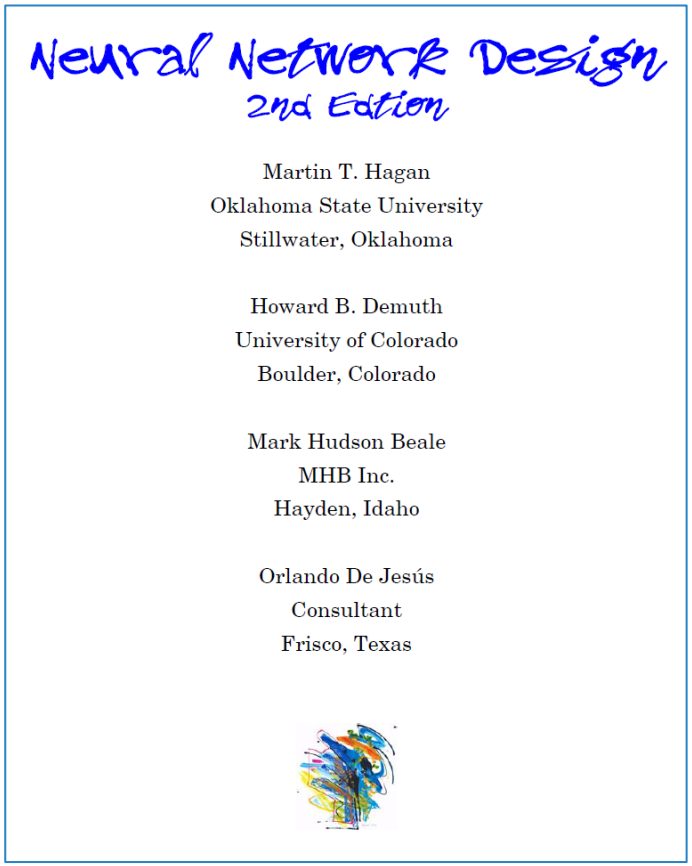
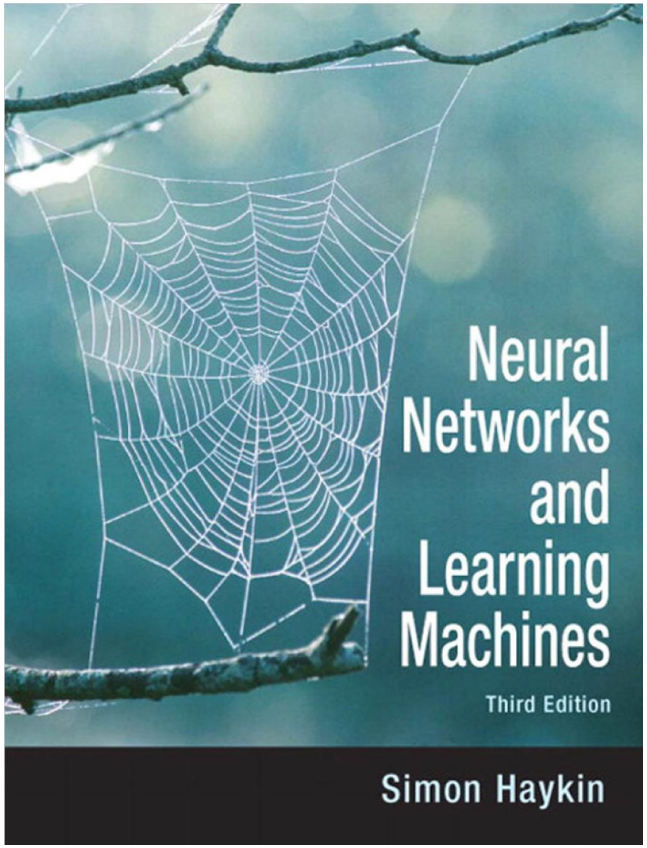
Neural Network and Deep Learning

Lecture2:

Basics of Artificial Neural Networks

REFERENCES


Neural Network Design
2nd Edition
Martin T. Hagan
Oklahoma State University
Stillwater, Oklahoma
Howard B. Demuth
University of Colorado
Boulder, Colorado
Mark Hudson Beale
MHB Inc.
Hayden, Idaho
Orlando De Jesús
Consultant
Frisco, Texas


Neural Networks and Learning Machines
Third Edition
Simon Haykin

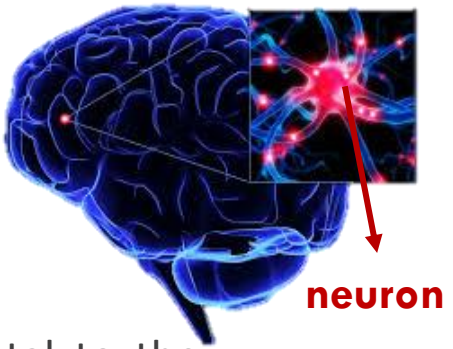
OUTLINE

- Biological Neural Networks
- Artificial Neural Networks
- Biological VS. Artificial Neural Networks
- Perceptron
- Learning rules in Neural Network
- Recurrent Network
- Adaline
- LMS Algorithm
- Backpropagation Algorithm
- Function Approximation
- Universal Approximation Theorem
- Activation Functions

BIOLOGICAL NEURAL NETWORKS

- **Human brain**: The brain is a highly **complex**, **nonlinear**, and **parallel** computer (information-processing system). It has the capability to organize its structural constituents, known as **neurons**, so as to perform certain computations (e.g., pattern recognition, perception, and motor control) many times faster than the fastest digital computer in existence today.

- **Biological systems**: consist of very simple but numerous nerve cells simple but many processing units that work massively in parallel and (which is probably one of the most significant aspects) have the capability to learn. It can learn from training samples.

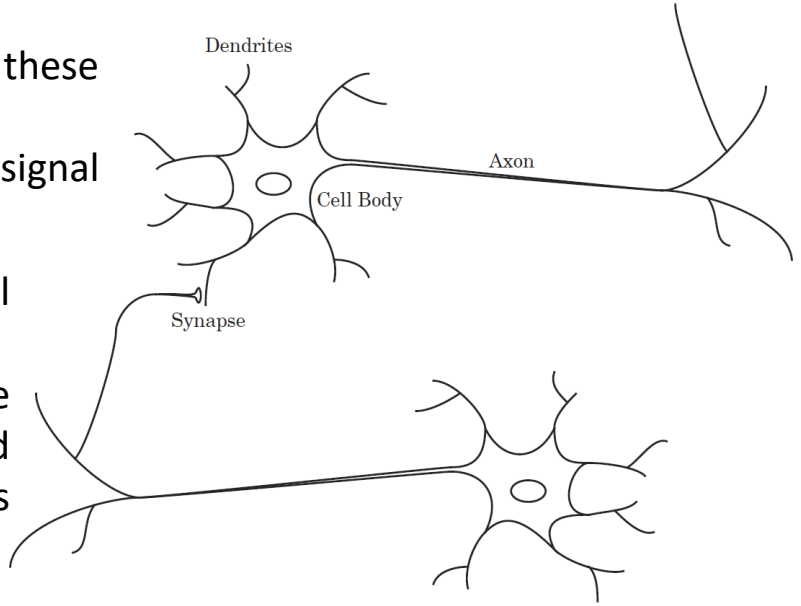


- A **neuron** is an information-processing unit that is fundamental to the operation of a neural network, which forms the basis for designing a large family of neural networks.

BIOLOGICAL NEURAL NETWORK

- The brain consists of a large number (approximately 10^{11}) of highly connected elements (approximately 10^4 connections per element) called **neurons**.

- For our purposes these neurons have three principal components: the **dendrites**, the **cell body** and the **axon**.
  1. The **dendrites** are tree-like receptive networks of nerve fibers that carry electrical signals into the cell body.
  2. The **cell body** effectively sums and thresholds these incoming signals.
  3. The **axon** is a single long fiber that carries the signal from the cell body out to other neurons.

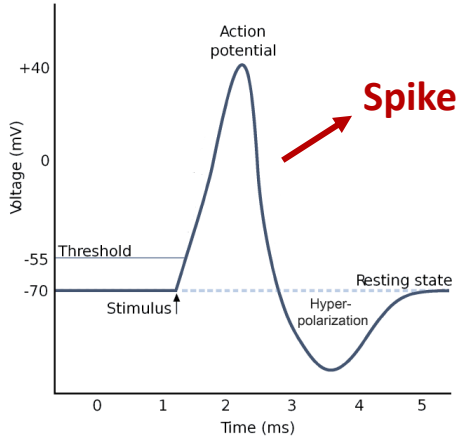


- The point of contact between an axon of one cell and a dendrite of another cell is called a **synapse**.
- It is the arrangement of neurons and the strengths of the individual synapses, determined by a complex chemical process, that establishes the function of the neural network.

