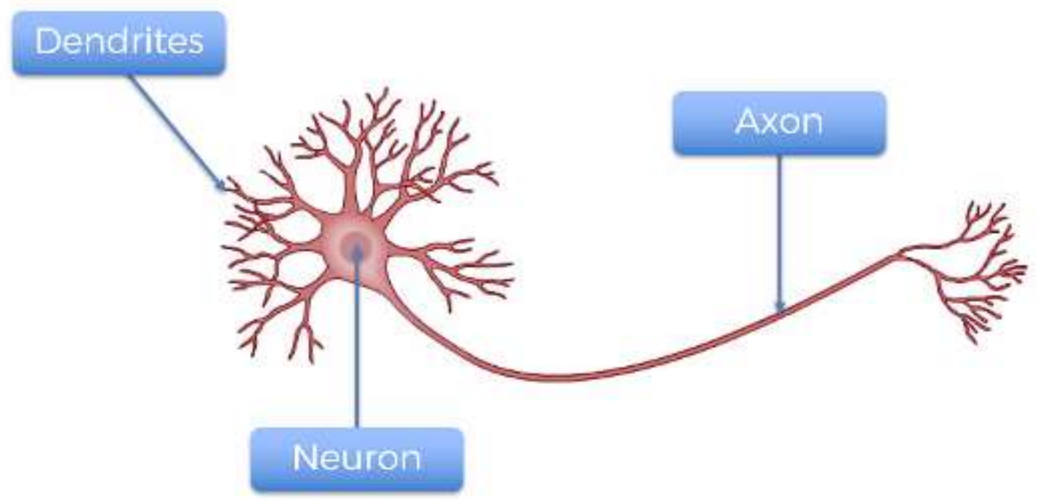
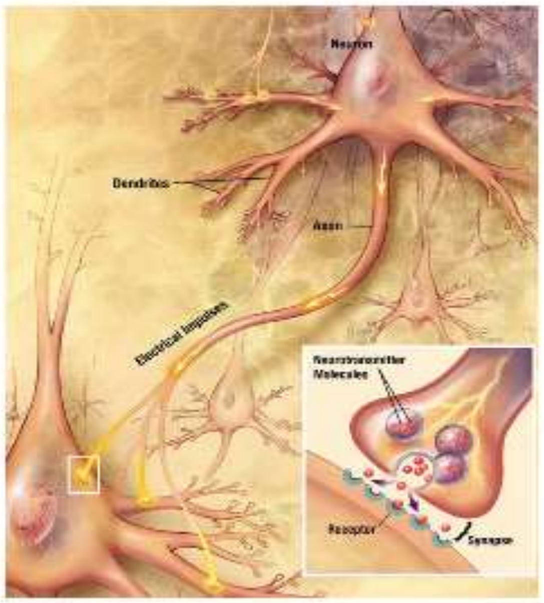
Artificial Neural Network : The Neuron





The Perceptron

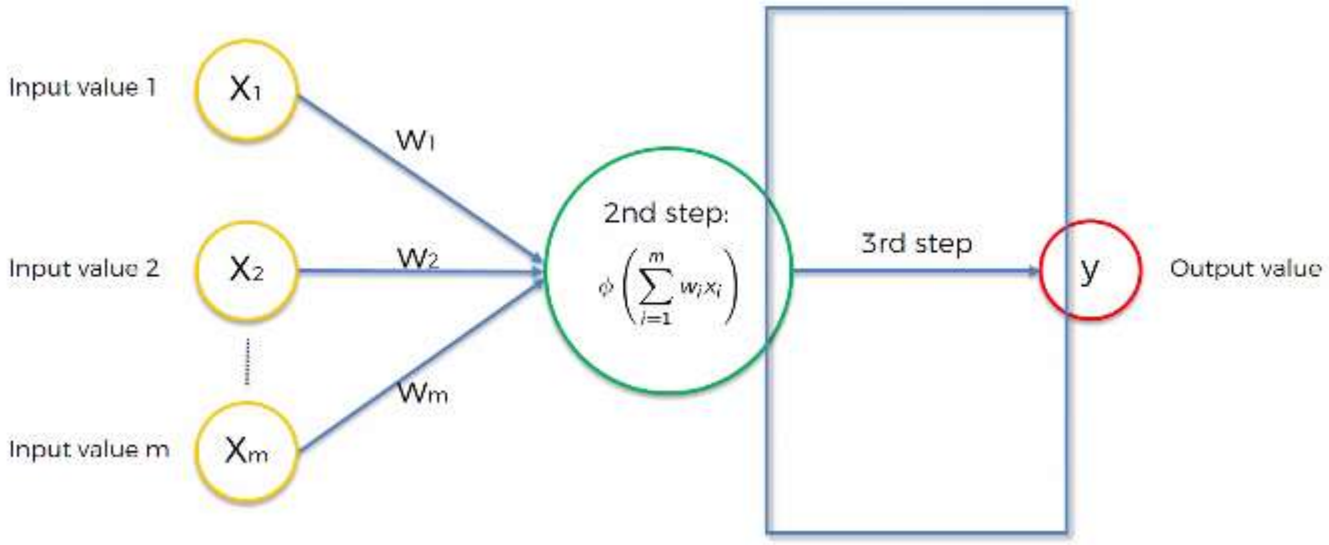
- The perceptron is a mathematical model of a biological neuron.
- While in actual neurons the dendrite receives electrical signals from the axons of other neurons, in the perceptron these electrical signals are represented as numerical values.
- At the synapses between the dendrite and axons, electrical signals are modulated in various amounts. This is also modelled in the perceptron by multiplying each input value by a value called the weight.
- An actual neuron fires an output signal only when the total strength of the input signals exceed a certain threshold. We model this phenomenon in a perceptron by calculating the weighted sum of the inputs to represent the total strength of the input signals, and applying a step function on the sum to determine its output.

Activation Function

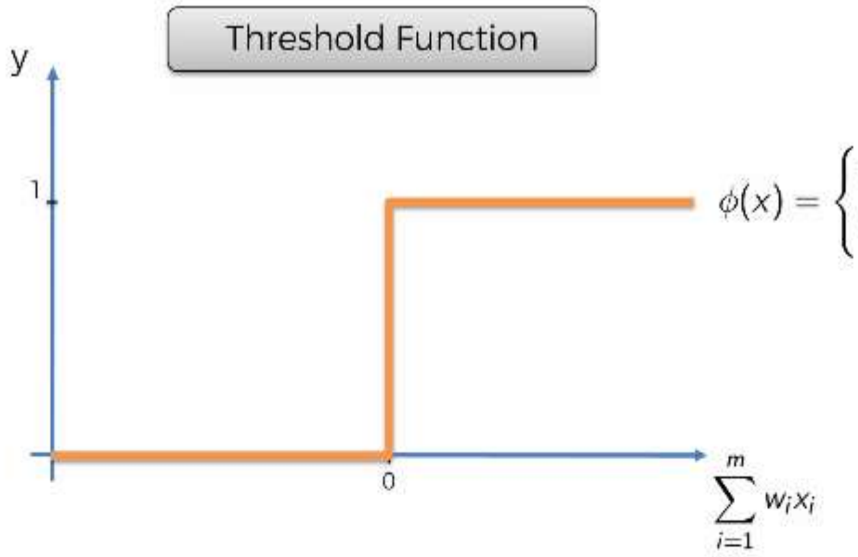
An activation function is a function that is added into an artificial neural network in order to help the **network learn complex patterns in the data**.

When comparing with a neuron-based model that is in our brains, the activation function is at the end deciding **what is to be fired to the next neuron**. That is exactly what an activation function does in an ANN as well.

