CSE-E4810 Machine Learning and Neural Networks (5 cr)

Lecture 2: Simple neural models

Prof. Juha Karhunen

https://mycourses.aalto.fi/course/view.php?id=13086

Introduction

- In this lecture, we first discuss the structure of a single neuron.
- Then we discuss briefly various neural network structures.
- And finally learning algorithms for a single neuron.
- Such models and learning algorithms are usually too simple for achieving appropriate performance.
- But single neurons and single-layer networks provide the foundation in building more complicated multilayer networks.
- Therefore it is useful to understand them well.

Basic models of artificial neurons

- A neuron is the fundamental information processing unit of a neural network.

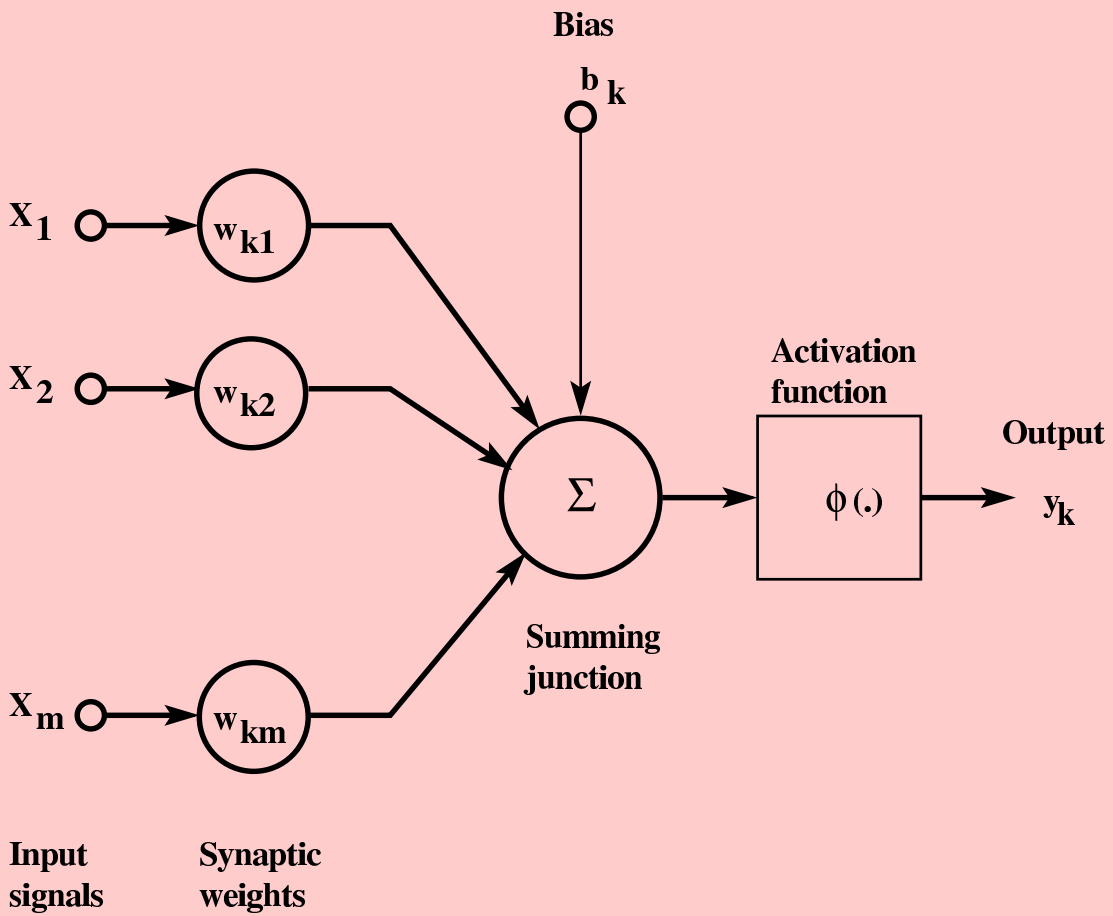

Figure 1: Basic model of an artificial neuron.

- The standard model of a single neuron is shown in Figure 1.
- The notations vary somewhat in different textbooks.
- The model in Figure 1 consists of four basic elements.
  1. A set of synapses or connecting links:
     - Characterized by synaptic weights (strengths).
     - Let x_j denote the jth component of the input vector $\mathbf{x}$.
     - It is the input signal to the synapse j.
     - When connected to neuron k, x_j is multiplied by the synaptic weight w_{kj}.
     - The weights are usually real numbers.
  2. A summing junction or linear combiner which sums the weighted inputs $w_{kj}x_j$.
  3. Typically a bias term b_k is added to this weighted sum.
     - In Du's and Swamy's book the bias term is denoted by $-\theta_k$.

- The weighted sum with the bias term is called the linear response of a neuron.

4. An activation function $\varphi(.)$:
   - Applied to the linear response of a neuron.
   - Typically a nonlinear function.
   - Called sometimes also squashing function.
   - Its value is the output y_k of the k:th neuron.

Mathematical equations describing neuron k

$$u_k = \sum_{j=1}^{m} w_{kj} x_j = \mathbf{w}_k^T \mathbf{x}, \tag{1}$$

$$y_k = \phi(u_k + b_k). \tag{2}$$

Here:

- $u_k = \mathbf{w}_k^T \mathbf{x}$ is the output of the linear combiner;

- $\phi(.)$ is the activation function;
- y_k is the output signal of the neuron k;
- $\mathbf{x} = [x_1, x_2, \ldots, x_m]^T$ is the column vector containing the m input signals;
- $\mathbf{w}_k = [w_{k1}, w_{k2}, \ldots, w_{km}]^T$ is the column vector containing the m synaptic weights of the k:th neuron.
- $b_k = -\theta_k$ is the bias term of the k:th neuron.
- Note that $\mathbf{w}_k^T \mathbf{x}$ is a hyperplane going through the origin $\mathbf{x} = \mathbf{0}$.
- In two-dimensional case it reduces to a straight line.
- The effect of the bias term $b_k = -\theta_k$ is to move this hyperplane away from the origin.

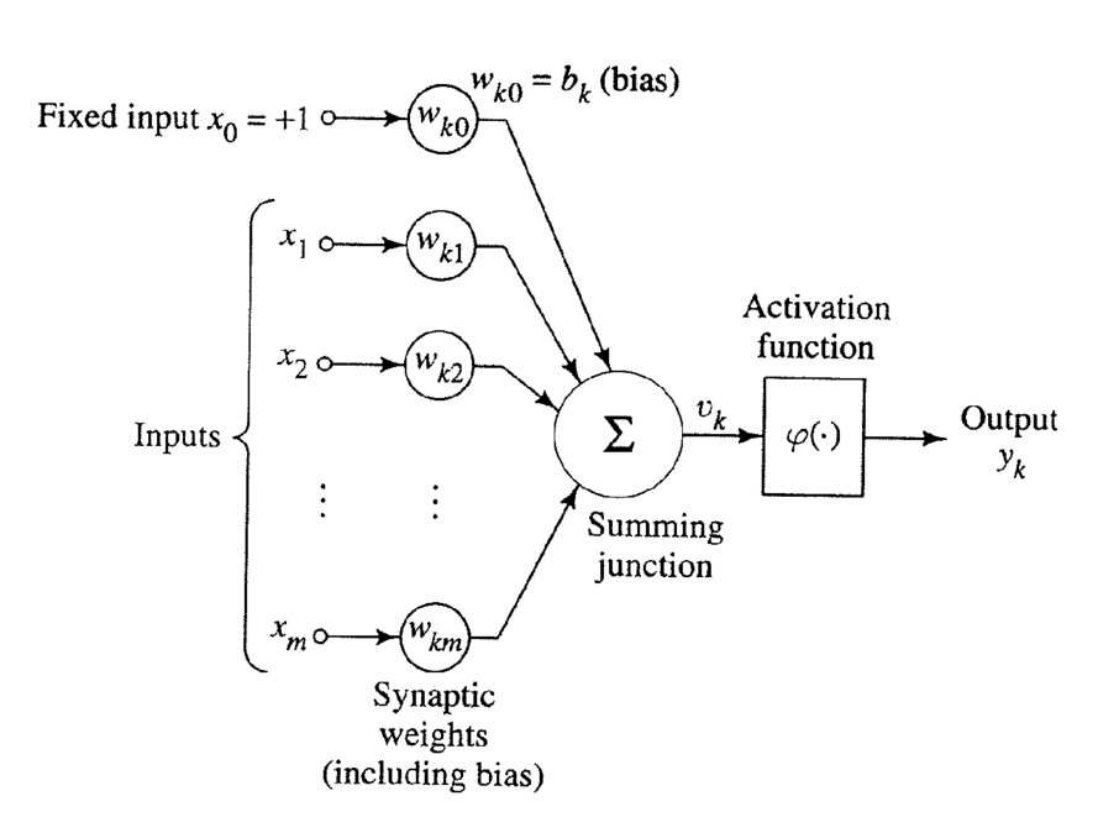

Figure 2: Alternative model of an artificial neuron.