BIOLOGICAL NEURAL NETWORKS

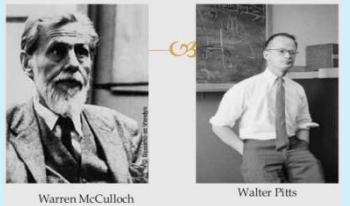
- After the **cell nucleus (soma)** has received a plenty of activating (=stimulating) and inhibiting (=diminishing) signals by **synapses** or **dendrites**, the soma accumulates these signals. As soon as the accumulated signal exceeds a certain value (called threshold value), the cell nucleus of the neuron activates an electrical pulse which then is transmitted to the neurons connected to the current one.

Only spikes are important since other neurons receive them (signals).



ARTIFICIAL NEURAL NETWORKS

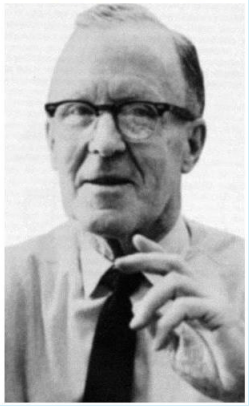
- In the formative years of neural networks (1943–1958), several researchers stand out for their pioneering contributions:
  1. McCulloch and Pitts (1943) for introducing the idea of neural networks as computing machines.
  2. Hebb (1949) for postulating the first rule for self-organized learning.
  3. Rosenblatt (1958) for proposing the perceptron (including neurons with hard-limiting activation function) as the first model for learning with a teacher (i.e., supervised learning).

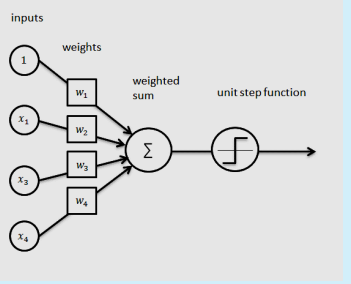


1943: Walter Pitts and Warren McCulloch
computer model based on the neural networks of the human brain

Donald O Hebb

- Wrote The Organization of Behavior in 1949
- "When an axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased" (Hebb 1949)

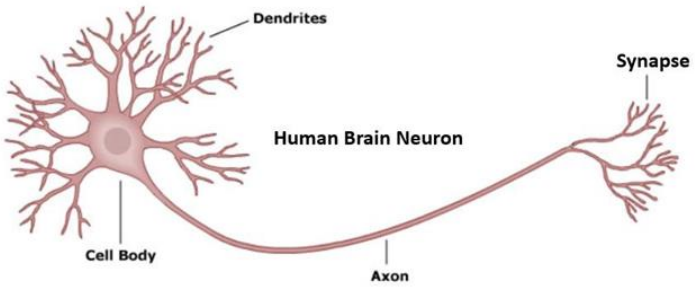


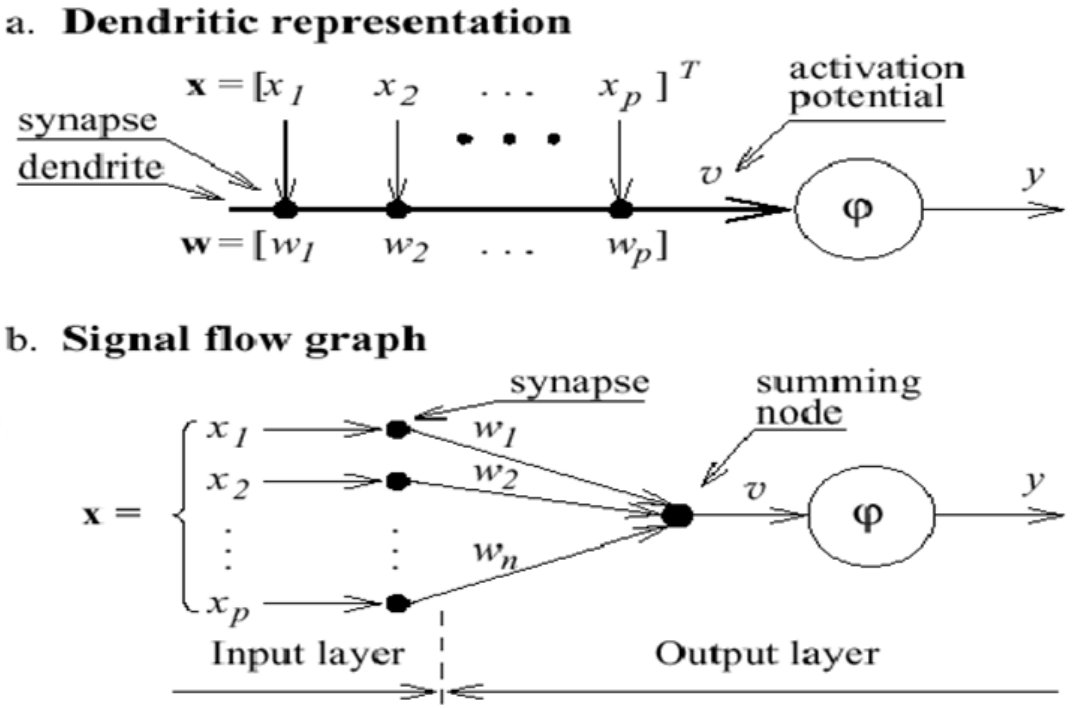


1958: Frank Rosenblatt
perceptron

BIOLOGICAL VS. ARTIFICIAL NEURAL NETWORKS

- Very simple neuron model receives a vectorial input: $\vec{x}$
- With components x_i. These are multiplied by the appropriate weights w_i and accumulated: $\sum_i x_i w_i$.
- the nonlinear mapping φ defines the scalar output y: $y = \varphi(\sum_i x_i w_i)$

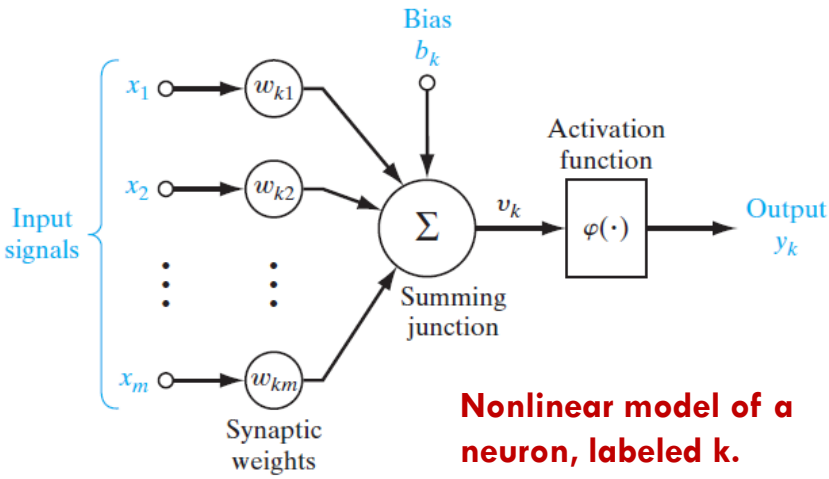




BIOLOGICAL VS. ARTIFICIAL NEURAL NETWORKS

➢ **Three basic elements of the neural model:**

1. A set of synapses, or connecting links, each of which is characterized by a weight or strength of its own. Specifically, a signal x_j at the input of synapse j connected to neuron k is multiplied by the synaptic weight w_{kj}.
2. An adder for summing the input signals, weighted by the respective synaptic strengths of the neuron; the operations described here constitute a linear combiner.
3. An activation function for limiting the amplitude of the output of a neuron. The activation function is also referred to as a squashing function, in that it squashes (limits) the permissible amplitude range of the output signal to some finite value, interval [0,1], or, alternatively, [-1,1].



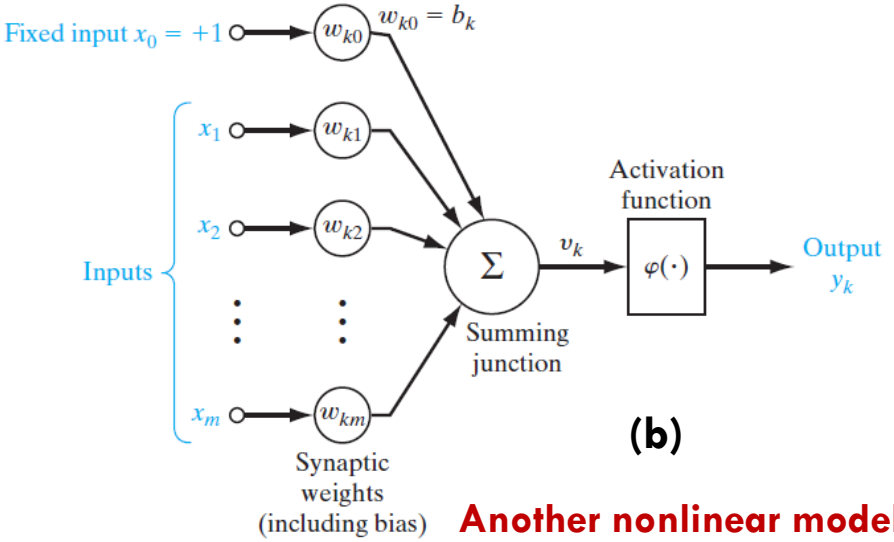
Nonlinear model of a neuron, labeled k.