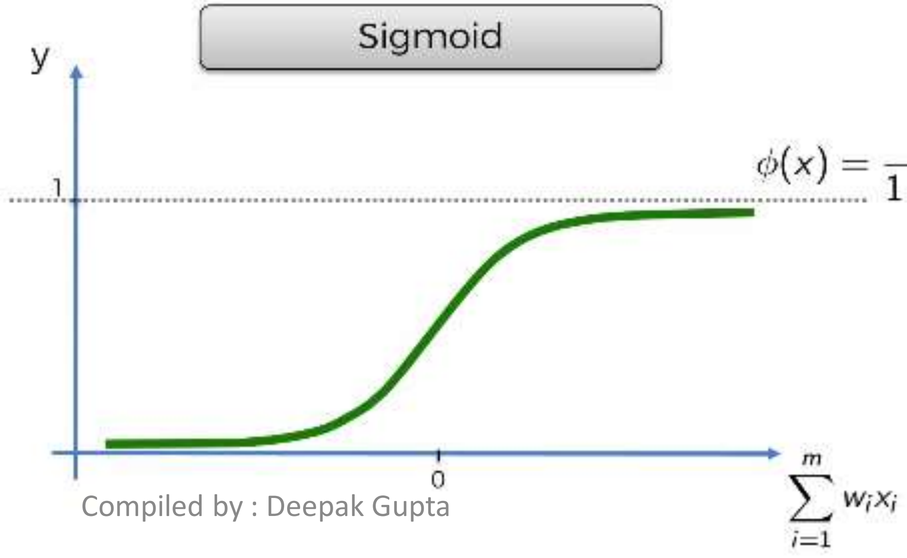
It takes in the output signal from the previous cell and converts it into some form that can be taken as input to the next cell.



Types of Activation Function

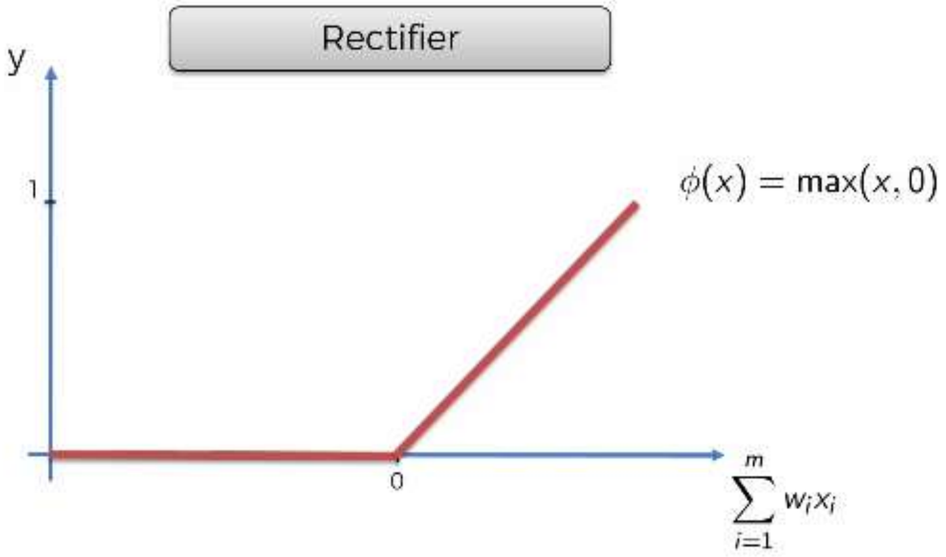


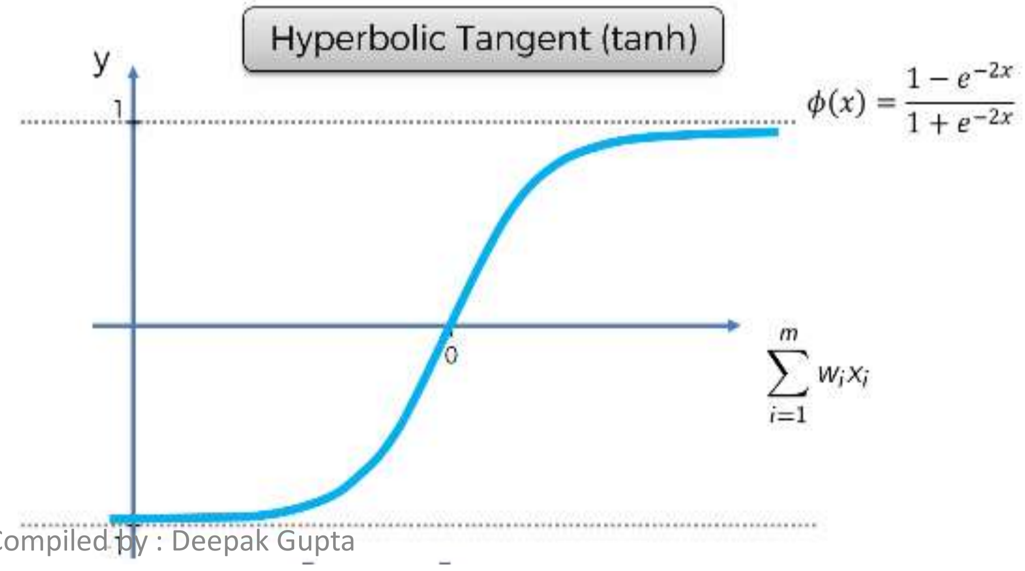
$$\phi(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$$



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

Types of Activation Function





Learning And Generalization

Let us suppose that a sufficiently large set of examples (the *training set)* is available. Two main learning strategies can be adopted.

- If the target output values - the desired answers - of the network are known for all the input patterns of the training set, a *supervised learning* strategy can be applied.

- In supervised learning the network's answer to each input pattern is directly compared with the known desired answer, and a feedback is given to the network to correct possible errors.

- In other cases, the target answer of the network is unknown. Thus the *unsupervised learning* strategy teaches the network to discover by itself correlations and similarities among the input patterns of the training set and, based on that, to group them in different clusters.

- In unsupervised learning, there is no feedback from the environment to say what the answer should be or whether it is correct.

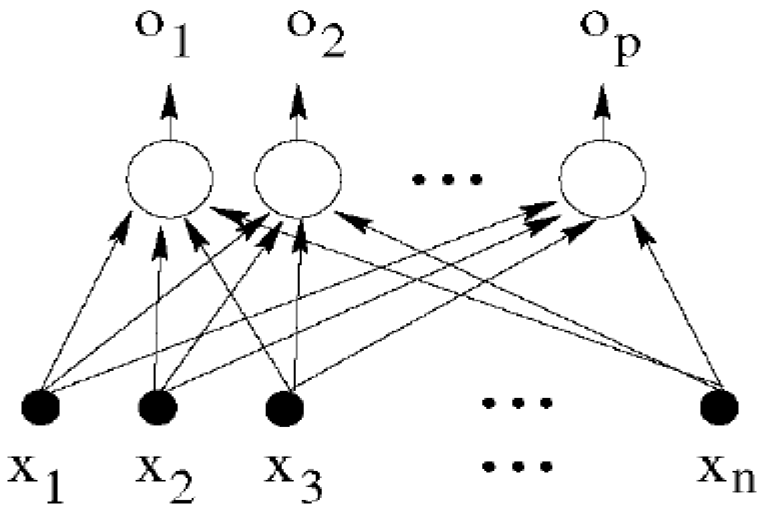
A Perceptron

- A simple neural network architecture allows only unidirectional forward connections
- among neurons and, because of that, it is called *feed-forward* neural network.

- The simplest type of feed-forward neural network, the *Perceptron*, consists of only one layer *of p* neural units connected with a set of *n* input terminals.

- It is a common convention not to include the set of input terminals in the count of the network layers, because they do not play any active role in the information processing.

- The number *p* of outputs is the same as the number of neural units.



A Perceptron

A Perceptron

In a Perceptron each output o_i is an explicit function of the input vector $\boldsymbol{x} = [x_1, \ldots, x_n]^T$, that can be straightforward calculated after propagating the input values through the network as in eq. 8.7.