- Figure 2 shows a slightly different alternative neuron model which is mathematically equivalent to the previous neuron model.
- An extra auxiliary zeroth synapse has been added to this model.
- The input to this synapse is constant $x_0 = +1$.
- The corresponding weight is the bias value; $w_{k0} = b_k$.
- The equations describing the neuron become now slightly simpler:

$$v_k = \sum_{j=0}^{m} w_{kj} x_j \tag{3}$$

$$y_k = \varphi(v_k) \tag{4}$$

- The sum in (3) can be expressed as the inner product $\mathbf{w}_k^T \mathbf{x}$ by adding these zero:th components to the vectors $\mathbf{w}_k$ and $\mathbf{x}$.

Activation functions

Linear activation function

- Activation function is the nonlinearity applied to the linear response $u_k + b_k$ or v_k of the k:th neuron.
- In various books it is denoted by $\phi(.)$, $\varphi(.)$, or $f(.)$.
- It yields the output $y_k = \phi(u_k + b_k) = \phi(v_k)$ of the k:th neuron.
- The simplest activation function is the linear one.
- It is simply $y = \phi(v) = v$.
- Linear activation functions are used in linear neurons and networks.
- However, neural networks get their representation power largely from the nonlinearities used in them.
- Linear neural networks suit well to linear processing only.

- But such linear problems can usually be solved much more efficiently and accurately using standard numerical methods.
- Examples are solving linear equations, eigenvalues, linear signal processing problems, etc.
- Therefore in this course we discuss only little linear networks.

Hard limiter activation function

- Figure 3 shows the hard limiter activation function $y = \phi(v)$.
- Sometimes it is called the threshold function.
- The hard limiter has just two values: $+1$ for $v > 0$ and 0 for $v < 0$.
- Using the bias b in $v = u + b$, the breakpoint can easily be moved to the point $u = -b$.
- In the symmetric version of the hard limiter, the output $y = \phi(v)$ is -1 for $v < 0$.

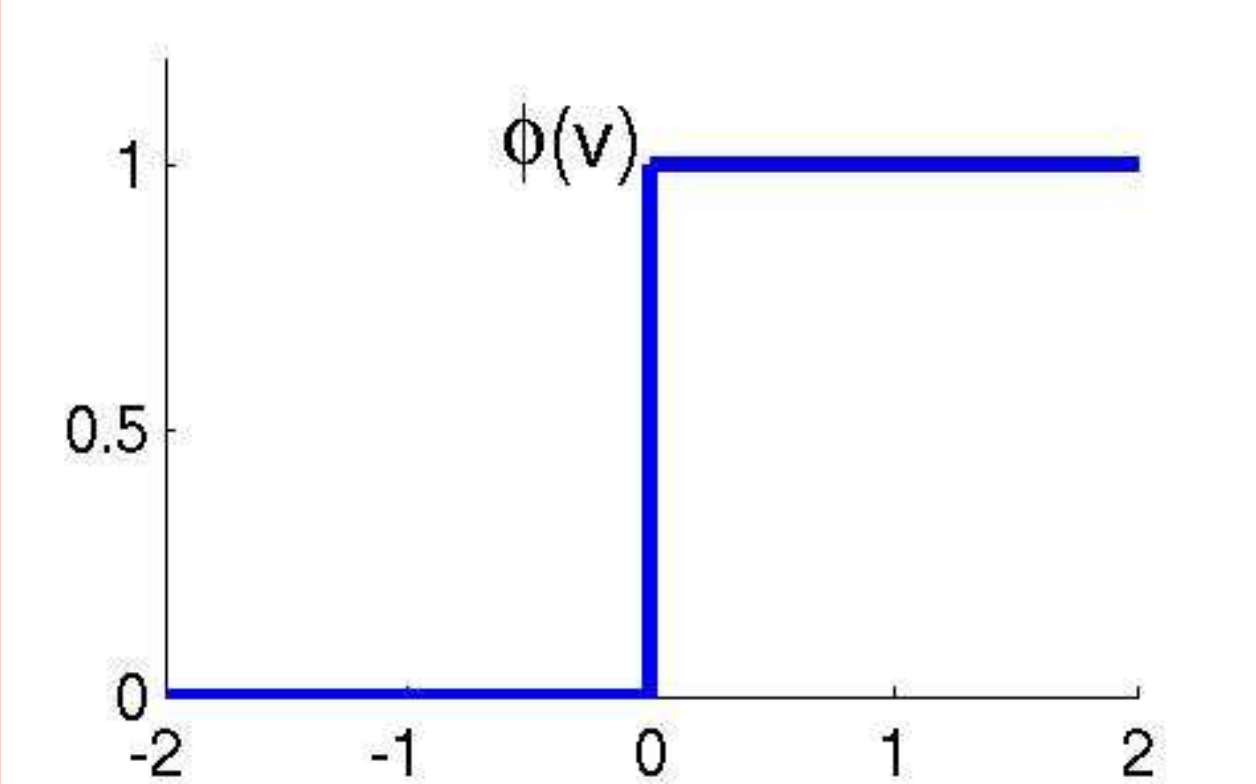

Figure 3: Hard limiter activation function.

- In general, symmetric activation functions are often preferable.
- Hard limiter is used in many early neural network models.
- Drawback: it is neither continuous or continuously differentiable.

Piecewise linear activation function

- Piecewise linear activation function is depicted in Figure 4.
- It is called also saturating linear function.
- A kind of combination of linear and hard limiter activation functions.
- Piecewise linear function is used in hardware realizations of neural networks, because it is fairly easy to implement.
- It is continuous but not continuously differentiable.

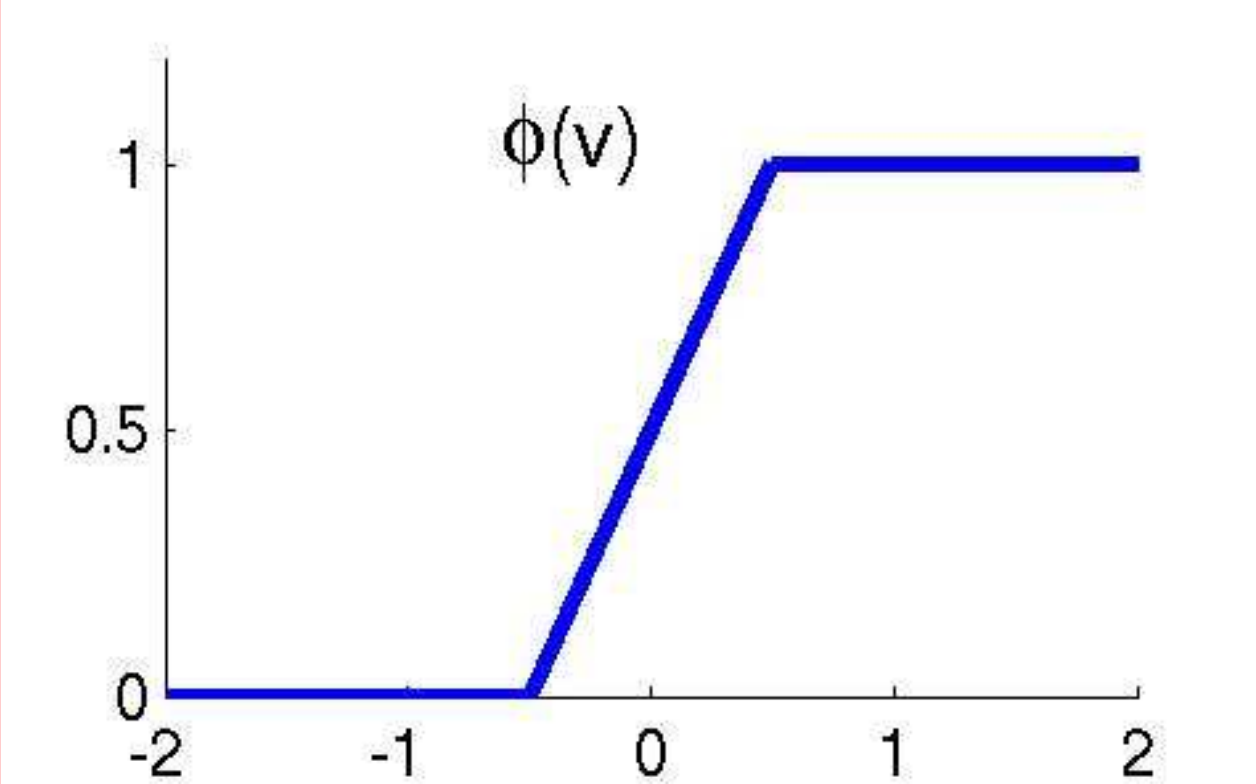

Figure 4: Piecewise linear activation function.