BIOLOGICAL VS. ARTIFICIAL NEURAL NETWORKS

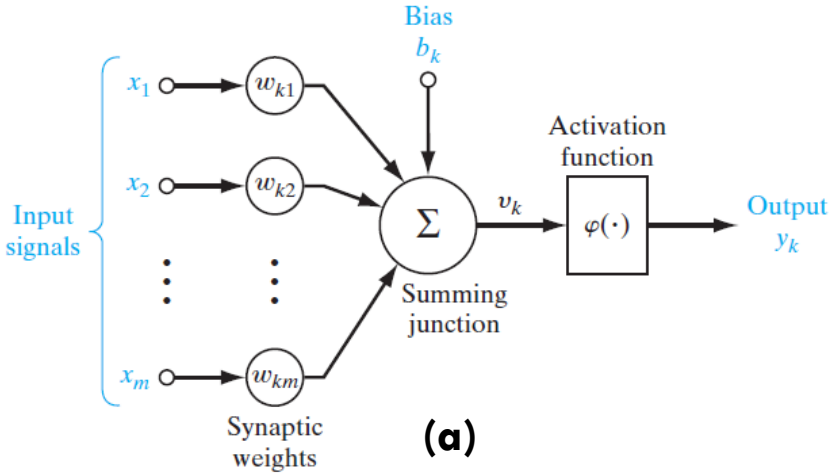
- **Neural model:**

(a): $u_k = \sum_{j=1}^{m} w_{kj} x_j \, , v_k = u_k + b_k, y_k = \varphi(v_k)$

(b): $v_k = \sum_{j=0}^{m} w_{kj} x_j$



(b)



(a)

Another nonlinear model of a neuron; w_{k0} accounts for the bias b_k.

ARTIFICIAL NEURAL NETWORKS

Neuron Model: Single-Input Neuron

The scalar **input** p is multiplied by the scalar ***weight*** w to form wp, one of the terms that is sent to the **summer**. The other **input, 1** , is multiplied by a **bias** b and then passed to the summer. The summer **output** n, often referred to as the net input, goes into a **transfer function** f, which produces the scalar neuron output .

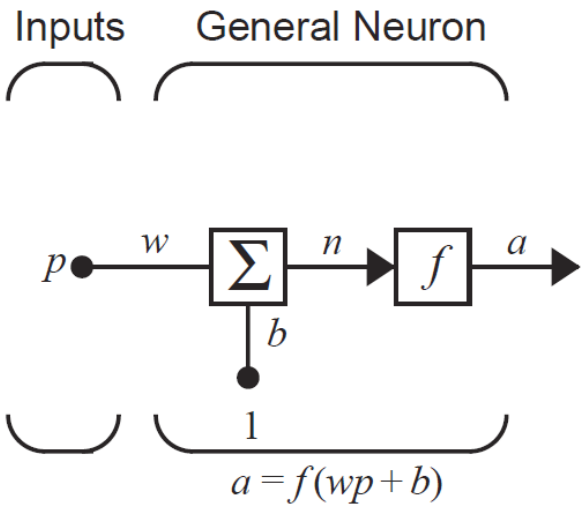
- (Some authors use the term "**activation function**" rather than ***transfer function*** and "**offset**" rather than **bias**.)
- **Artificial vs. biological neuron**
  - the **weight** corresponds to the **strength of a synapse**
  - the **cell body** is represented by the **summation and the transfer function**
  - the **neuron output** represents the **signal on the axon**

The neuron output is calculated as

$$a = f(wp + b).$$

If, for instance, $w = 3$, $p = 2$ and $b = -1.5$, then

$$a = f(3(2) - 1.5) = f(4.5)$$



ARTIFICIAL NEURAL NETWORKS

Multiple-Input Neuron

- The neuron has a bias b, which is summed with the weighted inputs to form the net input n:

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \cdots + w_{1,R}p_R + b$$

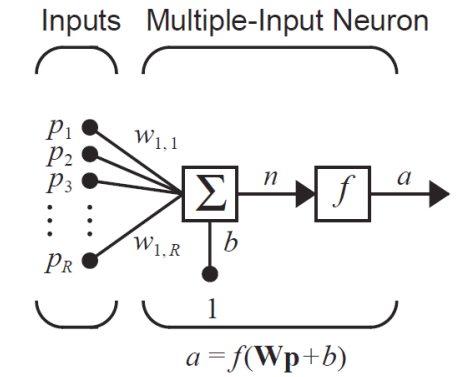
- This expression can be written in matrix form:

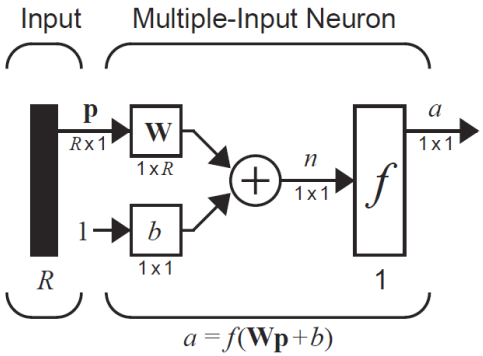
$$n = \mathbf{W}\mathbf{p} + b$$

where the matrix for the single neuron case has only one row.

- Now the neuron output can be written as

$$a = f(\mathbf{W}\mathbf{p} + b)$$





ARTIFICIAL NEURAL NETWORKS

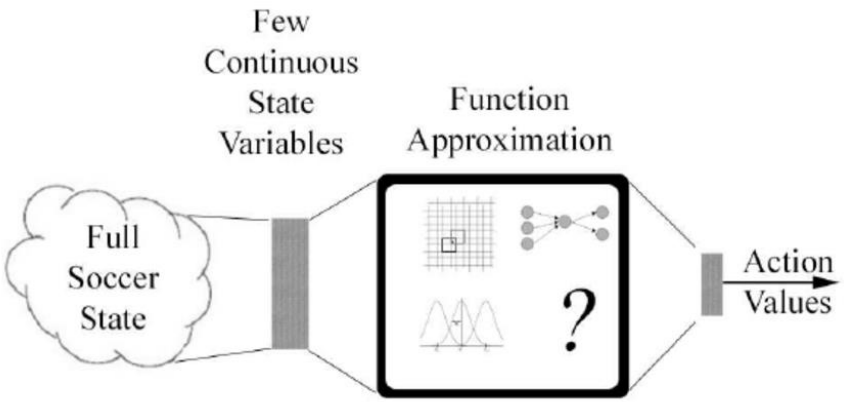
Applications of artificial neural network

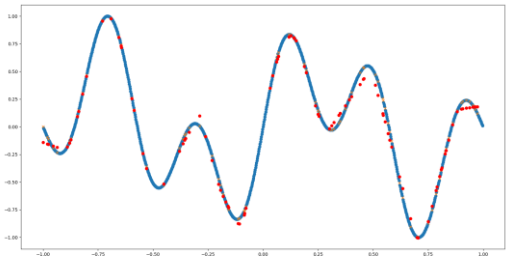
Neural networks can be used for a wide variety of applications, and it would be impossible to provide case studies for each application. We will limit our presentations to five important application areas:

- Function approximation (aka, nonlinear regression),
- Pattern recognition (aka, pattern classification),
- Prediction (aka, time series analysis, …),
- Clustering, and
- …

ARTIFICIAL NEURAL NETWORKS

- **Applications of artificial neural network**
  - For **function approximation** problems, the training set consists of a set of dependent variables (response variables) and one or more independent variables (explanatory variables). The neural network learns to create a mapping between the explanatory variables and the response variables.





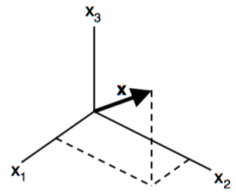
ARTIFICIAL NEURAL NETWORKS

Applications of artificial neural network

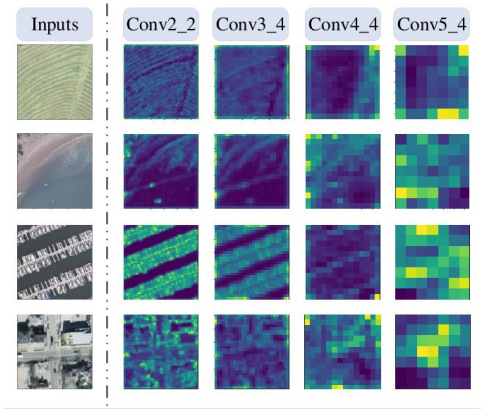
- **Pattern recognition**
  - A **pattern** is a composite of traits or features characteristic of an individual. A **pattern** is a collection of signals originating from similar objects.
  - **Feature** is any distinctive aspect, quality or characteristic of objects. Features may be symbolic (i. e., color) or numeric (i. e., height).
  - **Feature vector** is the combination of d features, represented as a d-dimensional column vector.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_d \end{bmatrix}$$

Feature vector



Feature space (3D)