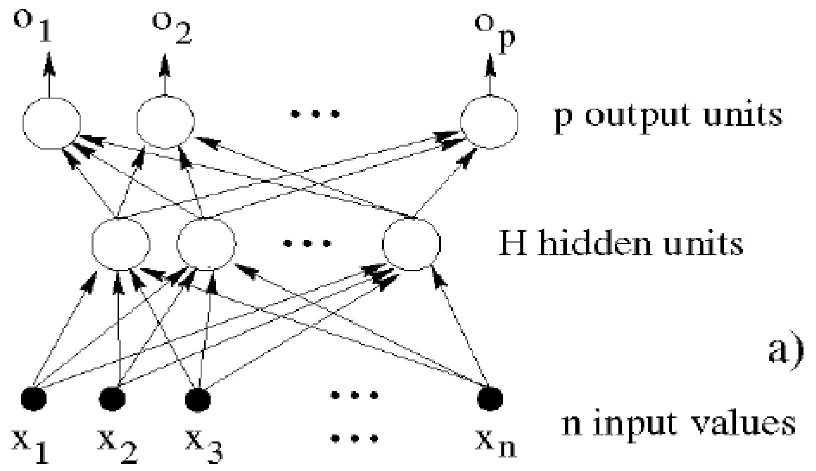
$$o_i = f(a_i) = f\left(\sum_{k=0}^{n} w_{ik}\, x_k\right) \qquad i = 1, \ldots, p$$

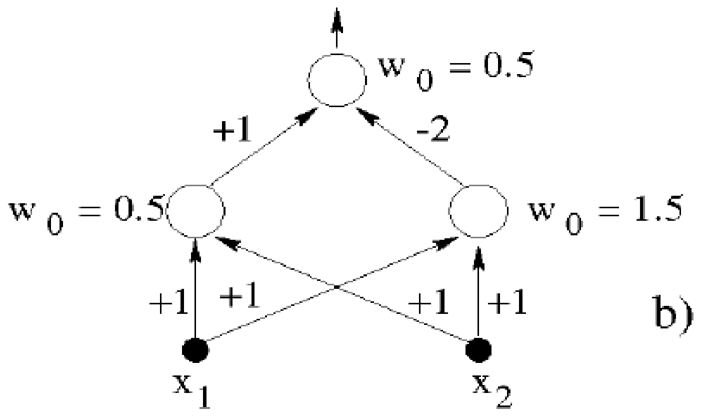
a_i indicates the net input to neuron i and w_{ik} the weight connecting the i-th output unit o_i with the k-th input value x_k, being $x_k \in [0,1]$, $k = 1, \ldots, n$, and $o_i \in [0,1]$, $i = 1, \ldots, p$. Considering a virtual input parameter x_0 permanently set to $+1$, the threshold value w_{i0} of the activation function of neuron i has been inserted into the sum.

Multilayer Feedforward Neural Networks

Multilayer Perceptron

- Networks with more than one layer of artificial neurons, where only forward connections from the input towards the output are allowed, are called *Multi- Layer Perceptron (MLP)* or ***Multilayer Feedforward Neural Networks***.
- Each MLP consists of a set of input terminals, an output neural layer, and a number of layers of ***hidden*** neural units between the input terminals and the output layer.
- The easiest way of transforming the input vector probably consists of introducing one or more layers of artificial neurons in the Perceptron architecture, so that the first layer of neurons pre-processes the input space and the second layer builds up the discrimination surfaces necessary to solve the problem.





Multilayer Feedforward Neural Networks

Let us suppose that we are dealing with a fully connected multilayer Perceptron with L layers, $l = 0, 1, \ldots, L$, with $l = 0$ denoting the set of input terminals, and $l = L$ the output layer. Each layer l has $n(l)$ neurons. The output value o_i of each unit i in layer l can still be calculated by means of eq. , where inputs x_k to unit i correspond to outputs o_k in layer $l - 1$. Thus eq. becomes:

$$o_i = f_i(a_i) = f_i\left(\sum_{k=0}^{n(l-1)} w_{ik}\ o_k\right) \qquad i = 1, \ldots, n(l)$$

where a_i represents the net input to unit i, $f_i()$ its activation function and $n(l-1)$ the number of units in layer $l-1$. If layers $l-1$ and l are only partially connected, the sum in eq. 8.19 has to be changed to cover only the units k in layer $l - 1$ that feed unit i in layer l.

Limitation of Multilayer Feedforward Neural Networks

- Even though multilayer feedforward neural networks offer great computational
- promises, the target patterns of the hidden layer is unknown.

- If an output unit produces an incorrect answer for a given input vector, it is impossible to detect which of the hidden units makes a mistake and consequently which weights should be changed.

- **Solution:**

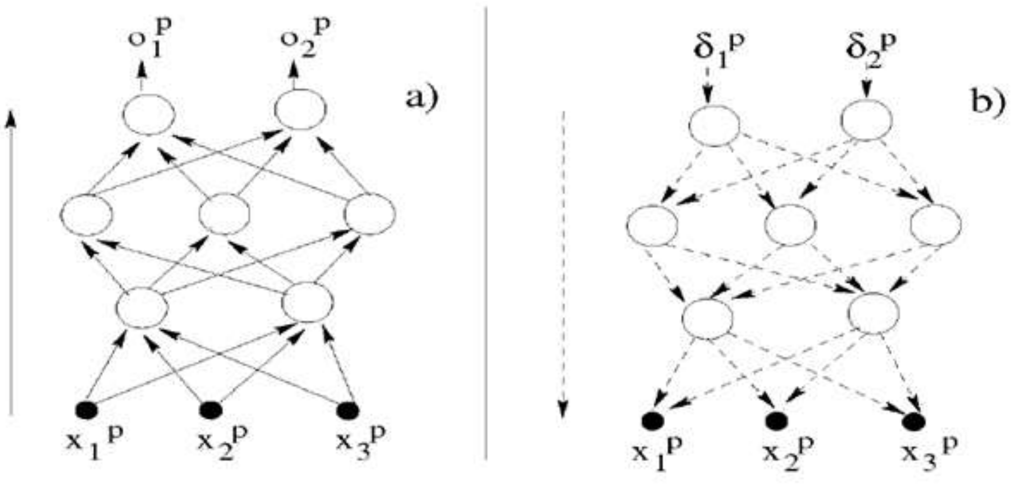
- **Back Propagation Learning Algorithm : Here,** It is not necessary to know the target pattern of the hidden layer. If the error function and the activation functions of the network units are differentiable, the gradient descent strategy can still be applied in order to find the network's weight configuration *W** that minimizes the error.

Back-Propagation Learning Algorithm

The algorithm that implements the gradient descent learning strategy for multilayer feedforward neural networks, independently reinvented multiple times, is known as *Back-Propagation,* because it consists of two steps:

1. forward propagation of the input pattern from the input to the output layer of the network;
2. back-propagation of the error vector from the output to the input layer of the network.

To apply the gradient descent strategy for the minimization of the error function, an arbitrary differentiable error function can be chosen.



a) Forward propagation of the input vector to calculate the error and b) back propagation of the error vector to update the weight matrix.

Back-Propagation Learning Algorithm

The Back-Propagation procedure can be summarized as follows.

1. Initialize the weights w_{ik} with random values.
2. Apply training pattern ***x***^{q} to the input layer.
3. Propagate $\boldsymbol{x}^q$ forward from the input terminals to the output layer.
4. Calculate error ***E*^{q}*{W)*** on the output layer.

$$E^q(\boldsymbol{W}) = \frac{1}{2}\sum_{i=1}^{p} (o_i^q - d_i^q)^2$$

5. Compute the (5's of the output layer as:

$$\delta_i^q = \frac{\partial E^q}{\partial a_i^q} = f'(a_i^q)\,(o_i^q - d_i^q) \qquad i \in \text{ output layer } L$$