Sigmoidal activation functions

- Sigmoidal activation functions are used commonly in modern neural networks.
- The basic sigmoid activation function is shown in Figure 5.
- It is also called the logistic function.
- This logistic sigmoidal function is defined by

$$\phi(v) = \frac{1}{1 + e^{-av}} \tag{5}$$

- The slope parameter a is important.
- When $a \to \infty$, the logistic sigmoid approaches the threshold function.
- On the other hand, when $a \to 0$, the function has a large nearly linear region in the middle.

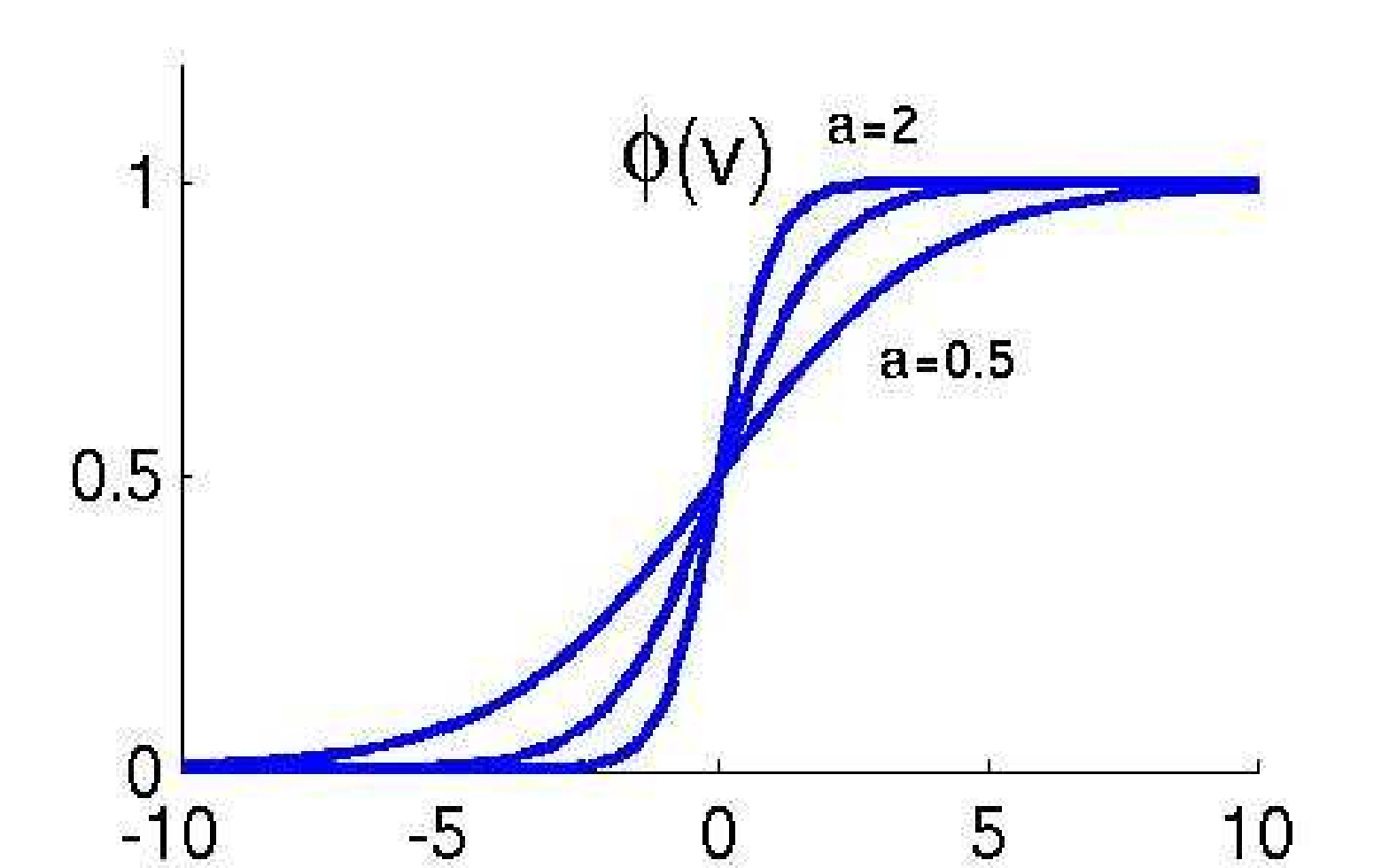

Figure 5: The sigmoid activation function.

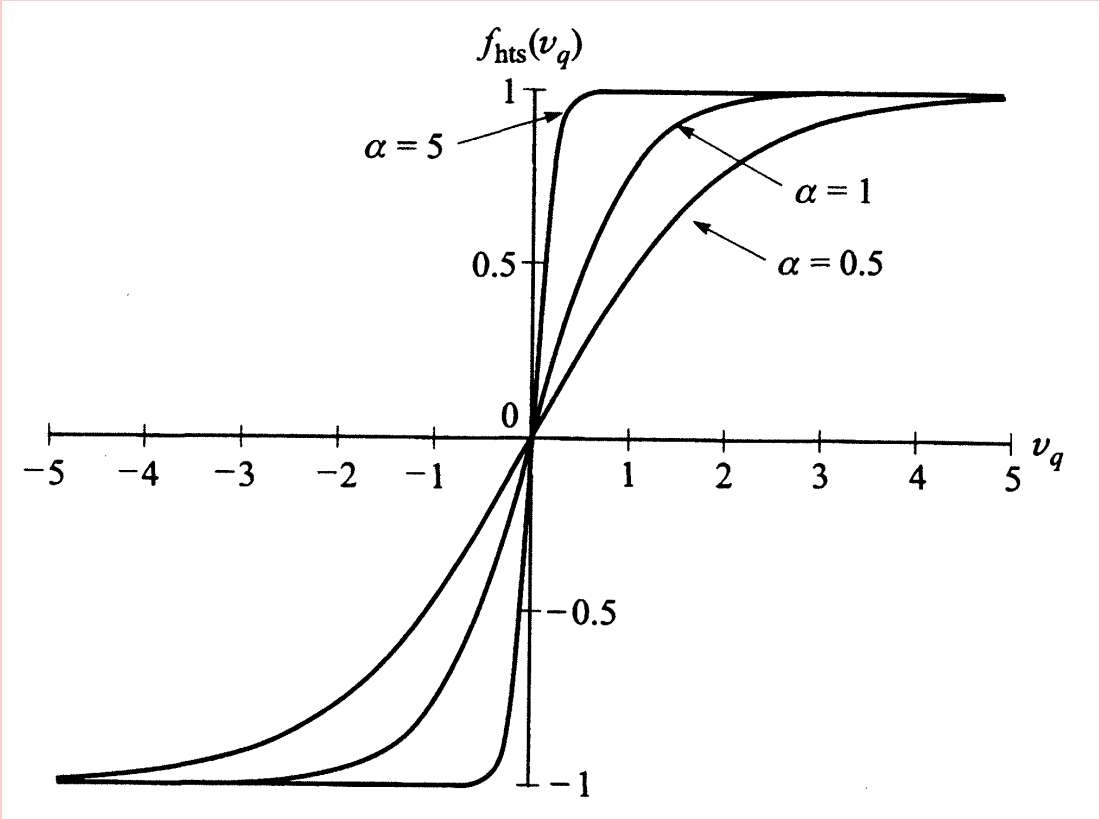

Figure 6: The $\tanh(\alpha v)$ activation function for 3 values of the slope parameter α.