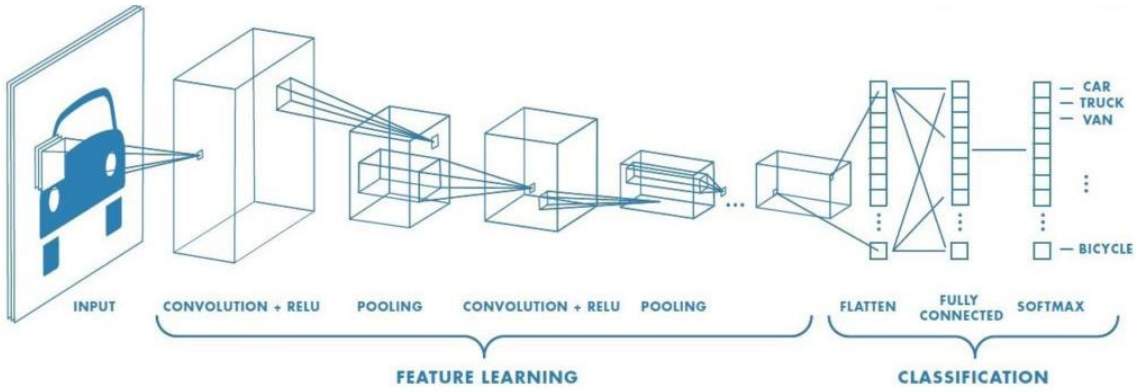


Feature map

ARTIFICIAL NEURAL NETWORKS

Applications of artificial neural network

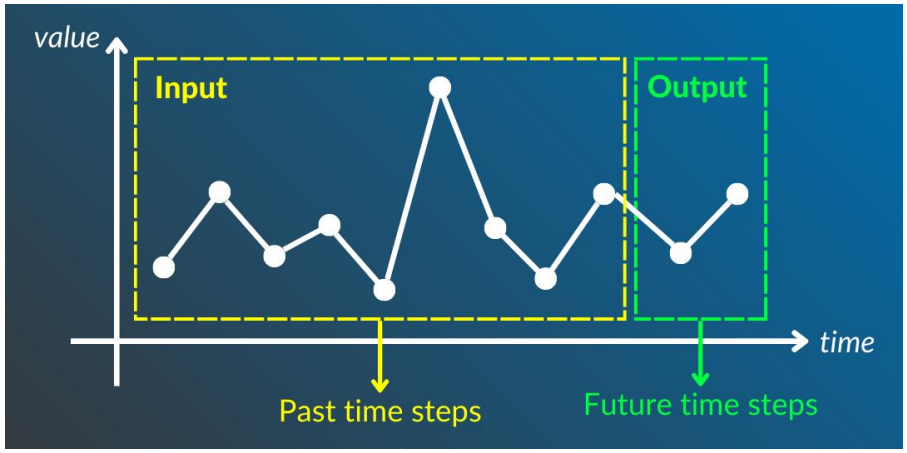
- **Pattern recognition** is the process of identifying features as originating from particular class of objects.
- In **pattern recognition** problems, you want a neural network to classify inputs into a set of target categories.
- **Pattern recognition** is the task of assigning a class to an observation based on patterns extracted from data.



ARTIFICIAL NEURAL NETWORKS

Applications of artificial neural network

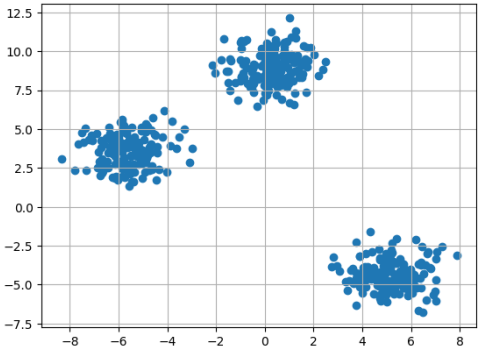
- **Prediction** is a kind of dynamic filtering, in which past values of one or more time series are used to predict future values.
- There are many applications for **prediction**. For example, a financial analyst might want to predict the future value of a stock, bond, or other financial instrument. An engineer might want to predict the impending failure of a jet engine.



ARTIFICIAL NEURAL NETWORKS

Applications of artificial neural network

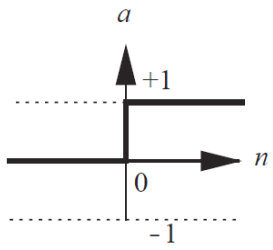
- The main job of artificial neural network **clustering** algorithms is to group data points.
- Data points in the same group are more alike than those in other groups.
- This is helpful for sorting data without labels, looking at data, and finding patterns.



ARTIFICIAL NEURAL NETWORKS

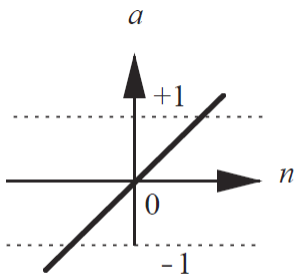
Activation Functions (transfer function)

- The activation function may be a linear or a nonlinear function of n. A particular transfer function is chosen to satisfy some specification of the problem that the neuron is attempting to solve.
- Three of the most commonly used functions are discussed below:
  - hard limit transfer function
  - linear transfer function
  - log-sigmoid transfer function



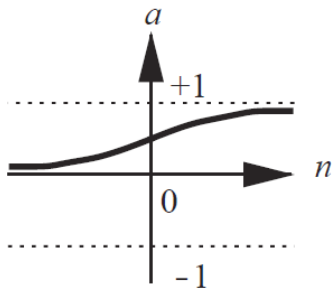
$a = hardlim(n)$

$$a = 0 \quad n < 0$$
$$a = 1 \quad n \geq 0$$



$a = purelin(n)$

$$a = n$$



$a = logsig(n)$

$$a = \frac{1}{1 + e^{-n}}$$

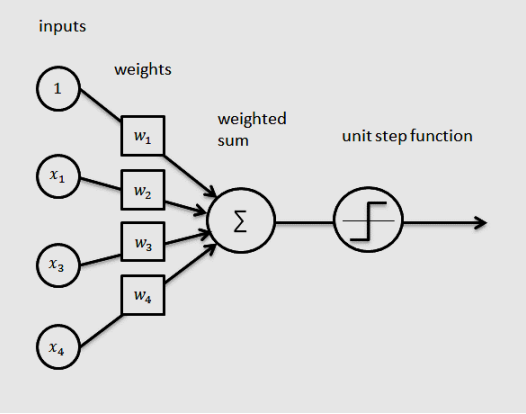
ARTIFICIAL NEURAL NETWORKS

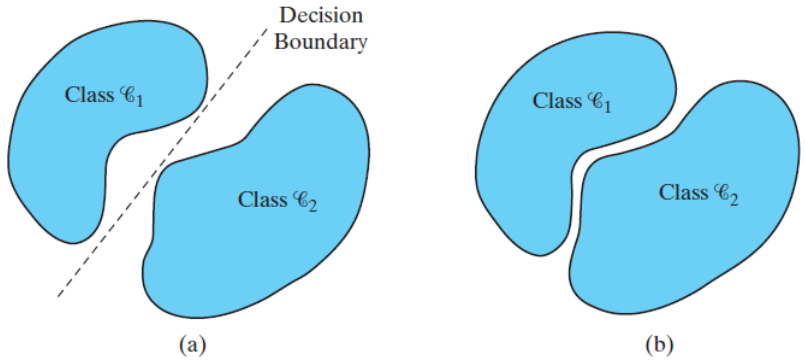
- **Activation Functions (transfer function)**
  - The **hard limit transfer function**: We will use this function to create neurons that classify inputs into two distinct categories.
  - **linear transfer function**: Neurons with this transfer function are used in the ADALINE networks
  - The **log-sigmoid transfer function** is commonly used in multilayer networks that are trained using the backpropagation algorithm. This transfer function takes the input (which may have any value between plus and minus infinity) and squashes the output into the range 0 to 1

PERCEPTRON

Perceptron

- Rosenblatt (1958) for proposing the perceptron as the first model for learning with a teacher (i.e., supervised learning).
- The perceptron is the simplest form of a neural network used for the classification of patterns said to be linearly separable (i.e., patterns that lie on opposite sides of a hyperplane).
- Rosenblatt's perceptron is built around a nonlinear neuron. Such a neural modeling consists of a linear combiner followed by a hard limiter (performing the signum function),





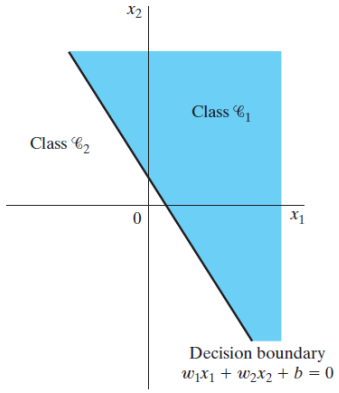
(a) A pair of linearly separable patterns. (b) A pair of non-linearly separable patterns.