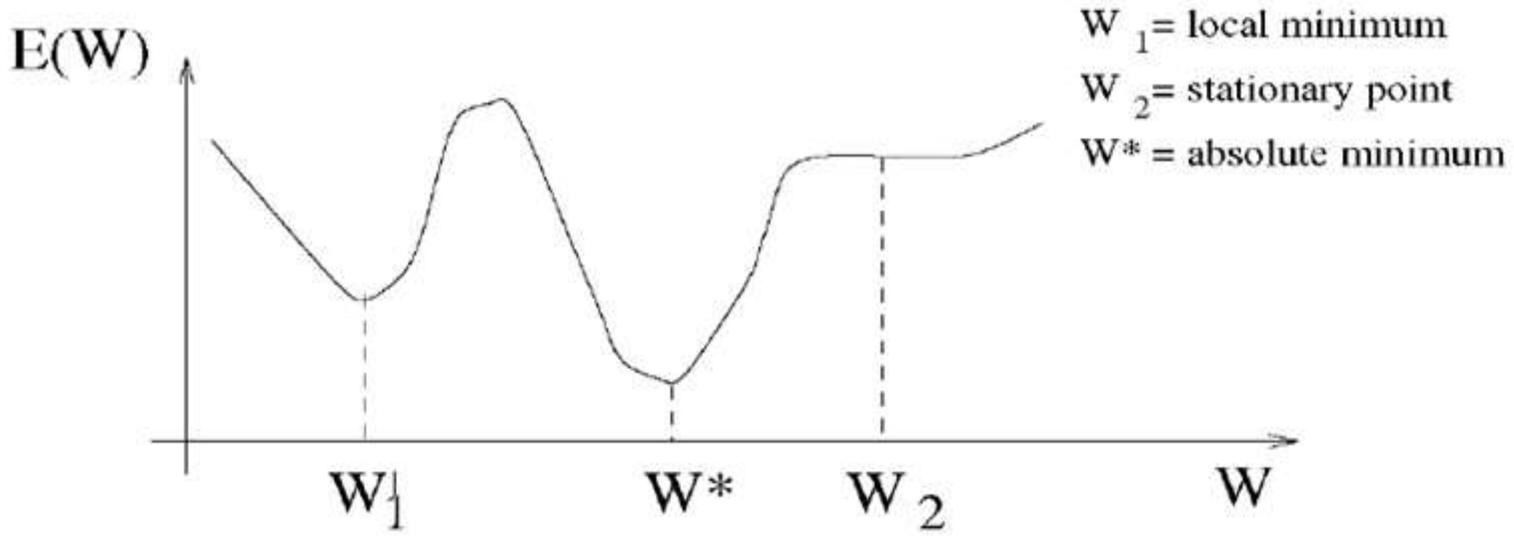
6. Compute the *d's* of the preceding layers, by propagating the *d's* backward

$$\delta_i^q = f'(a_i^q) \sum_{j=1}^{n(l+1)} w_{ji}\,\delta_j^q \qquad \begin{array}{l} i \in \text{ layer } l < L \\ j \in \text{ layer } l+1 \end{array}$$

7. Use $\Delta w_{ik}^q = -\eta\,\delta_i^q o_k^q$, where $i \in$ layer l and $k \in$ layer $l-1$, for all w_{ik} of the network.
8. $q \rightarrow q+1$ and go to 2.

Local Minima

The Back-Propagation learning algorithm is supposed to converge to the optimal weight configuration $\boldsymbol{W}^*$, that represents the location of the absolute minimum on the error surface $E(\boldsymbol{W})$, $\{\boldsymbol{W}^* : \ E(\boldsymbol{W}^*) = \min_{\boldsymbol{W}} \ E(\boldsymbol{W})\}$.



The general error surface as a function of the weight matrix W

Generalization

- The main reason for using an ANN is most often not to memorize the examples of the training set, but to build a general model of the input/output relationships based on the training examples.

- A general model means that the set of input/output relationships, derived from the training set, apply equally well to new sets of data from the same problem not included in the training set. The main goal of a neural network is thus the ***generalization*** to new data of the relationships learned on the training set.

- The composition of the training set represents a key point for the generalization property. The examples included in the training set have to fully and accurately describe those rules the network is supposed to learn. This means a sufficient number of clean and correctly labelled data equally distributed across the output classes. In practice, the data available for training are usually noisy, partly incorrectly labelled, and do not describe exhaustively all the particular relationships among the input patterns

Generalization and Overfitting

- A too large number of parameters can memorize all the examples of the training set with the associated noise, errors, and inconsistencies, and therefore perform a poor generalization on new data. This phenomenon is known as *overfitting*.

- In Figure, the dotted line represents the underlying function to learn, the circles the training set data, and the continuous line the estimated function by an oversized parametric model. Overfitting is thought to happen when the model has more degrees of freedom (roughly the number of weights in an ANN) than constraints (roughly the number of independent training examples). On the contrary, as in every approximation technique, a too small number of model parameters can prevent the error function from reaching satisfying values and the model from learning.

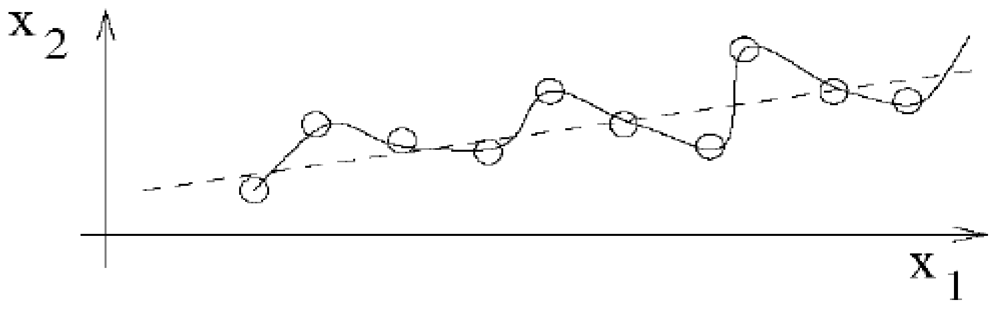
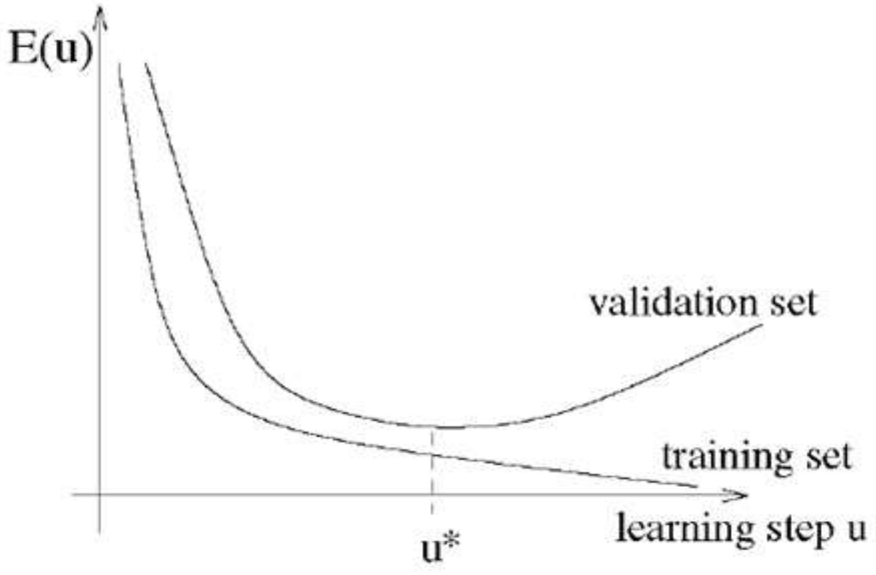


Fig. 8.9. An Example of Overfitting.

Generalization and Overfitting

- How can a good trade-off between learning and generalization be reached?

- Solution: One of the most common techniques to avoid overfitting in neural networks employs a small set of independent data, called ***validation set***, together with the training set.



The error plot on the training set and on the validation set as a function of the learning step u.

Generalization and Overfitting

- During the learning procedure, the error shows at first a decrease on the validation set as well as on the training set, as the network generalizes from the training data to the underlying input/output function.

- After some training steps *u**, usually in the later stages of learning, the network starts overfitting the spurious and misleading features of the training set. Consequently the error on the validation set increases, while the error on the training set keeps decreasing.

- The learning procedure can therefore be stopped at this step *u** with the smallest error on the validation set, because at this point the network is expected to yield the best generalization performance.

- This technique, known also as ***early stopping,*** however, may not be practical when only a small amount of data is available, since a requirement is that the validation set can not be used for training.

- Another approach, called ***training with noise***, adds noise to each training input pattern, to smooth out the training data so that overfitting does not occur. A new random vector is added to each input pattern before it feeds the network. For the network it becomes more and more difficult to fit individual data points precisely.

Generalization: Network Dimension

- A rule of thumb for obtaining good generalization is to use the smallest network that fits the training data. A small network, besides the better expected generalization, is also faster to train.

- Unfortunately, the smallest size of a MLP for a given task is not known a priori.

- What is the smallest number of hidden layers and units per layer necessary to approximate a particular set of functions with a given accuracy?

- Regarding the number of layers it was proven that with at most two hidden layers with a *sufficient number of units* it is possible to approximate any function with arbitrary accuracy. If the function is continuous, only one hidden layer is sufficient for this purpose.

- Even though in theory a two-layer neural network is able to approximate any function, in practice it is always advisable to perform an adequate pre-processing of the data for the best information representation. A bad representation of the data features could lead to a very complicated error surface with many local minima and therefore to a very difficult learning process. Similarly the outputs of a neural network are often post-processed to produce more significant output values.

