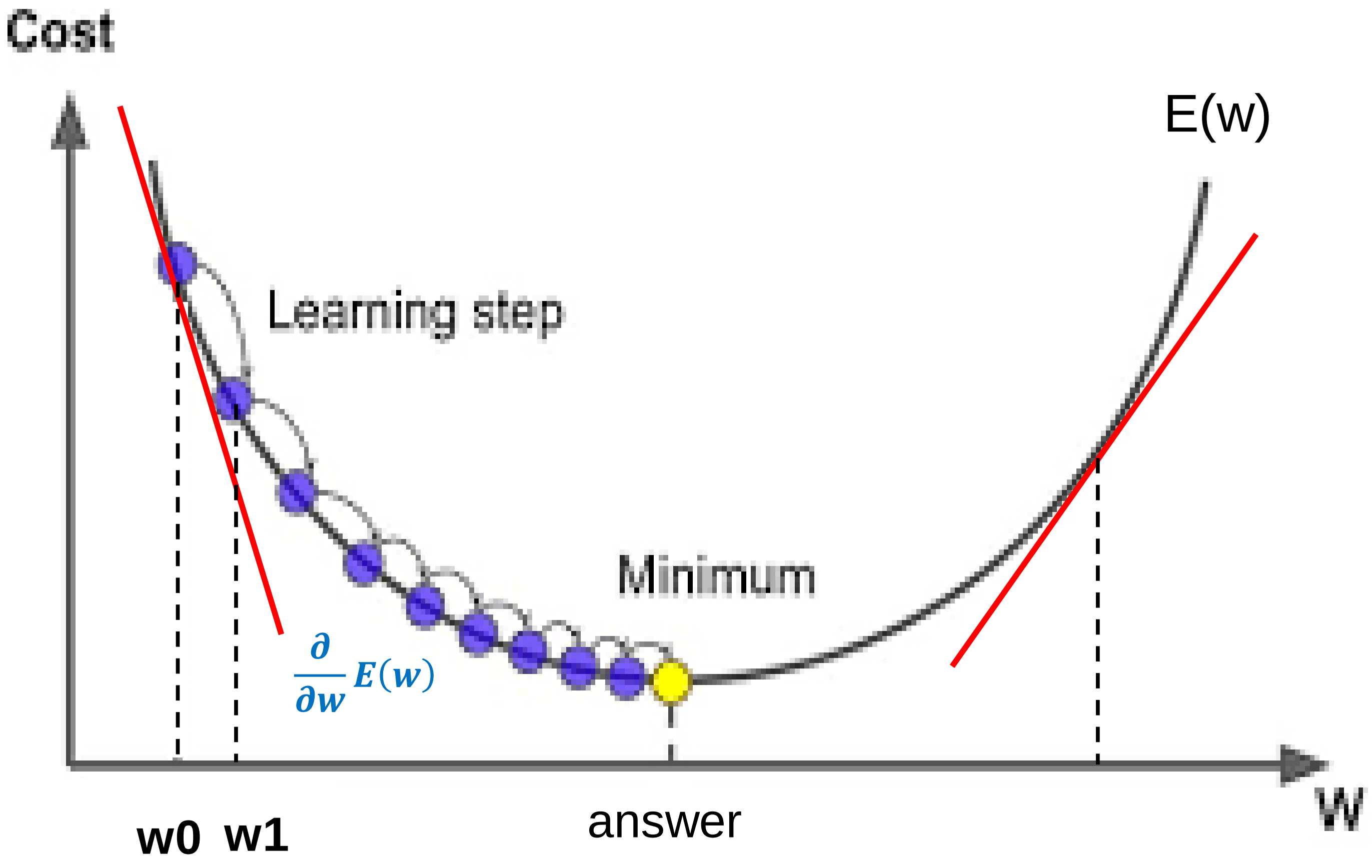
- We want to find w values which minimize E(w)
  - Maximize: gradient ascent