PERCEPTRON

Perceptron

- Rosenblatt proved that if the patterns (vectors) used to train the perceptron are drawn from two linearly separable classes, then the perceptron algorithm converges and positions the decision surface in the form of a hyperplane between the two classes.
- The perceptron built around a single neuron is limited to performing pattern classification with only two classes.
- By expanding the output layer of the perceptron to include more than one neuron, we may correspondingly perform classification with more than two classes. However, the classes have to be linearly separable for the perceptron to work properly.

Illustration of the hyperplane (in this example, a straight line) as decision boundary for a two-dimensional, two-class pattern-classification problem.

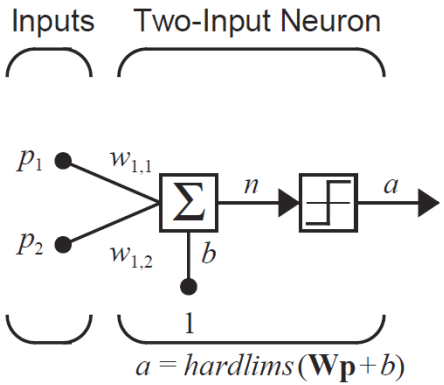


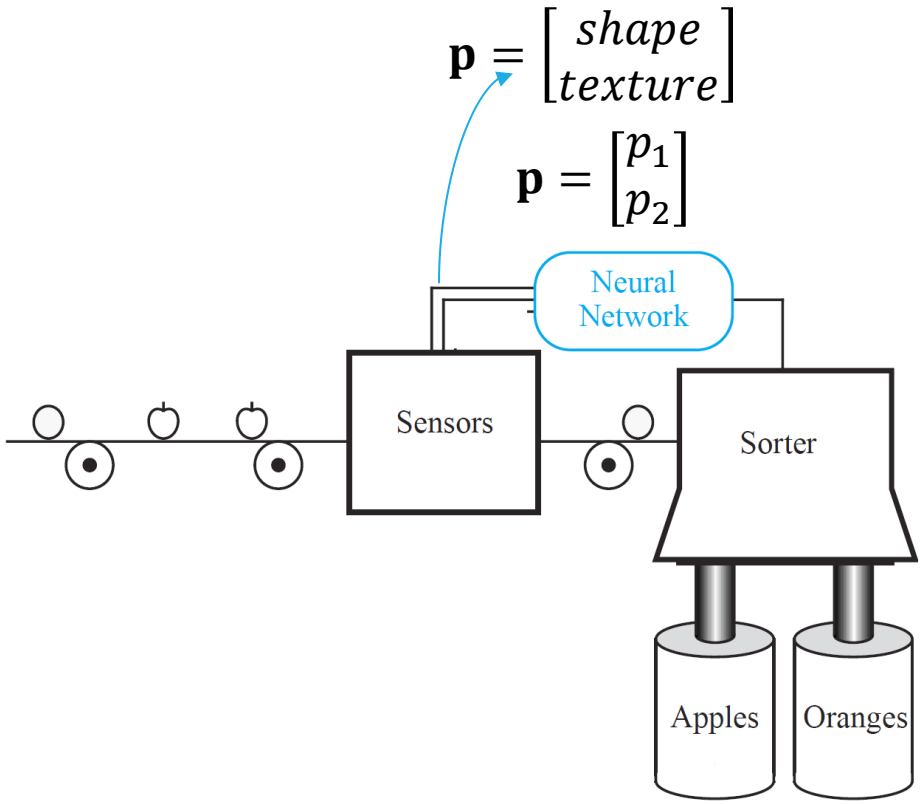
PERCEPTRON

- **Example: Single-neuron perceptrons**
  - Single-neuron perceptrons can classify input vectors into two categories.
  - For example, for a two-input perceptron

$$w_{1,1} = -1 \text{ and } w_{1,2} = 1$$

$$a = hardlims(n) = hardlims(\begin{bmatrix} -1 & 1 \end{bmatrix}\mathbf{p} + b)$$

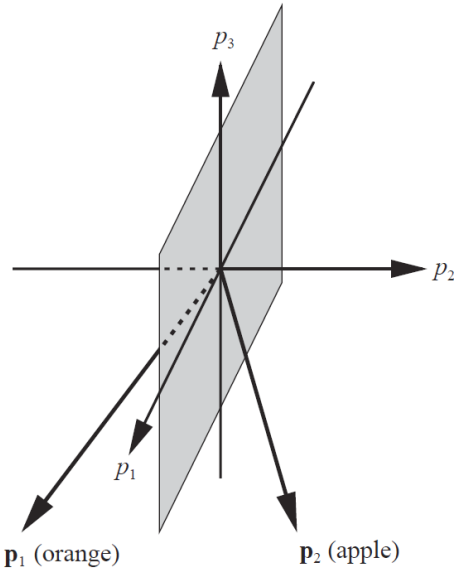




PERCEPTRON

- **Example: Single-neuron perceptrons (**for a three-input perceptron**)**

$$a = hardlims\left(\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \end{bmatrix}\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + b\right)$$



Orange:

$$a = hardlims\left(\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0\right) = -1(orange),$$

Apple:

$$a = hardlims\left(\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + 0\right) = 1(apple).$$

PERCEPTRON

- **Perceptron (learning rule):**
  - In the late 1950s, Frank Rosenblatt and several other researchers developed a class of neural networks called perceptrons.
  - Rosenblatt's key contribution was the introduction of a **learning rule** for training perceptron networks.
  - **He proved that his learning rule will always converge to the correct network weights, if weights exist that solve the problem.**
  - The perceptron could even learn when initialized with random values for its weights and biases.
  - Unfortunately, the perceptron network is inherently limited. They demonstrated that the **perceptron networks were incapable of implementing certain elementary functions**.
  - It was not until the 1980s that these limitations were overcome with improved **(multilayer) perceptron networks and associated learning rules**.

LEARNING RULES IN NEURAL NETWORK

➢ **Perceptron (learning rule):**

- By **learning rule** we mean a procedure for modifying the weights and biases of a network.
- The purpose of the learning rule is to train the network to perform some task. There are many types of neural network learning rules. They fall into three main broad categories:
  1. **supervised learning**,
  2. **unsupervised learning** and
  3. **reinforcement (or graded) learning**.

1. In **supervised learning**, the learning rule is provided with a set of examples (the training set) of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

   - Where p_q is an input to the network and t_q is the corresponding correct (*target*) output.
   - As the inputs are applied to the network, the network outputs are compared to the targets.
   - The learning rule is then used to adjust the weights and biases of the network in order to move the network outputs closer to the targets.

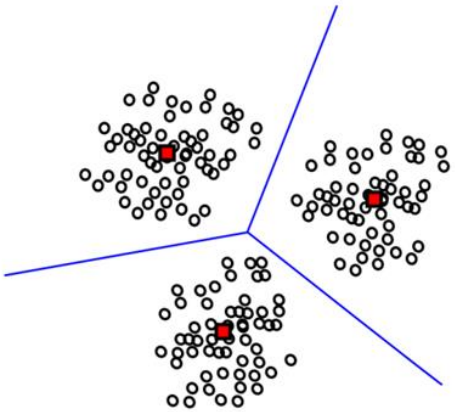
❖ **The perceptron learning rule falls in this supervised learning category.**

LEARNING RULES IN NEURAL NETWORK

- **Perceptron (learning rule):**

2. In **unsupervised learning**, the weights and biases are modified in response to network inputs only. There are no target outputs available.

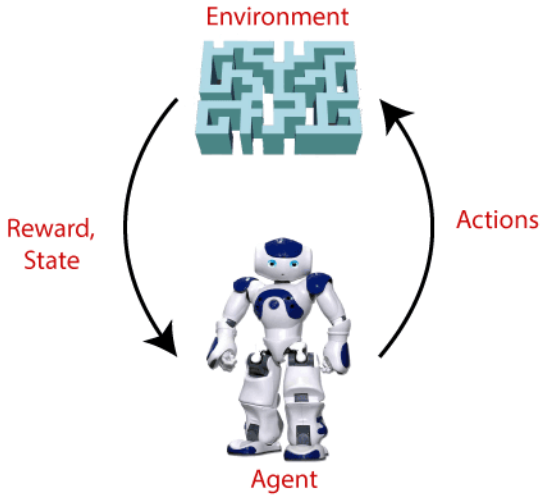
   Most of these algorithms perform some kind of **clustering** operation. They learn to categorize the input patterns into a finite number of classes.



LEARNING RULES IN NEURAL NETWORK

- **Perceptron (learning rule):**

3. **Reinforcement learning** is similar to supervised learning, except that, instead of being provided with the correct output for each network input, the algorithm is only given a grade. The grade (or score or reward) is a measure of the network performance over some sequence of inputs. This type of learning is currently much less common than supervised learning.