Generalization: Network Dimension

Two kinds of iterative techniques have been proposed to optimize the network size.

- **Growing Algorithms.** This group of algorithms begins with training a relatively small neural architecture and allows new units and connections to be added during the training process, when necessary. Three growing algorithms are commonly applied:

  a) **upstart algorithm, b) tiling algorithm, c) cascade correlation.**

  The first two apply to binary input/output variables and networks with step activation function. The third one, which is applicable to problems with continuous input/output variables and with units with sigmoidal activation function, keeps adding units into the hidden layer until a satisfying error value is reached on the training set.

- **Pruning Algorithms.** Unlike the growing algorithm the general pruning approach consists of training a relatively large network and gradually removing either weights or complete units that seem not to be necessary. The large initial size allows the network to learn quickly and with a lower sensitivity to initial conditions and local minima. The reduced final size helps to improve generalization.

Generalization: Network Dimension

There are basically two ways of reducing the size of the original network(pruning).

1. ***Sensitivity methods.*** After learning the sensitivity of the error function to the removal of every element (unit or weight) is estimated: the element with the least effect can be removed.

   In general, sensitivity methods modify a trained network: the network is trained, the sensitivity of each parameter is estimated, and the weights or nodes are removed consequently.

2. ***Penalty-term methods***. Weight decay terms are added to the error function, to reward the network for choosing efficient solutions. That is networks with Neural Networks small weight values are privileged. At the end of the learning process, the weights with smallest values can be removed, but, even in case they are not, a network with several weights close to 0 already acts as a smaller system.

   Penalty-term methods modify the error function so that the Back-Propagation algorithm drives unnecessary weights to zero and, in effect, removes them during training.

Unsupervised Learning

- In this case the training set *T* consists only of examples of input patterns x^q without a corresponding target answer, that is $T = \{x^q\}$. Without a target answer to learn, the neural network can only discover by itself typical patterns, regularities, clusters, or any other relationships of interest inside the training set, without using any feedback from the environment. This kind of learning is called *unsupervised*.

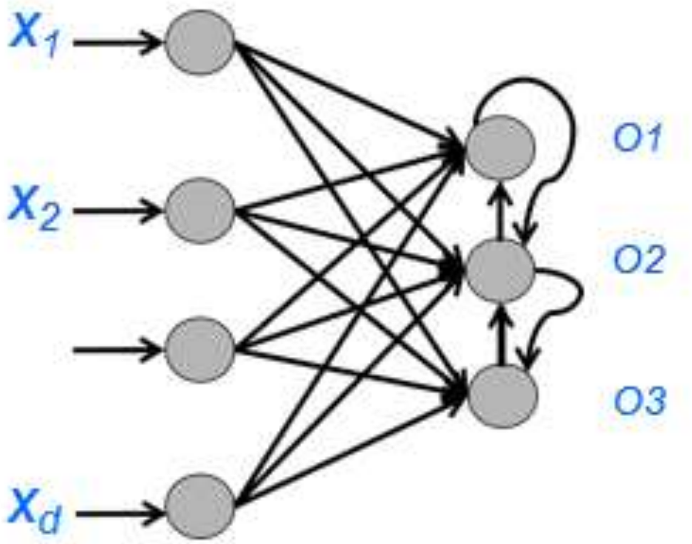
- In order to learn without a teacher, neural networks need some *self-organization* property, that keeps track of previously seen patterns and answers on the basis of some *familiarity* criterion to the current input. The input patterns are grouped into clusters on the basis of each others' similarity and independently from the external environment.

- It is important to notice that unsupervised learning can happen only if there is *redundancy* in the data, because without an external feedback only redundancy can provide knowledge about the input space properties.

- Two main philosophies lead the unsupervised learning approaches:
  - ***Competitive learning***: In the first case the resulting networks are mainly oriented to clustering or classification,
  - ***Hebbian learning:*** This is more oriented to measure familiarity or to project the input data onto their principal components.

Competitive Learning

- **A form of unsupervised training where output units are said to be in competition for input patterns.**
  - During training, the output unit that provides the highest activation to a given input pattern is declared the weights of the winner and is moved closer to the input pattern, whereas the rest of the neurons are left unchanged.
  - This strategy is also called winner-take-all since only the winning neuron is updated.
  - Output units may have lateral inhibitory connections so that a winner neuron can inhibit others by an amount proportional to its activation level



Competitive Learning

The standard competitive learning process is the *winner-take-all* strategy. In a one-layer architecture, the neural units compete to be the winner for a given input pattern and only the winner unit is allowed to fire.

Let us consider a single layer of *p* units, fully connected to the *n* input terminals *Xj* by excitatory feedforward connections, *Wij* > 0. Each unit *i* will receive an input value Oj as:

$$a_i = \sum_{j=1}^{n} w_{ij} x_j = \boldsymbol{w}_i^T \boldsymbol{x}$$

The neural unit with highest input value ai (the winner) will be the one to fire, that is with output Oj = 1.

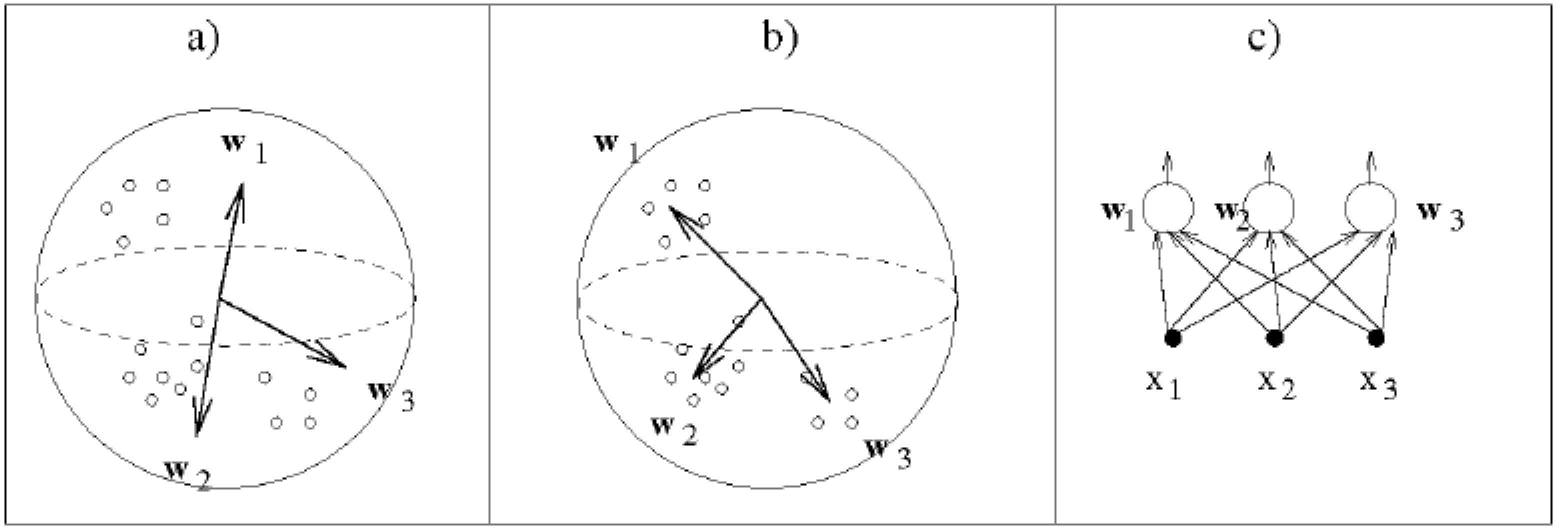
$$\begin{cases} o_i = 1 & \text{if: } \boldsymbol{w}_i^T \boldsymbol{x} = \max_{k=1,\ldots,p} (\boldsymbol{w}_k^T \boldsymbol{x}) \\ o_i = 0 & \text{otherwise} \end{cases}$$

If all the weight vectors are normalized so that ‖ *Wi* ‖= 1, the winner-take-all condition can also be expressed as:

$$\begin{cases} o_i = 1 & \text{if: } \| \boldsymbol{w}_i - \boldsymbol{x} \| = \min_{k=1,\ldots,p} \| \boldsymbol{w}_k - \boldsymbol{x} \| \\ o_i = 0 & \text{otherwise} \end{cases}$$

Competitive Learning

Previously defined conditions reward the unit *i* with weight vector *Wi* closest to the input vector *x* as the winning unit. Thus in a winner-take-all strategy, each neuron represents a group of input patterns by means of its weight vector *Wi*. The task of the learning algorithm is to choose such weight vectors as representative prototypes of data clusters in the input space.



a) The weight vectors at the beginning and
b) at the end of the training process
c) a single layer neural structure trained with the standard competitive learning rule.