- The sigmoidal activation function provides a graceful balance between linearity, nonlinearity, and saturation.
- It is both continuous and continuously differentiable.
- The symmetric version $\phi(v) = \tanh(av)$ is especially useful.
- This hyperbolic tangent function is depicted in Figure 6.
- The derivatives of these two sigmoidal functions can be expressed in terms of the functions itself.
- The sigmoidal functions keep the outputs of the neurons always between $(0, 1)$ or $(-1, +1)$.
- Therefore neural networks are always stable: finite inputs yield finite outputs.

Rectified linear activation function

- Quite recently, rectified linear activation function has become popular.
- Especially in deep networks having many layers.
- The reason is that neural networks learn considerably faster when it is used instead of sigmoidal nonlinearities.
- The rectified linear activation function is depicted in Figure 7 (left).
- Its values are $\phi(v) = 0$ for $v < 0$ and $\phi(v) = v$ for $v \geq 0$.
- Figure 7 (right) shows parametric rectified linear unit whose values are $\phi(v) = av$ for $v < 0$ where a is a small positive constant.
- It may improve further learning and performance of the neural network.

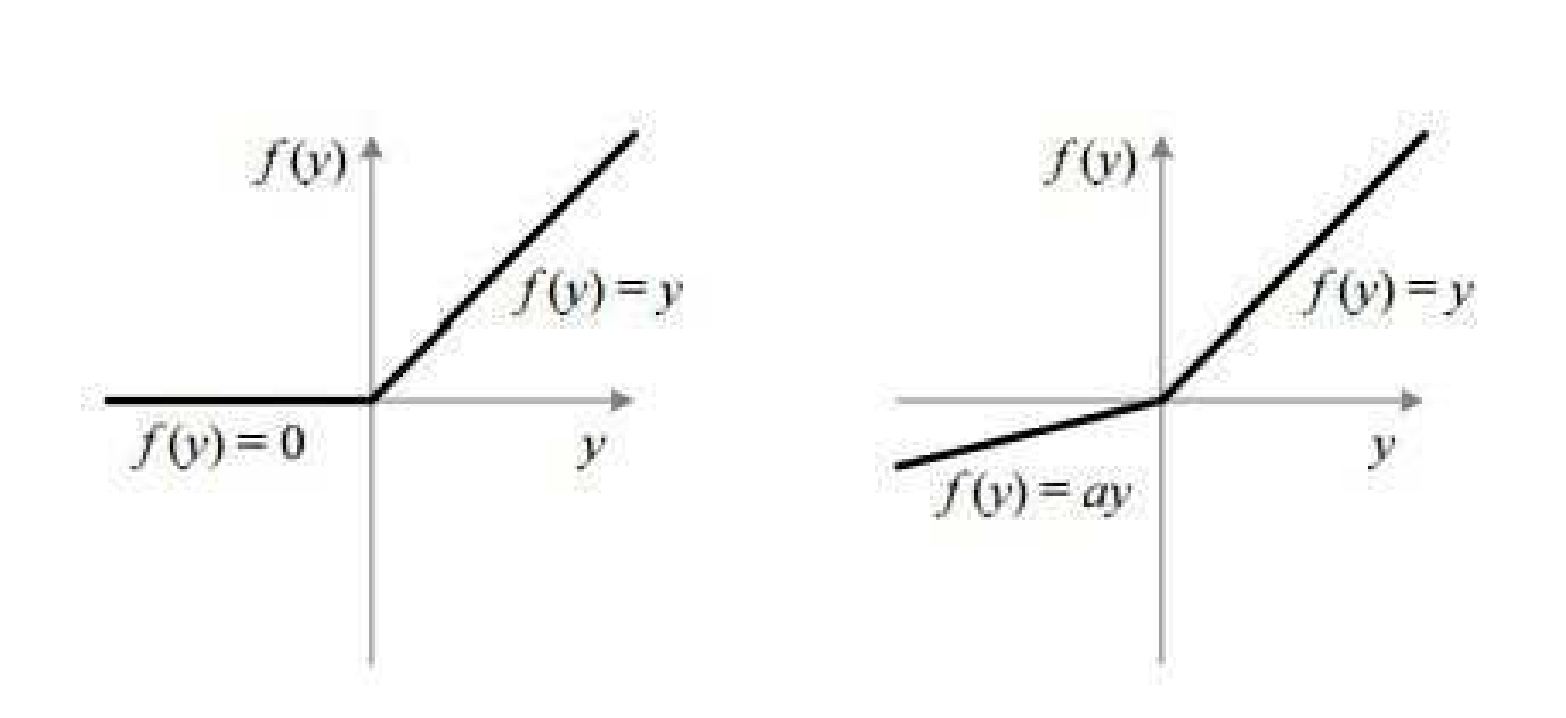

Figure 7: Rectified linear activation function (left) and its parametric modification (right).

Network architectures and approaches

- The structure of a neural network is closely related with the learning algorithm used to train the network.
- Learning algorithms for neural networks have been developed from quite different starting points, such as:
  - Multilayer perceptrons: Minimization of the mean-square error of the input-output mapping.
  - Radial-basis function networks: Curve-fitting (approximation) problem in a high-dimensional space.
  - Self-organizing maps: Competition of neurons.
  - Principal component networks: Hebbian learning
  - Simulated annealing: Thermal balance in statistical physics.
  - Support vector machines: Maximization of the margin of correct classification.

  - Independent component analysis: Statistical independence of the outputs.

- Most of these neural network models and their learning algorithms will be discussed later on in this course.
- Neural network architectures can be divided into three broad categories: 1. Single-layer feedforward networks; 2. Multilayer feedforward networks; and 3. Recurrent networks.

Single-layer feedforward networks

- The simplest form of neural networks.
- The input layer of source nodes distributes the input signals to the output layer of neurons.
- Each source node ("neuron") of the input layer receives as its input one component of the input (data) vector.

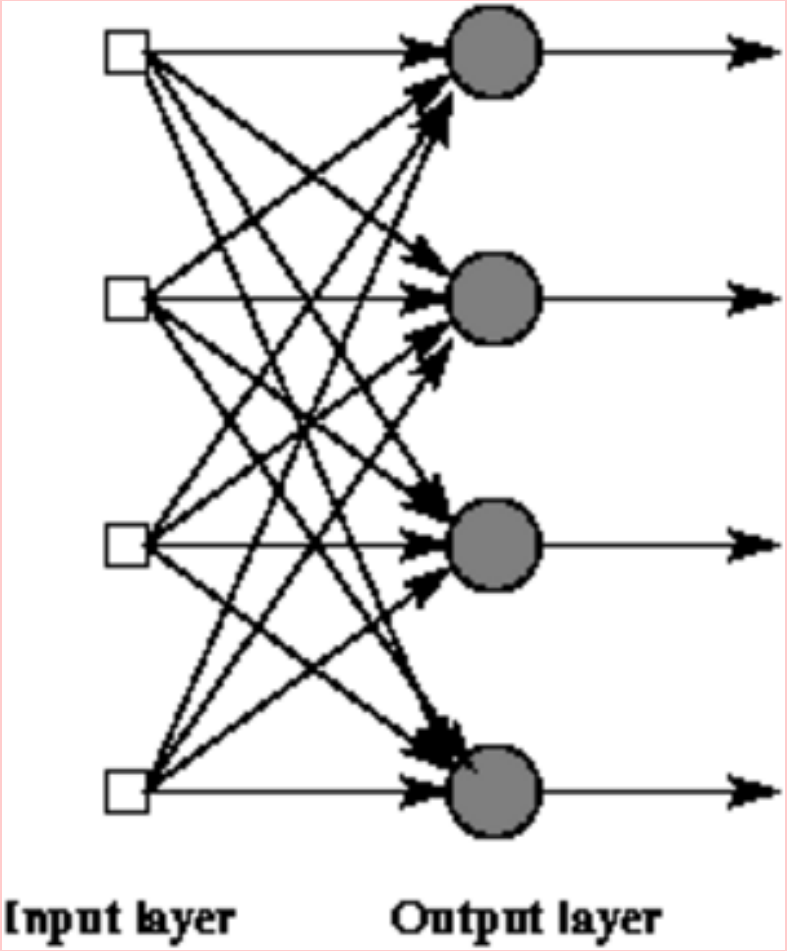

Figure 8: An example of a single-layer feedforward network.