PERCEPTRON

- **Single-Neuron Perceptron (finding decision boundary graphically):**

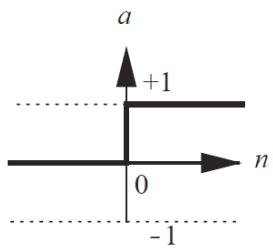
$$w_1 p_1 + w_2 p_2 + b = n$$

$$w_1 p_1 + w_2 p_2 + b = 0$$

$$p_1 = 0 \rightarrow p_2 = -\frac{b}{w_2}$$

$$p_2 = 0 \rightarrow p_1 = -\frac{b}{w_1}$$

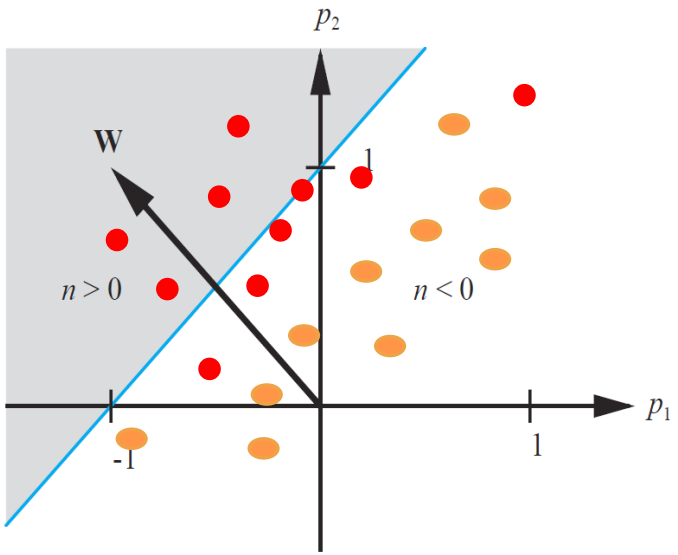
$$n = \begin{bmatrix} -1 & 1 \end{bmatrix} \mathbf{p} - 1 = 0$$



$a = hardlim(n)$

$a = 0 \quad n < 0$

$a = 1 \quad n \geq 0$



PERCEPTRON

- **Single-Neuron Perceptron (finding decision boundary graphically):**

$$a = hardlim(n) = hardlim(\mathbf{W}\mathbf{p} + b)$$

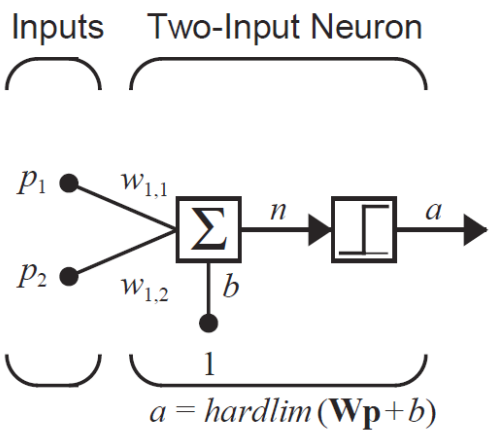
$$= hardlim({}_1\mathbf{w}^T\mathbf{p} + b) = hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b)$$

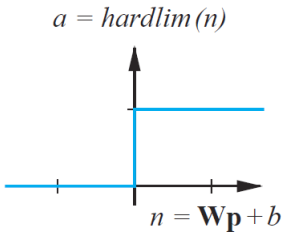
The decision boundary is determined by the input vectors for which the net input is zero:

$$n = {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b = 0$$

$$w_{1,1} = 1,\ w_{1,2} = 1,\ b = -1$$

$$n = {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b = p_1 + p_2 - 1 = 0$$





$$a = hardlim(n) = \begin{cases} 1 & if\ n \geq 0 \\ 0 & otherwise \end{cases}$$

PERCEPTRON

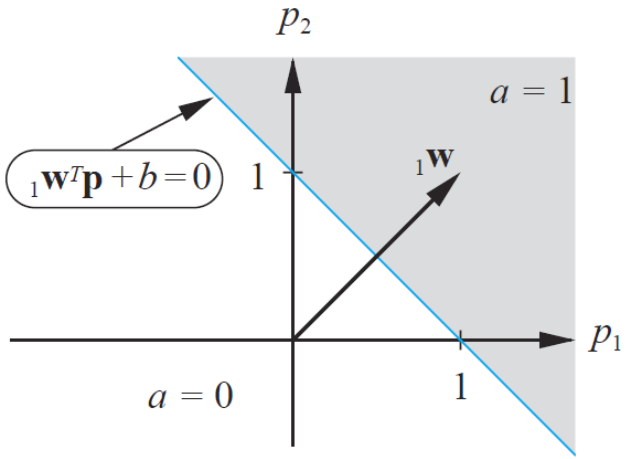
- **Single-Neuron Perceptron (finding decision boundary graphically):**

$$p_2 = -\frac{b}{w_{1,2}} = -\frac{-1}{1} = 1 \qquad \text{if } p_1 = 0$$

$$p_1 = -\frac{b}{w_{1,1}} = -\frac{-1}{1} = 1 \qquad \text{if } p_2 = 0$$

For the input $\mathbf{p} = [2 \quad 0]^T$, the network output will be

$$a = hardlim({}_1\mathbf{w}^T\mathbf{p} + b) = hardlim\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}2\\0\end{bmatrix} - 1\right) = 1$$

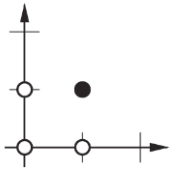


Decision Boundary for Two-Input Perceptron

PERCEPTRON

➢ **Single-Neuron Perceptron (finding decision boundary graphically):**

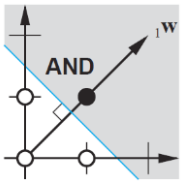
- design of a perceptron network to implement a simple logic function: the AND gate. The input/target pairs for the AND gate are



$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 0\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 0\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 1\right\}$$

- The first step of the design is to select a decision boundary. We want to have a line that separates the dark circles and the light circles.
- There are an infinite number of solutions to this problem. It seems reasonable to choose the line that falls “halfway” between the two categories of inputs, as shown in the adjacent figure.
- Next we want to choose a weight vector that is orthogonal to the decision boundary. The weight vector can be any length, so there are infinite possibilities. One choice is as displayed in the figure.

$${}_1\mathbf{w} = \begin{bmatrix}2\\2\end{bmatrix}$$



PERCEPTRON

- **Single-Neuron Perceptron (finding decision boundary graphically):**
  - Finally, we need to find the bias, *b*. We can do this by picking a point on the decision boundary and satisfying. If we use $\mathbf{p} = [1.5 \quad 0]^T$ we find

$$ {}_1\mathbf{w}^T\mathbf{p} + b = \begin{bmatrix} 2 & 2 \end{bmatrix} \begin{bmatrix} 1.5 \\ 0 \end{bmatrix} + b = 3 + b = 0 \quad \Rightarrow \quad b = -3 $$

  - We can now test the network on one of the input/target pairs. If we apply $\mathbf{p}_2$ to the network, the output will be

$$ a = hardlim({}_1\mathbf{w}^T\mathbf{p}_2 + b) = hardlim\left(\begin{bmatrix} 2 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} - 3 \right) $$

$$ a = hardlim(-1) = 0, $$

PERCEPTRON

Perceptron Learning Rule

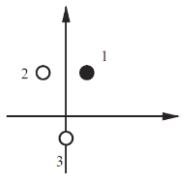
- This learning rule is an example of supervised training, in which the learning rule is provided with a set of examples of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

- Where $\mathbf{p}_q$ is an input to the network and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target.
- The learning rule then adjusts the weights and biases of the network in order to move the network output closer to the target.
- **Test Problem**

  The input/target pairs for our test problem are

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, t_1 = 1\right\} \left\{\mathbf{p}_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, t_2 = 0\right\} \left\{\mathbf{p}_3 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, t_3 = 0\right\}$$



PERCEPTRON

- **Perceptron Learning Rule**
  - **Test Problem**

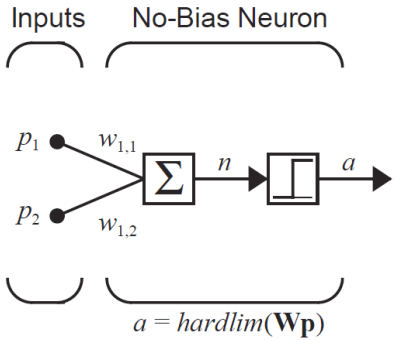
The network for this problem should have two-inputs and one output.

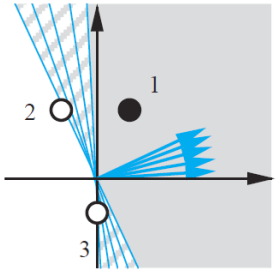
To simplify our development of the learning rule, we will begin with a network without a bias.

By removing the bias we are left with a network whose decision boundary must pass through the origin.

We need to be sure that this network is still able to solve the test problem. There must be an allowable decision boundary that can separate the vectors $\mathbf{p}_2$ and $\mathbf{p}_3$ from the vector $\mathbf{p}_1$. The figure illustrates that there are indeed an infinite number of such boundaries.