Competitive Learning: Self Organizing Maps

- A **self-organizing map** (**SOM**) or **self-organizing feature map** (**SOFM**) is a type of artificial neural network (ANN) that is trained using unsupervised learning to produce a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples, called a **map**, and is therefore a method to do dimensionality reduction.

- Self-organizing maps differ from other artificial neural networks as they apply competitive learning as opposed to error-correction learning (such as backpropagation with gradient descent or activation function), and in the sense that they use a neighbourhood function to preserve the topological properties(adjusting weights) of the input space.

Competitive Learning: Self Organizing Maps

SOM's architecture :

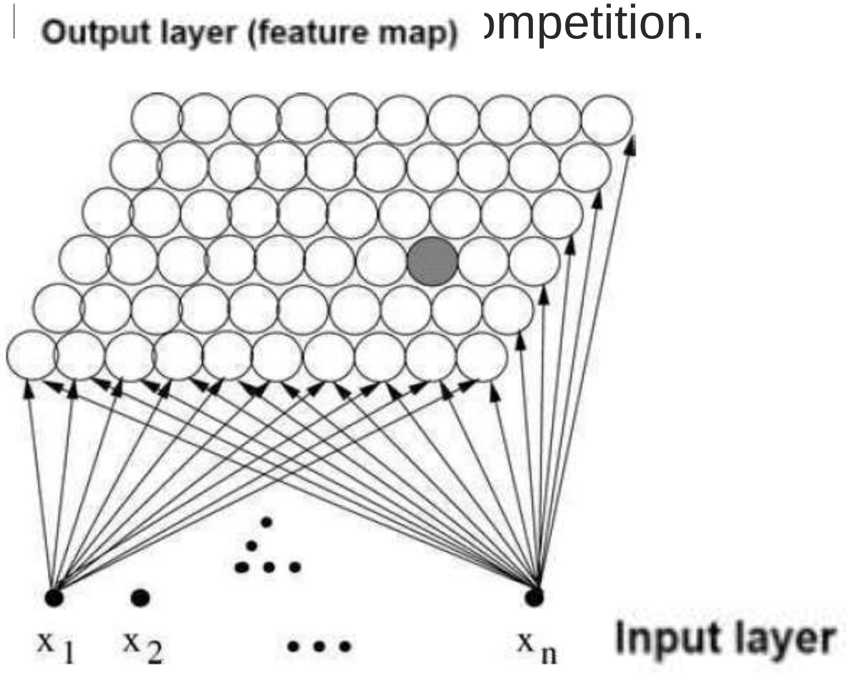
- Self organizing maps have two layers, the first one is the input layer and the second one is the output layer or the feature map.
- Unlike other ANN types, SOM doesn't have in neurons, we directly pass weights to output layer without doing anything.
- Each neuron in a SOM is assigned a weight vector with the same dimensionality d as the input space.

- As SOM doesn't use backpropagation with SGD to update weights, this type of unsupervised artificial neural network uses competitive learning to update its weights, which is based on three processes :
  a) Competition
  b) Cooperation
  c) Adaptation

Competitive Learning: Self Organizing Maps

a) **Competition**

Each neuron in a SOM is assigned a weight vector with the same dimensionality as the input space. In the example below, in each neuron of the output layer we will have a vector with dimension n. We compute distance between each neuron (neuron from the output layer) and the input data, and the neuron with the lowest distance will ompetition.

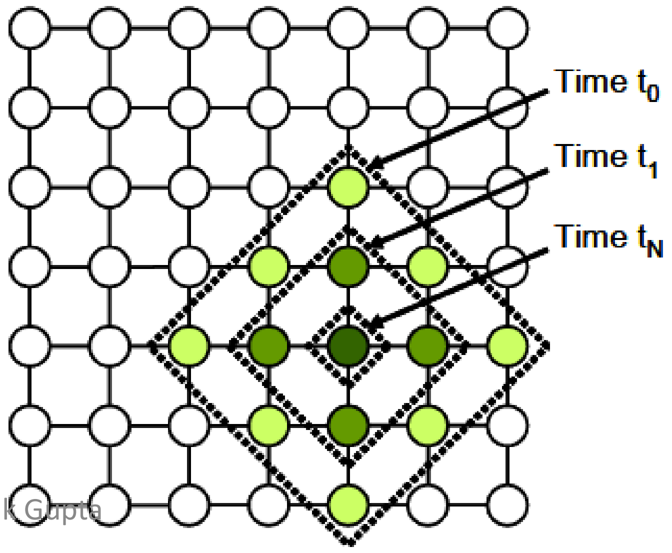


. A two-dimensional Kohonen map, with the winning unit in gray.

Competitive Learning: Self Organizing Maps

b) **Cooperation**

- **The activation of the winning neuron is spread to neurons in its immediate neighbourhood.** This allows topologically close neurons to become sensitive to similar patterns.
- **The winner's neighbourhood is determined on the lattice topology :** To choose neighbours we use neighbourhood kernel function, this function depends on two factor : time (time incremented each new input data) and distance between the winner neuron and the other neuron (How far is the neuron from the winner neuron).
- **The size of the neighbourhood is initially large, but shrinks over time :** An initially large neighborhood promotes a topology-preserving mapping. Smaller neighborhoods allows neurons to specialize in the latter stages of training. The image below show us how the winner neuron's (The most green one in the center) neighbors are chosen depending on distance and time factors.



Competitive Learning: Self Organizing Maps

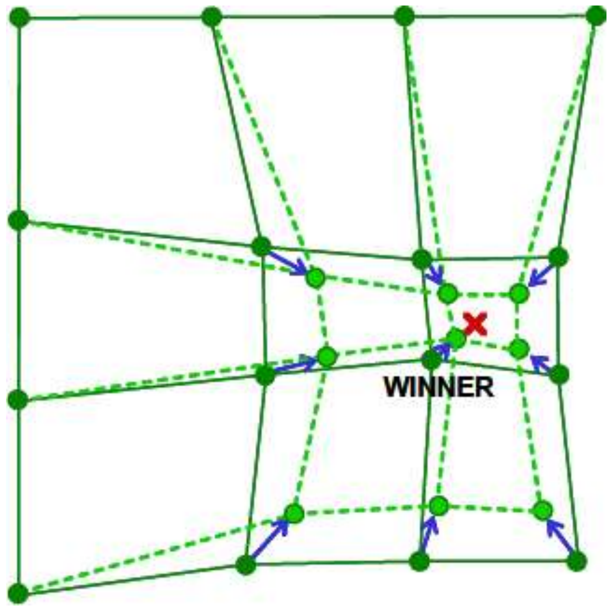
c) Adaptation: During training, the winner neuron and its topological neighbours are adapted to make their weight vectors more similar to the input pattern that caused the activation.

After choosing the winner neuron and it's neighbours we compute neurons update. Those chosen neurons will be updated but not the same update, more the distance between neuron and the input data grow less we adjust it like shown in the image below :

Neurons that are closer to winner will adapt more heavily than the neurons which are farther Away.

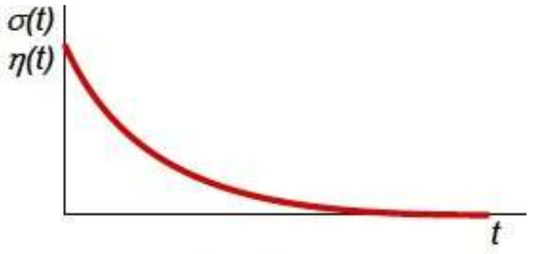
The magnitude of the adaptation is controlled with a learning rate, which decays over time to ensure convergence of the SOM. This learning rate indicates how much we want to adjust our weights.