- No computations are performed in the input layer.
- Therefore it is often not counted as a proper layer.
- All the computations take place in the output layer.
- The network is of feedforward type, because there is no feedback after the learning phase.
- Figure 8 shows a single-layer network with four nodes in both the input and output layers.

Multilayer feedforward networks

- In a multilayer network, there is one or more hidden layers.
- Their computation nodes are called hidden neurons or hidden units.

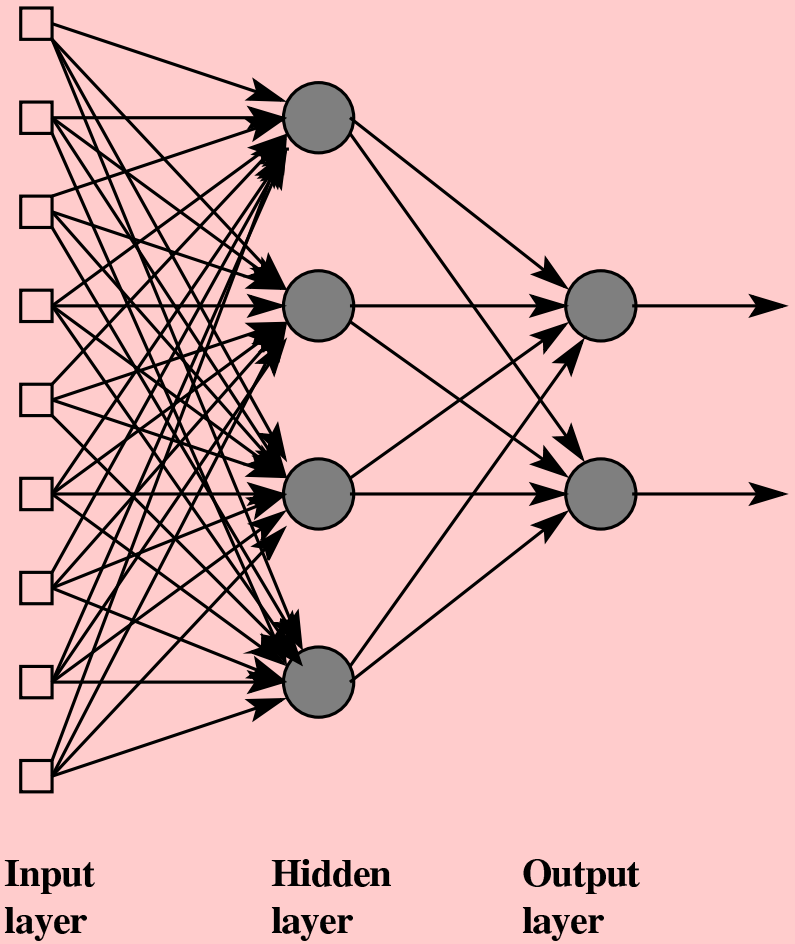

Figure 9: An example of a multilayer feedforward network.

- The hidden neurons can extract higher-order statistics and acquire more global information.
- Typically the input signals of each layer consist of the output signals of the preceding layer only.
- Figure 9 shows a feedforward network with 9 source (input) nodes, 4 hidden neurons, and 2 output neurons.
- The network is fully connected: all the nodes between subsequent layers are connected.
- Multilayer feedforward network is the most commonly used type of neural networks.
- After learning, all the signals and computations proceed layer-by-layer from the input layer to the output layer.
- However, feedback is often used in the learning phase.

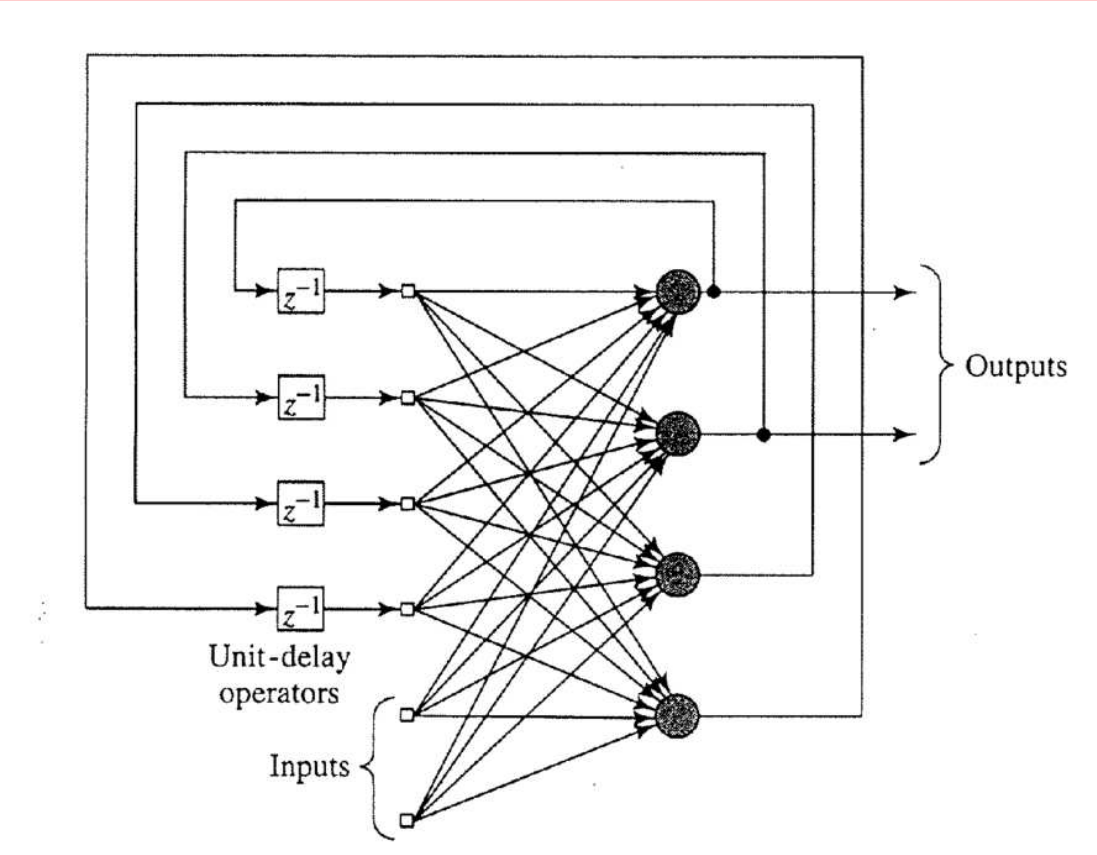

Figure 10: A recurrent network with hidden neurons.

Recurrent networks

- A recurrent neural network has at least one feedback loop.
- In a feedforward network there are no feedback loops after learning.
- Figure 10 shows a recurrent network which has:
  - Two inputs from outside data;
  - Four hidden neurons;
  - Four feedback inputs to hidden neurons;
  - Self-feedback connections;
  - Two outputs.
- The feedback with unit delay z^{-1} means that the previous value $y_k(n-1)$ of the output $y_k(n)$ of the neuron k is fed back to the network.
- There n denotes discrete time instant, in practice usually sample or iteration number.

- The feedback loops have a profound impact on the learning capability and performance of the network.
- The unit-delay elements result in a nonlinear dynamical behavior if the network contains nonlinear elements.
- In this course, we consider very little recurrent networks.
- Some neural network researchers use and study them.
- There exist some other neural network architectures that we do not discuss in this course.
- They are described briefly in Du's and Swamy's book on pages 8-10 and in Figure 1.5 there.