- Gradient Descent Rule:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \boldsymbol{\eta} \frac{\boldsymbol{\partial}}{\boldsymbol{\partial w}} \boldsymbol{E}(\boldsymbol{w})$$

- η is called the learning rate. A small positive number, e.g. $\eta = 0.05$

Basics of Gradient Descent Method



$$w \leftarrow w - \eta \frac{\partial}{\partial w} E(w)$$

Derivation of the Gradient Descent Rule

- Given error(loss) function E(w), this vector derivative is called the *gradient* of *E(w)* with respect to the vector <$w_1, \ldots, w_n$>, written $\frac{\partial}{\partial \boldsymbol{w}} \boldsymbol{E}(\boldsymbol{w})$

$$\frac{\partial}{\partial \boldsymbol{w}} \boldsymbol{E}(\boldsymbol{w}) = \begin{pmatrix} \frac{\partial}{\partial \boldsymbol{w}_1} \boldsymbol{E}(\boldsymbol{w}) \\ \vdots \\ \frac{\partial}{\partial \boldsymbol{w}_n} \boldsymbol{E}(\boldsymbol{w}) \end{pmatrix}$$

- Notice $\frac{\partial}{\partial w} E(w)$ is itself a vector, whose components are the partial derivatives of *E* with respect to each of the w_i.

- The training rule for gradient descent is

$$\boxed{\boldsymbol{w}_i \leftarrow \boldsymbol{w}_i - \boldsymbol{\eta} \frac{\partial}{\partial \boldsymbol{w}_i} \boldsymbol{E}(\boldsymbol{w})}$$

where w_i is the *i*-th weight

Gradient Descent Method

- The **gradient descent** algorithm is as follows:

  1) Pick an initial random weight vector.

  2) Compute Δw_i for each weight

$$\Delta \mathbf{w_i} = -\boldsymbol{\eta} \frac{\boldsymbol{\partial}}{\boldsymbol{\partial w_i}} \boldsymbol{E}(\boldsymbol{w})$$

  3) Update each weight w_i by adding Δw_i ,

$$\mathbf{w_i} \leftarrow \mathbf{w_i} + \Delta \mathbf{w_i}$$

  4) then repeat the process.

- Because the error surface in perceptron contains only a single global minimum, delta update rule in perceptron will converge to a **global optimal** weight vector with minimum error

The Error/Loss Function of Perceptron

- We define the ***error function*** of neural network with a weight vector. The Error function (E[w]) can be defined as the following squared error

$$\boldsymbol{E[W] = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2}$$

where D is set of training examples, t_d is the target output for the training example d and o_d is the output of the neural network for the training example d. o_d is a function of w (weights).

- Here we characterize E as ***a function of weight vector*** because the neural network output o_d depends on this weight vector.

Derivation of Delta Rule

- We have to compute $\frac{\partial}{\partial w_i} E(w)$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$= \sum_{d \in D} (t_d - o_d)\,(-x_{i,d}) \qquad \textit{since } o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