Test Problem Network





PERCEPTRON

- **Perceptron Learning Rule**
  - **Test Problem**

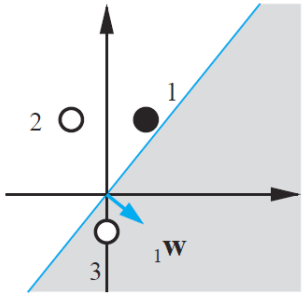
1. Training begins by assigning some initial values for the network parameters.

$$ {}_1\mathbf{w}^T = \begin{bmatrix} 1.0 & -0.8 \end{bmatrix} $$

2. We will now begin presenting the input vectors to the network. We begin with $\mathbf{p}_1$:

$$ a = hardlim({}_1\mathbf{w}^T\mathbf{p}_1) = hardlim\left(\begin{bmatrix} 1.0 & -0.8 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right) $$

$$ a = hardlim(-0.6) = 0 . $$

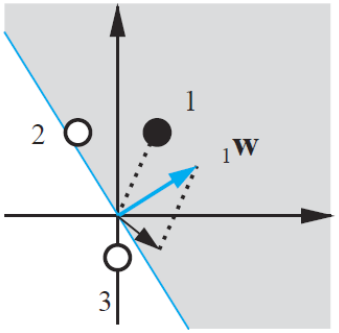


If $t = 1$ and $a = 0$, then ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}$

PERCEPTRON

- **Perceptron Learning Rule**
  - **Test Problem**

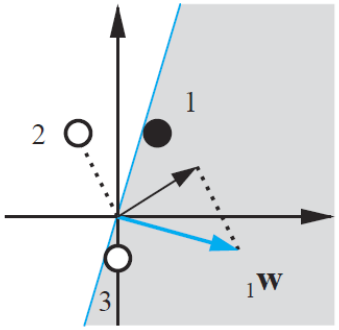
$$ {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}_1 = \begin{bmatrix} 1.0 \\ -0.8 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix} $$



$$ a = hardlim({}_1\mathbf{w}^T\mathbf{p}_2) = hardlim\left(\begin{bmatrix} 2.0 & 1.2 \end{bmatrix}\begin{bmatrix} -1 \\ 2 \end{bmatrix}\right) $$

$$ = hardlim(0.4) = 1 . $$

If $t = 0$ and $a = 1$, then ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}$



$$ {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}_2 = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix} - \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix} $$

PERCEPTRON

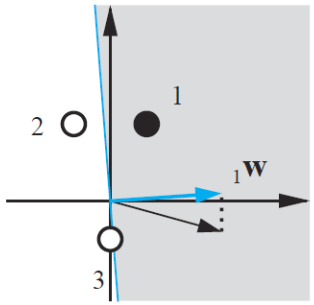
- **Perceptron Learning Rule**
  - **Test Problem**

$$a = hardlim({}_1\mathbf{w}^T\mathbf{p}_3) = hardlim\left(\begin{bmatrix}3.0 & -0.8\end{bmatrix}\begin{bmatrix}0\\-1\end{bmatrix}\right)$$

$$= hardlim(0.8) = 1 .$$

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}_3 = \begin{bmatrix}3.0\\-0.8\end{bmatrix} - \begin{bmatrix}0\\-1\end{bmatrix} = \begin{bmatrix}3.0\\0.2\end{bmatrix}$$

If $t = a$, then ${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}$.



PERCEPTRON

Perceptron Learning Rule

- **Test Problem**

$$\text{If } t = 1 \text{ and } a = 0, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}.$$

$$\text{If } t = 0 \text{ and } a = 1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}.$$

$$\text{If } t = a, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}.$$

- **Unified Learning Rule**

$$e = t - a \qquad {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + e\mathbf{p} = {}_1\mathbf{w}^{old} + (t-a)\mathbf{p}$$

$$\text{If } e = 1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}.$$

$$\text{If } e = -1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}.$$

$$\text{If } e = 0, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}.$$

PERCEPTRON

- **Perceptron Learning Rule**
  - **Test Problem**

This rule can be extended to train the bias by noting that a bias is simply a weight whose input is always 1. We can thus replace the input $\mathbf{p}$ in equation

$$_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + e\mathbf{p} = {}_1\mathbf{w}^{old} + (t-a)\mathbf{p}$$

with the input to the bias, which is 1. The result is the perceptron rule for a bias:

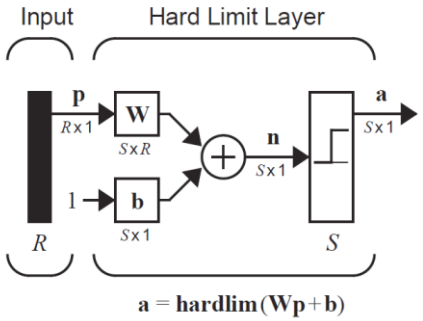
$$b^{new} = b^{old} + e$$

PERCEPTRON

- **Perceptron Learning Rule**
  - **Multiple-Neuron Perceptron**

Note that for perceptrons with multiple neurons, as in the Figure, there will be one decision boundary for each neuron. The decision boundary for neuron will be defined by

$$ {}_i\mathbf{w}^T\mathbf{p} + b_i = 0 $$



$$ \mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix} $$

$$ \mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix} $$

We will define a vector composed of the elements of the *i*th row of W:

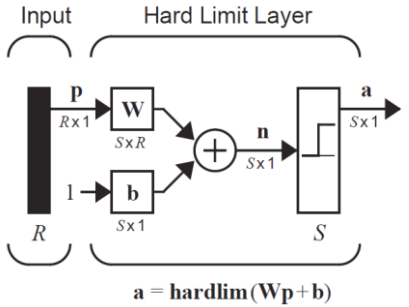
$$ {}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix} $$

Now we can partition the weight matrix:

$$ a_i = hardlim(n_i) = hardlim({}_i\mathbf{w}^T\mathbf{p} + b_i) $$

PERCEPTRON

- **Perceptron Learning Rule**
  - **Multiple-Neuron Perceptron**
    - A single-neuron perceptron can classify input vectors into two categories, since its output can be either 0 or 1.
    - A multiple-neuron perceptron can classify inputs into many categories. Each category is represented by a different output vector. Since each element of the output vector can be either 0 or 1, there are a total of 2^Spossible categories, where S is the number of neurons.
    - Therefore, if the inner product of the ith row of the weight matrix with the input vector is greater than or equal to $-b_i$, the output will be 1, otherwise the output will be 0. Thus each neuron in the network divides the input space into two regions.



PERCEPTRON

- **Perceptron Learning Rule**
  - **Training Multiple-Neuron Perceptrons**

$$ {}_i\mathbf{w}^{new} = {}_i\mathbf{w}^{old} + e_i\mathbf{p} $$

$$ b_i^{new} = b_i^{old} + e_i $$

$$ \mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T $$

$$ \mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e} $$

$$ \mathbf{W} = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix}, \; b = 0.5 $$

$$ a = hardlim(\mathbf{W}\mathbf{p}_1 + b) = hardlim\left(\begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0.5 \right) $$

$$ = hardlim(2.5) = 1 $$

$$ e = t_1 - a = 0 - 1 = -1 $$

$$ \left\{ \mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, t_1 = \begin{bmatrix} 0 \end{bmatrix} \right\} \qquad \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = \begin{bmatrix} 1 \end{bmatrix} \right\} $$

PERCEPTRON

- **Perceptron Learning Rule**

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} + (-1)\begin{bmatrix} 1 & -1 & -1 \end{bmatrix}$$
$$= \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix}.$$

$$b^{new} = b^{old} + e = 0.5 + (-1) = -0.5$$

$$a = hardlim\,(\mathbf{W}\mathbf{p}_2 + b) = hardlim\,(\begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + (-0.5))$$

$$= hardlim\,(-0.5) = 0$$

$$e = t_2 - a = 1 - 0 = 1$$

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} + 1\begin{bmatrix} 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix}$$

$$b^{new} = b^{old} + e = -0.5 + 1 = 0.5$$

PERCEPTRON

- **Perceptron Learning Rule**
  - **Training Multiple-Neuron Perceptrons**

$$a = hardlim(\mathbf{W}\mathbf{p}_1 + b) = hardlim\left(\begin{bmatrix}0.5 & 1 & -0.5\end{bmatrix}\begin{bmatrix}1\\-1\\-1\end{bmatrix} + 0.5\right)$$

$$= hardlim(0.5) = 1$$

$$e = t_1 - a = 0 - 1 = -1$$

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix}0.5 & 1 & -0.5\end{bmatrix} + (-1)\begin{bmatrix}1 & -1 & -1\end{bmatrix}$$

$$= \begin{bmatrix}-0.5 & 2 & 0.5\end{bmatrix}$$

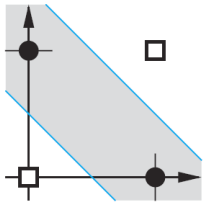
$$b^{new} = b^{old} + e = 0.5 + (-1) = -0.5$$