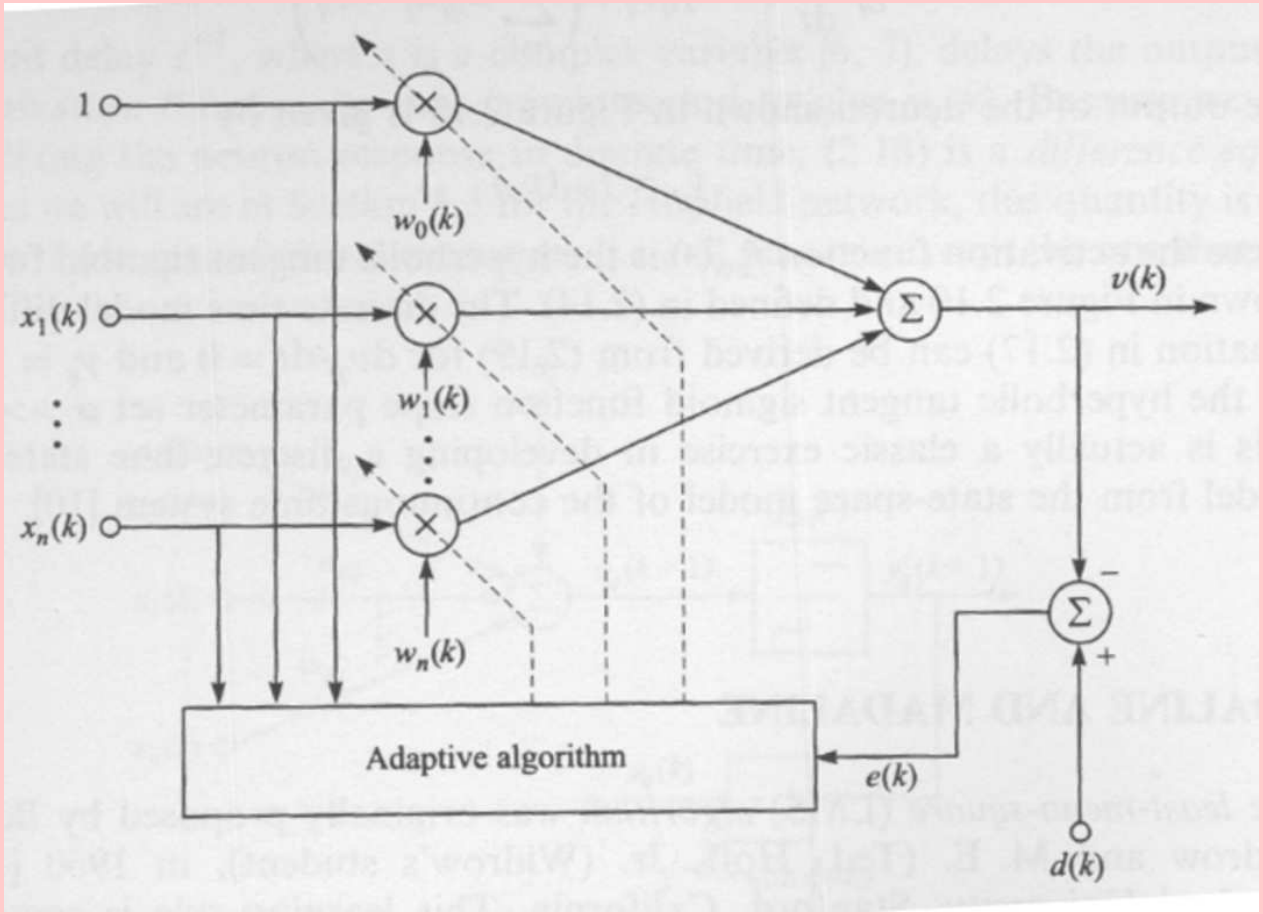

Figure 11: Simple adaptive linear combiner.

Learning rules for single neurons

Simple adaptive linear combiner

- Figure 11 shows the architecture of a simple adaptive linear combiner.
- It is a single linear neuron trained with an adaptive algorithm.
- In Figure 11, the uppermost input $x_0(k) = 1$ at the discrete time k is constant.
- It corresponds to the bias term $w_0(k) = \beta$.
- The k:th input vector to the neuron is denoted by $\mathbf{x}(k) = [x_0(k), x_1(k), \ldots, x_n(k)]^T$.
- The corresponding weight vector of the neuron in Figure 11 is denoted by $\mathbf{w}(k) = [w_0(k), w_1(k), \ldots, w_n(k)]^T$.

- The output of the neuron corresponding to the k:th input vector $\mathbf{x}(k)$ is the inner product

$$v(k) = \mathbf{x}^T(k)\mathbf{w}(k) = \mathbf{w}^T(k)\mathbf{x}(k) \tag{6}$$

- It is assumed that the simple adaptive linear combiner is trained in supervised manner.
- That is, we know K training pairs $\{\mathbf{x}(k), d(k)\}$, $k = 1, 2, \ldots, K$.
- There $d(k)$, called the desired response, is the true output corresponding to the input.
- First, the neuron is trained with the known training data.
- After that, unseen new data vectors can be presented to the simple adaptive linear combiner.
- And it computes the corresponding output (6).

The least-mean square (LMS) algorithm

- Consider now training of the simple adaptive linear combiner shown in Figure 11.
- To this end, a suitable adaptive learning algorithm is applied.
- The learning algorithm adapts the adjustable weights $w_i(k)$ of the linear neuron.
- It tries to make the error

$$e(k) = d(k) - v(k) = d(k) - \mathbf{w}^T(k)\mathbf{x}(k) \tag{7}$$

between the outputs $v(k)$ of the neuron and the corresponding desired responses $d(k)$ as small as possible.

The optimal Wiener-Hopf solution

- But the optimal weight vector $\mathbf{w}(k)$ would be somewhat different

for each training pair $\{\mathbf{x}(k), d(k)\}$.

- Therefore, the average error over all training pairs is minimized.
- The standard error criterion used here is the mean-square error

$$J(\mathbf{w}) = \frac{1}{2}\mathsf{E}[e(k)^2] = \frac{1}{2}\mathsf{E}\{[d(k) - \mathbf{w}^T(k)\mathbf{x}(k)]^2\} \tag{8}$$

- Here the operator E denotes mathematical expectation.
- The constant $1/2$ is introduced for mathematical convenience only.
- In (8), it is assumed that both $\mathbf{x}(k)$ and $d(k)$ are statistically wide-sense stationary random variables.
- Then the expectations can be replaced by the corresponding time averages (ergodicity condition).
- One can derive the optimal Wiener-Hopf solution which minimizes

the MSE error (8):

$$\mathbf{w}^* = \mathbf{C}_x^{-1}\mathbf{p} \tag{9}$$

- There $\mathbf{p} = \mathrm{E}\{d(k)\mathbf{x}(k)\}$ the cross-correlation vector between the desired responses $d(k)$ and input vectors $\mathbf{x}(k)$.
- And $\mathbf{C}_x = \mathrm{E}\{\mathbf{x}(k)\mathbf{x}^T(k)\}$ the covariance matrix of the input vectors $\mathbf{x}(k)$.
- These quantities can be estimated from the training data by replacing the expectatations with averages over samples:

$$\hat{\mathbf{p}} = \frac{1}{K}\sum_{i=1}^{K} d(k)\mathbf{x}(k), \qquad \hat{\mathbf{C}}_x = \frac{1}{K}\sum_{i=1}^{K} \mathbf{x}(k)\mathbf{x}^T(k) \tag{10}$$

- But if new input vectors $\mathbf{x}(k)$ come in, the Wiener-Hopf solution must be re-computed, and computing the inverse $\mathbf{C}_x^{-1}$ is costly.

Derivation of the LMS algorithm

- To circumvent these problems, Widrow and Hoff developed the least-mean square (LMS) algorithm.
- It is still used widely in adaptive signal processing.
- In neural networks, it is a starting point in developing the famous back-propagation algorithm for multilayer perceptron networks.
- A basic idea in the LMS algorithm is to seek the minimum of the MSE error by proceeding into the direction of steepest descent.
- This is given by the negative gradient of the MSE error (8)

$$-\frac{\partial J(\mathbf{w}(k))}{\partial \mathbf{w}(k)} = \mathrm{E}\{e(k)\mathbf{x}(k)\} \tag{11}$$

- To simplify the resulting algorithm, instantaneous stochastic gradient is used instead of the steepest descent direction.