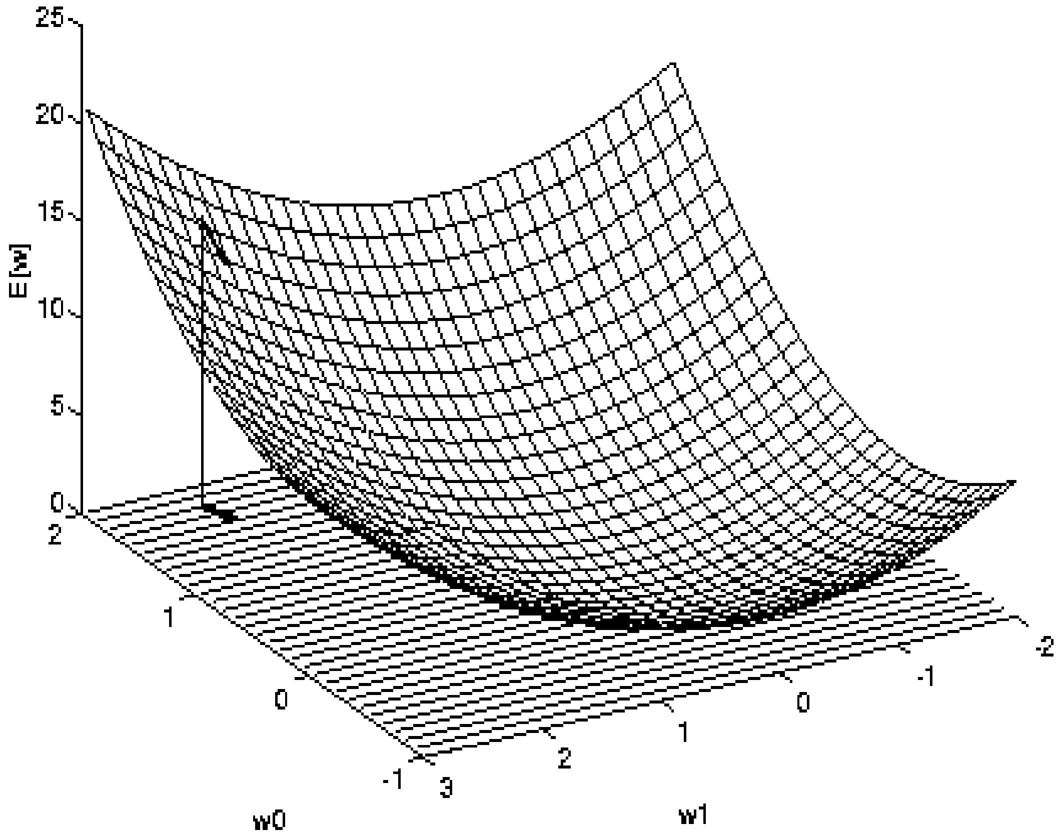
where $x_{i,d}$ denotes the single input component x_i for the training example *d*

- The final **weight update rule** in Delta Rule:

$w_i \leftarrow w_i + \Delta w_i$

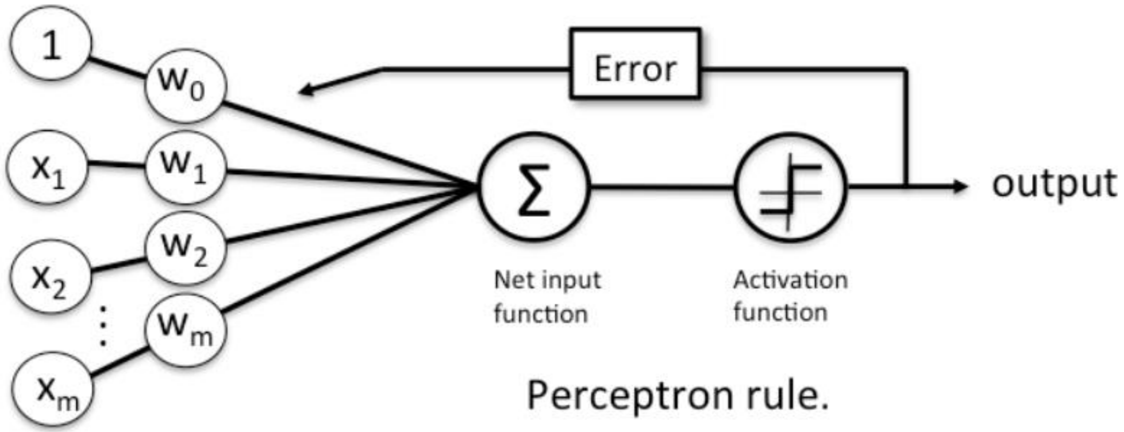
where $$\boldsymbol{\Delta w_i = -\eta \frac{\partial}{\partial w_i} E(w) = \eta \sum_{d \in D} (t_d - o_d) x_{i,d}}$$