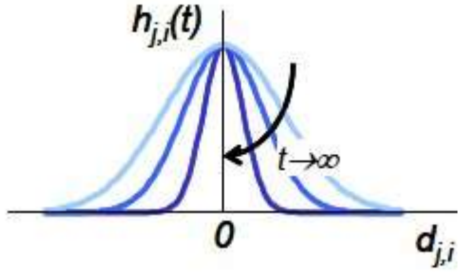
After time t (positive infinite), this learning rate will converge to zero so we will have no update even for the neuron winner .



SOM Algorithm

- **Define**
  - A learning rate decay rule $\eta(t) = \eta_0 \exp\left(-\frac{t}{\tau_1}\right)$
  - A neighborhood kernel function $h_{ik}(t) = \exp\left(-\frac{d_{ik}^2}{2\sigma^2(t)}\right)$
    - where d_{ik} is the lattice distance between w_i and w_k
  - A neighborhood size decay rule $\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\tau_2}\right)$





```
1.  Initialize weights to some small, random values
2.  Repeat until convergence
    2a. Select the next input pattern x^(n from the database
        2a1. Find the unit w_i that best matches the input pattern x^(n
                 $i(x^{(n}) = \arg\min_j \|x^{(n} - w_j\|$
        2a2. Update the weights of the winner w_i and all its neighbors w_k
                 $w_k = w_k + \eta(t) \cdot h_{ik}(t) \cdot (x^{(n} - w_k)$
    2b. Decrease the learning rate η(t)
    2c. Decrease neighborhood size σ(t)
```

Dimensionality Reduction

- ***Reducing*** the ***dimension*** of the feature space is called "***dimensionality reduction***." There are many ways to achieve dimensionality reduction, but most of these techniques fall into one of two classes:
  - •Feature Elimination
  - •Feature Extraction

- **Feature elimination**: In feature elimination we reduce the feature space by eliminating features.
  - •Advantages of feature elimination methods include simplicity and maintaining interpretability of your variables.
  - •As a disadvantage, though, you gain no information from those variables you've dropped.
  - •By eliminating features, we've also entirely eliminated any benefits those dropped variables would bring.

- **Feature extraction**: In feature extraction, we create "new" independent variables, where each "new" independent variable is a combination of each of "old" independent variables. However, we create these new independent variables in a specific way and order these new variables by how well they predict our dependent variable.

Principle Component Analysis

- Principal components analysis (PCA) is a statistical technique that allows identifying underlying linear patterns in a data set so it can be expressed in terms of other data set of lower dimension without much loss of information.

- The final data set should explain most of the variance of the original data set by reducing the number of variables. The final variables will be named as principal components.

- The idea behind PCA is simply to find a low-dimension set of axes that summarize data.

- PCA doesn't selects some features out of the dataset and discards others. The algorithm actually constructs new set of properties based on combination of the old ones.

- Mathematically speaking, PCA is an orthogonal transformation of a set of (possibly) correlated variables into a set of linearly uncorrelated ones, and the uncorrelated (pseudo-) variables, called principal components (PCs), are linear combinations of the original input variables.

Principle Component Analysis

- This orthogonal transformation is performed such that the first principal component has the greatest possible variance (variation within the dataset).

- This procedure is then followed for the second component, then the third component, etc. This means that each succeeding component in turn has the highest variance when it is orthogonal to the preceding components

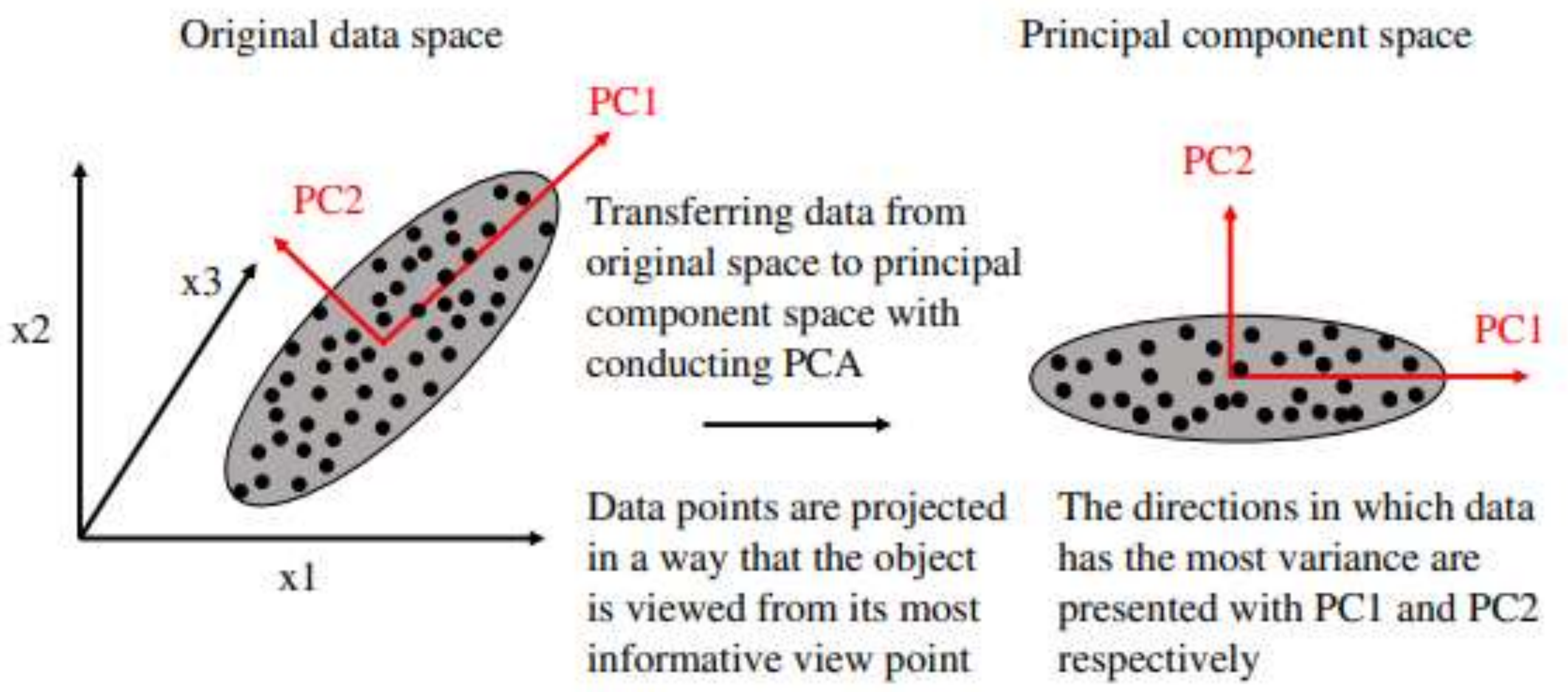
- The algorithm use the concepts of variance, covariance matrix, eigenvector and eigenvalues pairs to perform PCA, providing a set of eigenvectors and its respectively eigenvalues as a result.

- Eigenvectors represents the new set of axes of the principal component space and the eigenvalues carry the information of quantity of variance that each eigenvector have.

- Thus, in order to reduce the dimension of the dataset we are going to choose those eigenvectors that have more variance and discard those with less variance.

Principle Component Analysis

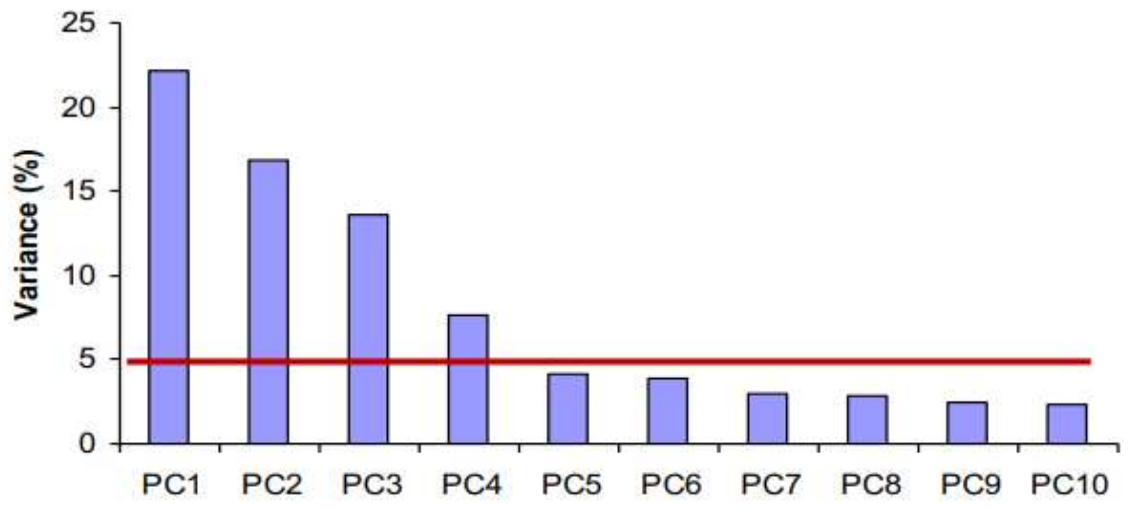
To help visualize the PCA transformation, a schematic dataset with three input variables is presented in Figure (left). As shown in this figure, in conducting PCA the data points are transferred from the original 3D original input space (on the left) to a 2D principal component space (on the right).