- There the expectation in (11) is replaced by its instantaneous value, yielding

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} \approx -e(k)\mathbf{x}(k) \tag{12}$$

- Recall that the instantaneuos error (7) is

$$e(k) = d(k) - \mathbf{w}^T(k)\mathbf{x}(k) \tag{13}$$

- Proceeding to the direction of the negative instantaneous gradient by the amount determined by the step size μ,

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \mu\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} \tag{14}$$

  yields for the weight vector the update formula

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu e(k)\mathbf{x}(k) \tag{15}$$

- Eqs. (13) and (15) together constitute the famous LMS algorithm.

Properties of the LMS algorithm

- Note that the LMS algorithm is very simple: the most complicated operation is computing an inner product!
- The use of instantaneous stochastic gradient causes that the LMS algorithm fluctuates at each iteration.
- This is because the estimates of the cross-covariance vector $\mathbf{p}$ and covariance matrix $\mathbf{C}_x$ based on a single data vector are very crude.
- But with many iterations, the LMS algorithm proceeds approximately into the direction of steepest descent.
- The learning parameter μ is called also gain parameter or learning rate or step size.
- It determines how large updates are made at each iteration.
- If μ is chosen too small, the LMS algorithm converges slowly.

- On the other hand, if μ is chosen too large, the algorithm diverges (becomes unstable or "explodes").
- It can be shown theoretically that the LMS algorithm remains stable and converges in the mean if the learning parameter

$$0 < \mu < \frac{2}{\lambda_{max}} \tag{16}$$

- Here λ_{max} is the largest eigenvalue of the covariance matrix $\mathbf{C}_x$ on the input vectors $\mathbf{x}$.
- If λ_{max} is unknown one can use the normalized version of the LMS algorithm.
- There the learning parameter $\mu(k)$ on iteration k is

$$\mu(k) = \frac{\mu_0}{\| \mathbf{x}(k) \|^2} \tag{17}$$

where μ_0 is a fixed constant.

- Now stability is guaranteed if $0 < \mu_0 < 2$.
- However, the practical range is $0.1 \leq \mu_0 \leq 1$.
- Using the finite learning parameter (17) or (16), the LMS algorithm converges in the mean only.
- That is, the average value of the weight vector $\mathbf{w}(k)$ is correct.
- But the weight vector fluctuates around its true value the more the larger the learning parameter $\mu(k)$ is.
- To achieve convergence to the true value $\mathbf{w}$, the learning parameter $\mu(k)$ must be driven to zero when $k \to \infty$.
- In the theory of stochastic approximation, this is achieved by using the learning parameter

$$\mu(k) = \frac{\mu_0}{k} \tag{18}$$

where μ_0 is a fixed constant.

- But in practice the learning parameter (18) approaches zero too quickly with the iteration number k.
- Causing that the LMS algorithm stops too early, and never converges to the true value $\mathbf{w}$ in reality.
- Therefore, it is better to use the search-then-converge version where

$$\mu(k) = \frac{\mu_0}{1 + k/\tau} \tag{19}$$

- There the search time constant τ is typically $100 \leq \tau \leq 500$.
- The learning parameter (19) decreases slowly in the beginning, allowing the LMS algorithm to find the region near the true minimum of the mean-square error.
- After this, the learning parameter (19) decreases radually towards zero, and the LMS algorithm converges to the true minimum.

- Figure 12 illustrates three learning rate scedules used in the LMS algorithm: classical (constant), (18), and (19).
- Note that the scales are logarithmic in Figure 12.

Adaline

- Adaline (adaptive linear element) is an old adaptive pattern classification method trained by the LMS algorithm.
- The schematic diagram of Adaline is shown in Figure 13.
- Adaline uses a single linear neuron trained using the LMS algorithm based on the linear error $e(k)$ in Figure 13.
- The symmetric hard-limiting quantizer in Figure 13 is used only after training of the Adaline for classification purposes.

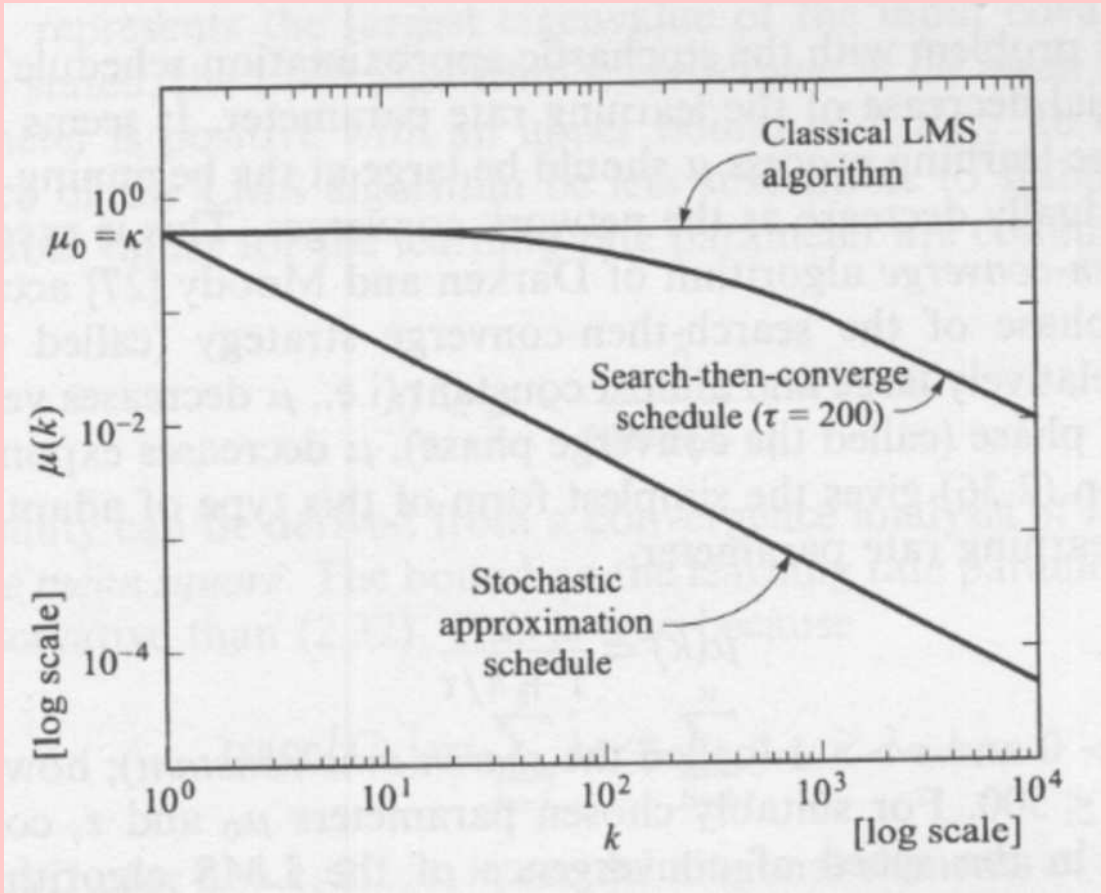

Figure 12: A comparison of different learning rate schedules of the LMS algorithm.