The Error Surface of Perceptron

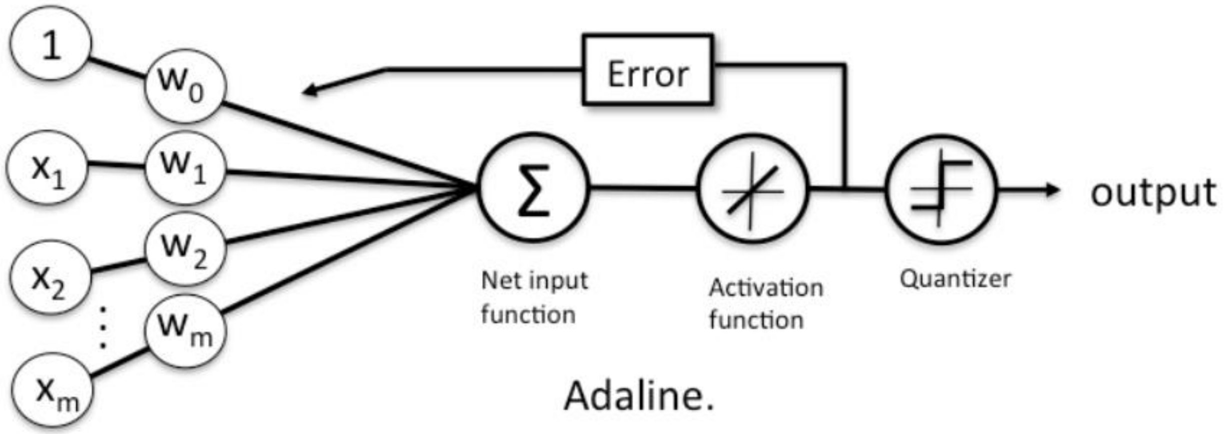


- Here the axes w_o,w_1 represents possible values for the two weights
- The vertical axis indicates the error *E* relative to some fixed set of training examples

Perceptron Rule vs Delta Rule(Adaline)



Perceptron rule.



Adaline.

https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html

Delta Rule

- One pass through all the weights for the whole training set is called an epoch of training of training.

- After many epochs, the network outputs match the targets for all the training data and the training process ceases. We then say that the training process has converged to a solution.

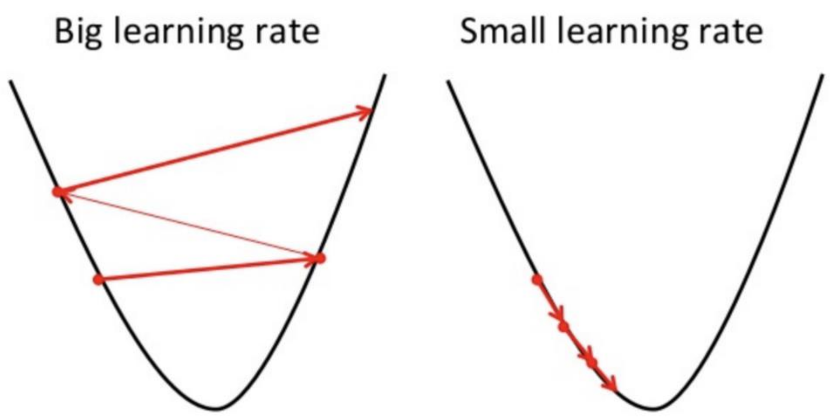
- If the problem is linearly separable, then both PTR and DR will find a set of weights in a finite number of iterations that solves the problem correctly.

- If the problem is NOT linearly separable, then the DR will find a set of weights in a finite number of iterations that minimizes the error while PTR keeps oscillating.

Proper values of learning rate are important

- If learning rate(η value in update rule) is too large, the **gradient descent** search runs the risk of overstepping the minimum in the error surface

- For this reason, we sometimes **gradually reduce** the value of η as the number of gradient descent steps grows.



Oscillate

Too slow