Original data space
Principal component space
PC1
PC2
x3
x2
x1
Transferring data from original space to principal component space with conducting PCA
PC2
PC1
Data points are projected in a way that the object is viewed from its most informative view point
The directions in which data has the most variance are presented with PC1 and PC2 respectively

Principle Component Analysis

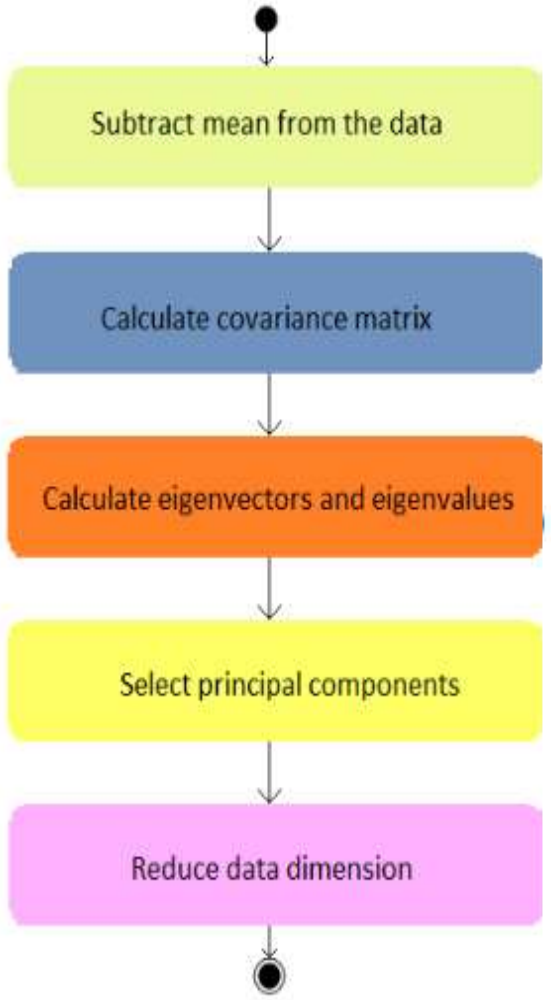
- For M original dimensions, sample covariance matrix is MxM, and has up to M eigenvectors. So M PCs.
- Where does dimensionality reduction come from?

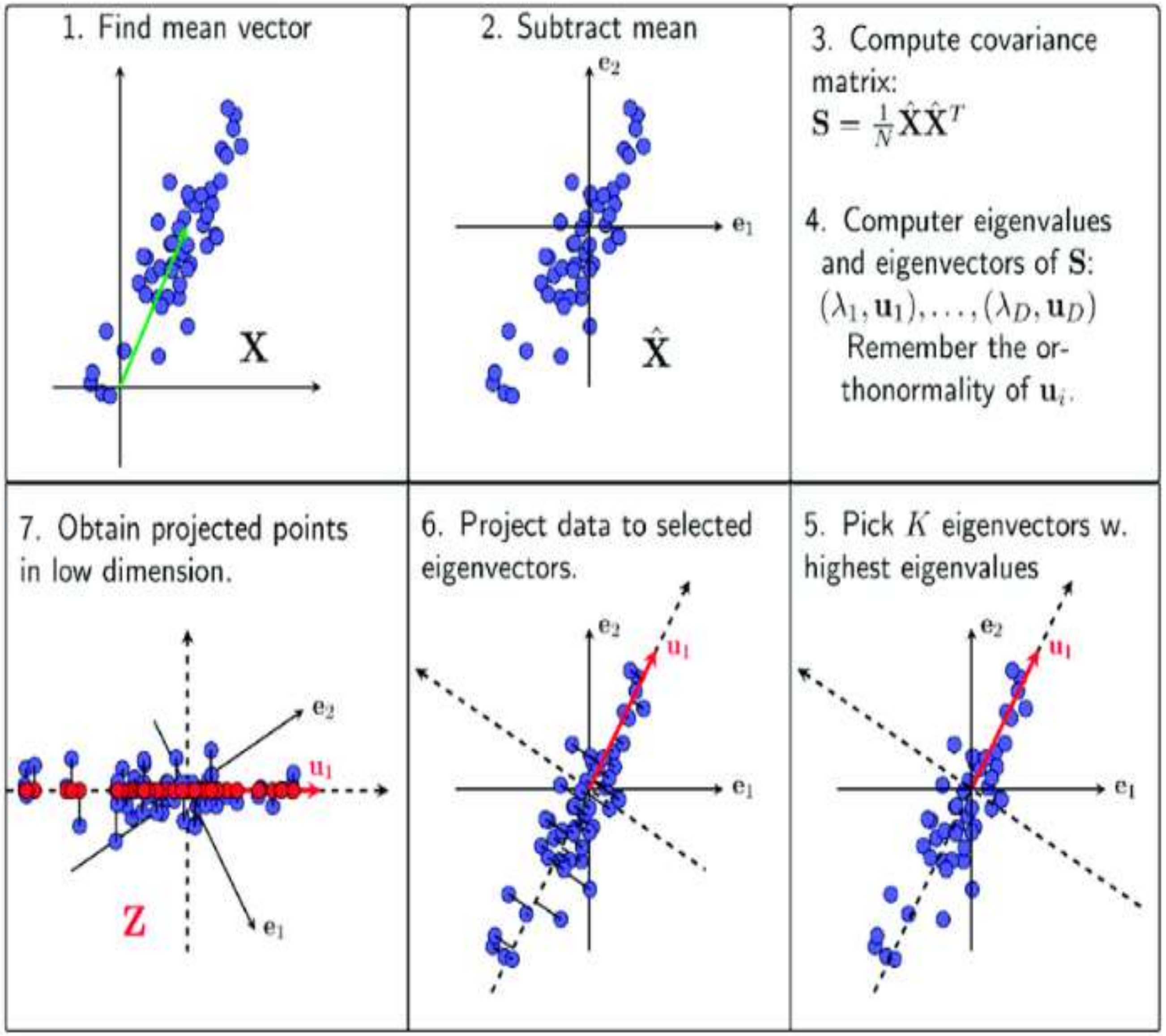
  Can *ignore the components of lesser significance*.



- You do lose some information, but if the eigenvalues are small, you don't lose much
  - M dimensions in original data
  - calculate M eigenvectors and eigenvalues
  - choose only the first D eigenvectors, based on their eigenvalues
  - final data set has only D dimensions

Principle Component Analysis


Subtract mean from the data
Calculate covariance matrix
Calculate eigenvectors and eigenvalues
Select principal components
Reduce data dimension


1. Find mean vector
$\mathbf{X}$
2. Subtract mean
$\mathbf{e}_2$
$\mathbf{e}_1$
$\hat{\mathbf{X}}$
3. Compute covariance matrix:
$\mathbf{S} = \frac{1}{N}\hat{\mathbf{X}}\hat{\mathbf{X}}^T$
4. Computer eigenvalues and eigenvectors of $\mathbf{S}$:
$(\lambda_1, \mathbf{u}_1), \ldots, (\lambda_D, \mathbf{u}_D)$
Remember the orthonormality of $\mathbf{u}_i$.
7. Obtain projected points in low dimension.
$\mathbf{Z}$
$\mathbf{u}_1$
6. Project data to selected eigenvectors.
5. Pick K eigenvectors w. highest eigenvalues