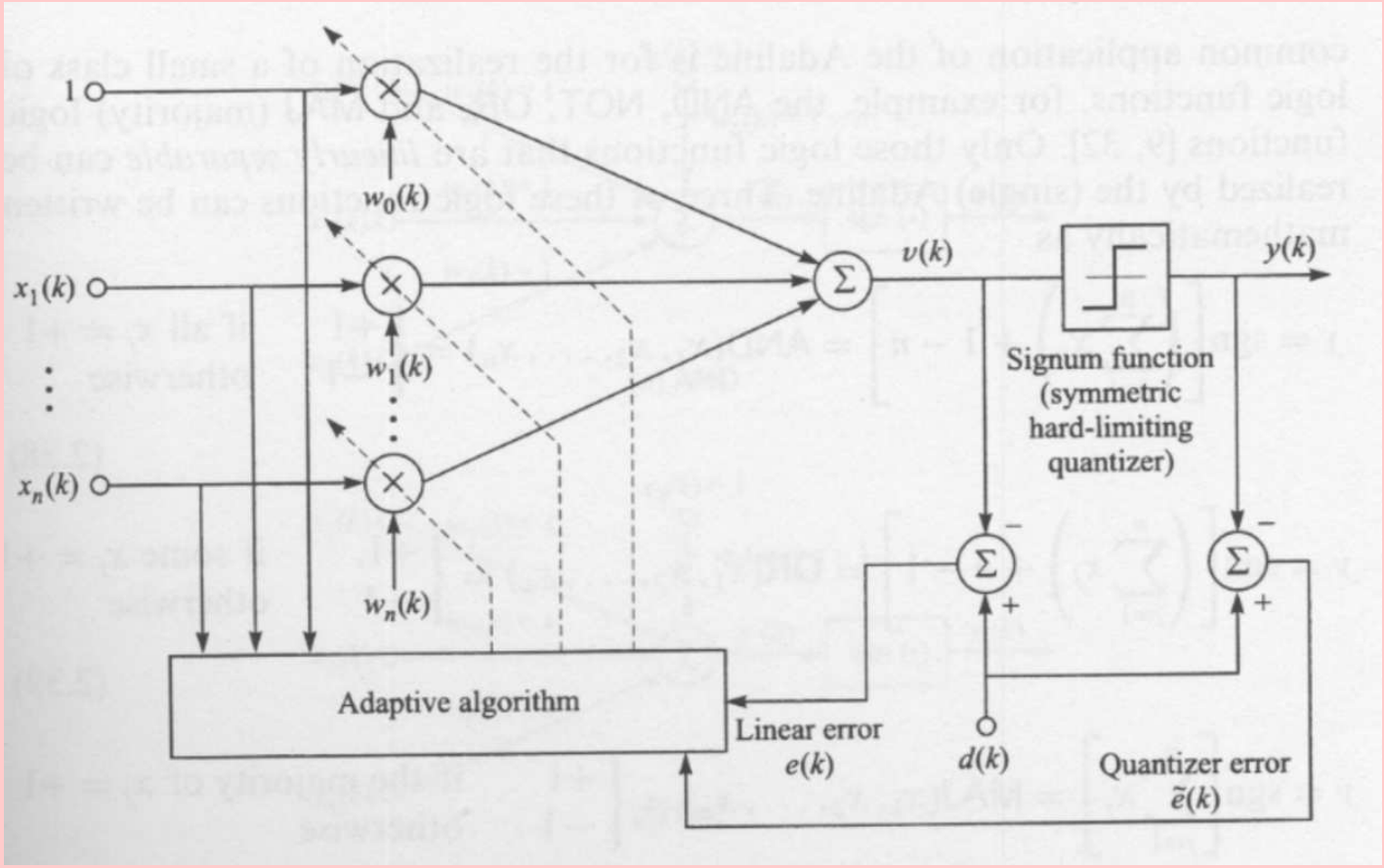

Figure 13: A schematic diagram of Adaline.

- Ideally, it should provide the output $+1$ for all the data (pattern) vectors belonging to the class 1.
- And the output -1 for the pattern vectors belonging to the class 2.
- Adaline is able to classify data vectors belonging to two classes only.
- Furthermore, it can classify the data vectors correctly only if the classes are linearly separable.
- That is, the regions of the classes must be separable by a hyperplane (straight line in two-dimensional case).
- See Figure 14 for an illustration of this property.
- Linear separability is a serious limitation in practice.
- In most classification problems, the pattern classes are at least partly overlapping and not linearly separable.
- Therefore, more powerful classifiers than Adaline are needed.

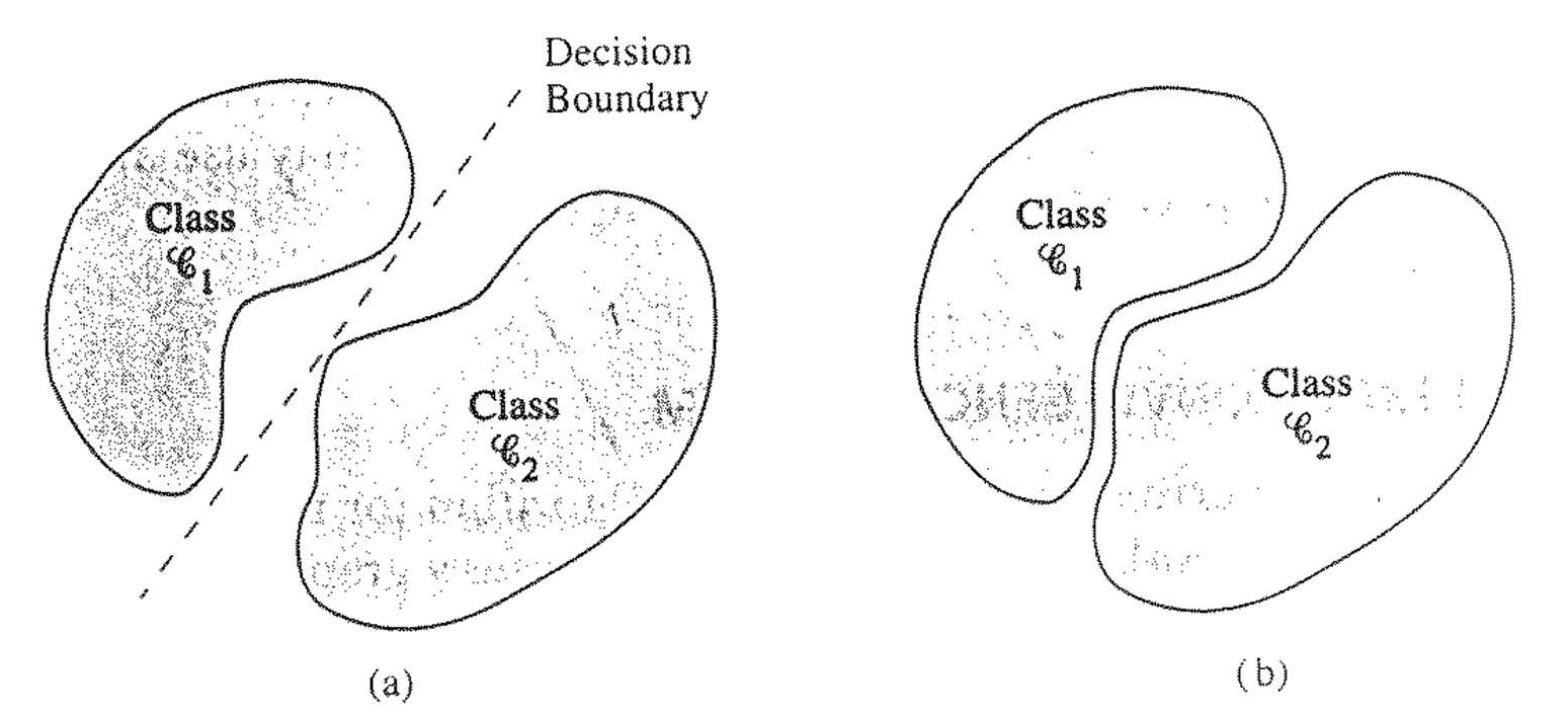

Figure 14: An example of linearly separable classes (a) and nonlinearly separable classes (b)

Simple perceptron

- The simple (single-layer) perceptron has probably had the most significant impact on the development of modern neural networks.
- The original concept was developed by F. Rosenblatt in the late 1950s.
- Minsky and Papert clearly demonstrated the limitations of the simple perceptron in their book in 1968.
- This decreased a lot the research activity on neural networks in the 1970s.
- Until a real boom developed in the 1980s, caused by the popularity of Hopfield networks and multilayer perceptron networks.
- The simple perceptron is very similar to the Adaline; see Figure 13.
- However, the perceptron does not use the linear error $e(k)$ in

Figure 13.

- Instead, the perceptron is taught using the nonlinear quantizer error

$$\tilde{e}(k) = d(k) - y(k) = d(k) - \text{sgn}[v(k)] \tag{20}$$

- Here $v(k) = \mathbf{w}^T(k)\mathbf{x}(k)$ is the linear response of the neuron.
- And $\text{sgn}(v)$ is the signum function: -1 for $v < 0$, and $+1$ for $v \geq 0$.
- The learning rule of the simple perceptron

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \alpha\tilde{e}(k)\mathbf{x}(k) \tag{21}$$

  is stable irrespective of the choice of the learning parameter α.
- This is because the signum function keeps the error $\tilde{e}(k)$ in (20) always finite.
- But the perceptron has the same deficiency as the Adaline: it can classify correctly only linearly separable pattern vectors after training.