PERCEPTRON

Linear Separability

- The perceptron can be used to classify input vectors that can be separated by a linear boundary. We call such vectors linearly separable.
- The logical AND gate illustrates a two-dimensional example of a linearly separable problem.
- Unfortunately, many problems are not linearly separable. The classic example is the XOR gate. The input/target pairs for the XOR gate are

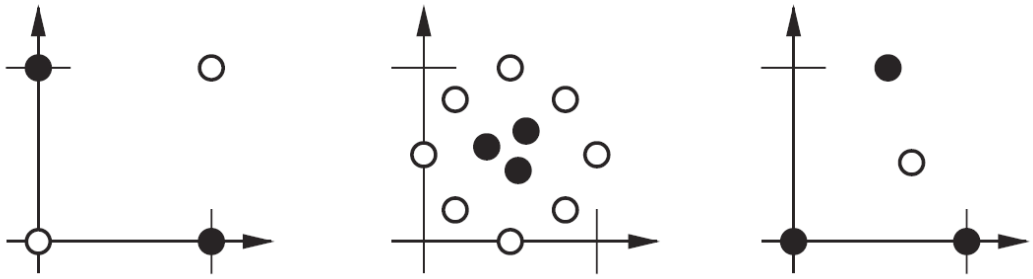
$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 1\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 0\right\}$$



PERCEPTRON

Linear Separability

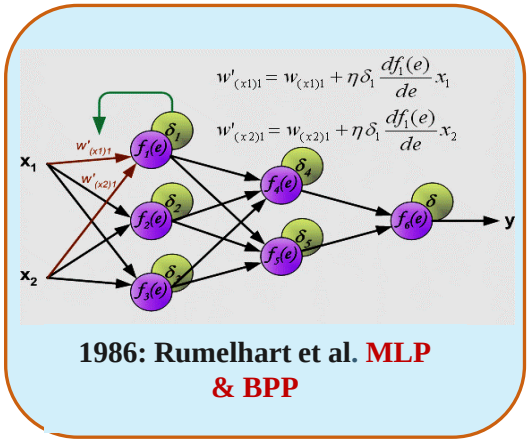
- Multilayer perceptrons can solve arbitrary classification problems, and will describe the backpropagation algorithm, which can be used to train them.

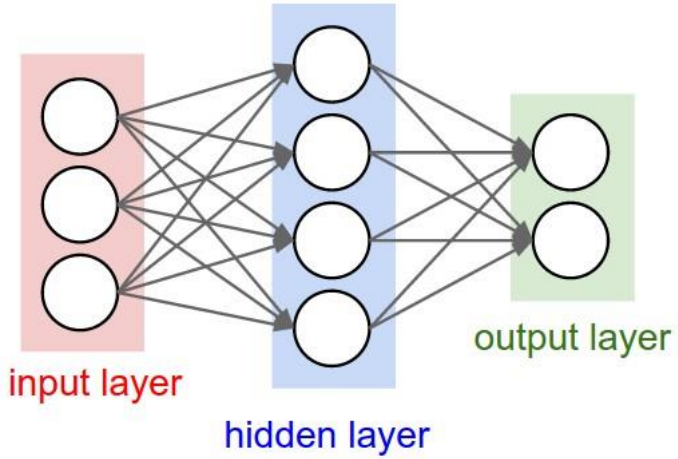


Linearly Inseparable Problems

PERCEPTRON

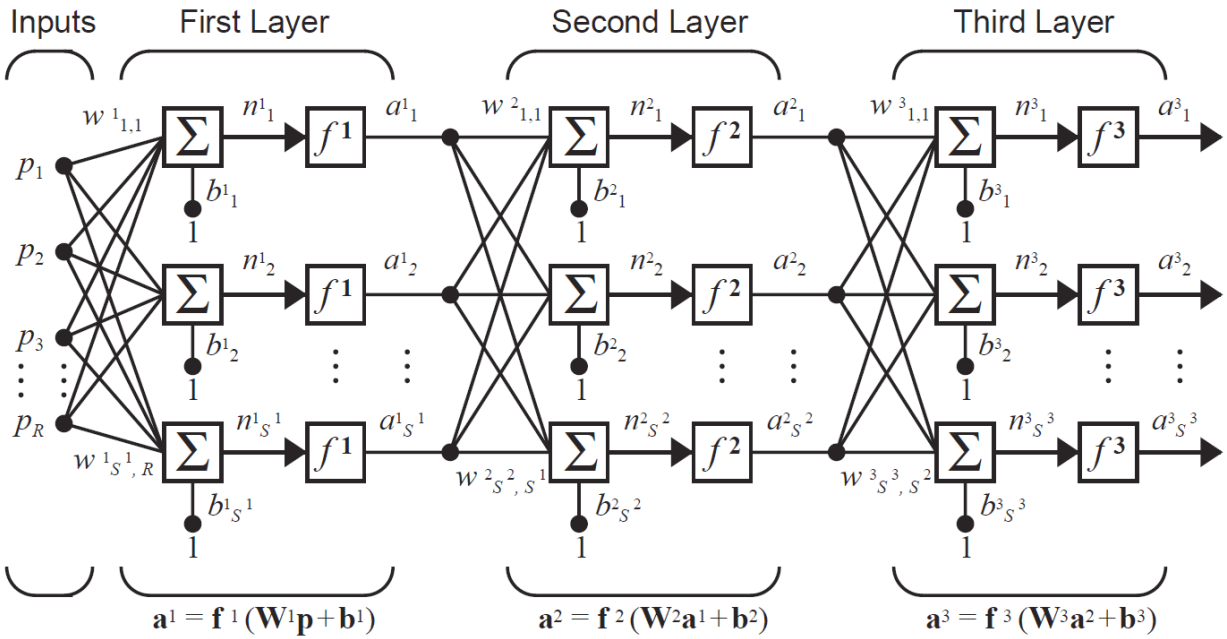
- **Multilayer Perceptrons:** An MLP consists of **at least three layers** of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a **nonlinear activation function**. MLP utilizes a supervised learning technique called **backpropagation** for training.





PERCEPTRON

- **Multilayer Perceptrons:** number of inputs is followed by the number of neurons in each layer: $R - S^1 - S^2 - S^3$

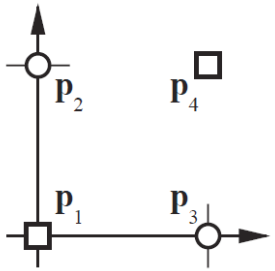


$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

Three-Layer Network

PERCEPTRON

- **Multilayer Perceptrons**
  - To illustrate the capabilities of the multilayer perceptron for pattern classification, consider the classic exclusive-or (XOR) problem. The input/target pairs for the XOR gate.
  - Because the two categories are not linearly separable, a single-layer perceptron cannot perform the classification.



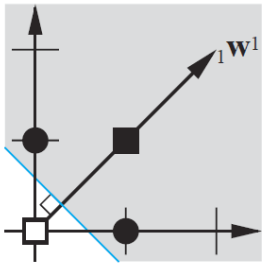
$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 1\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 0\right\}$$

PERCEPTRON

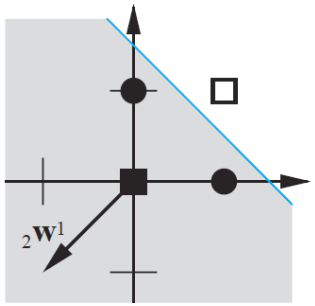
- **Multilayer Perceptrons**

  A two-layer network can solve the XOR problem. In fact, there are many different multilayer solutions.

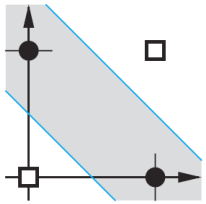
  One solution is to use two neurons in the first layer to create two decision boundaries. The first boundary separates p_1 from the other patterns, and the second boundary separates p_4. Then the second layer is used to combine the two boundaries together using an AND operation.

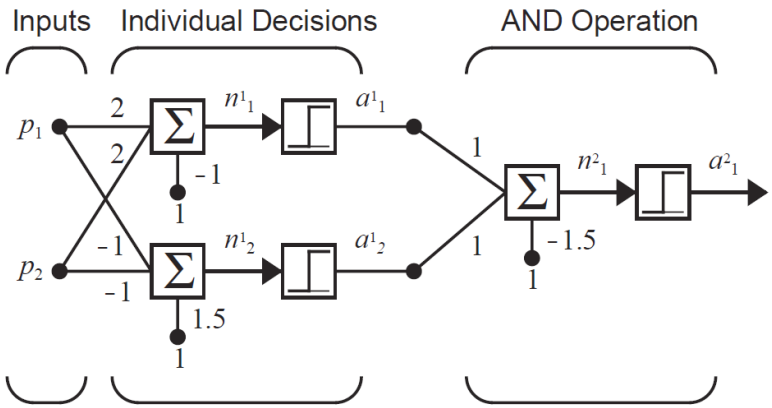


Layer 1/Neuron 1



Layer 1/Neuron 2

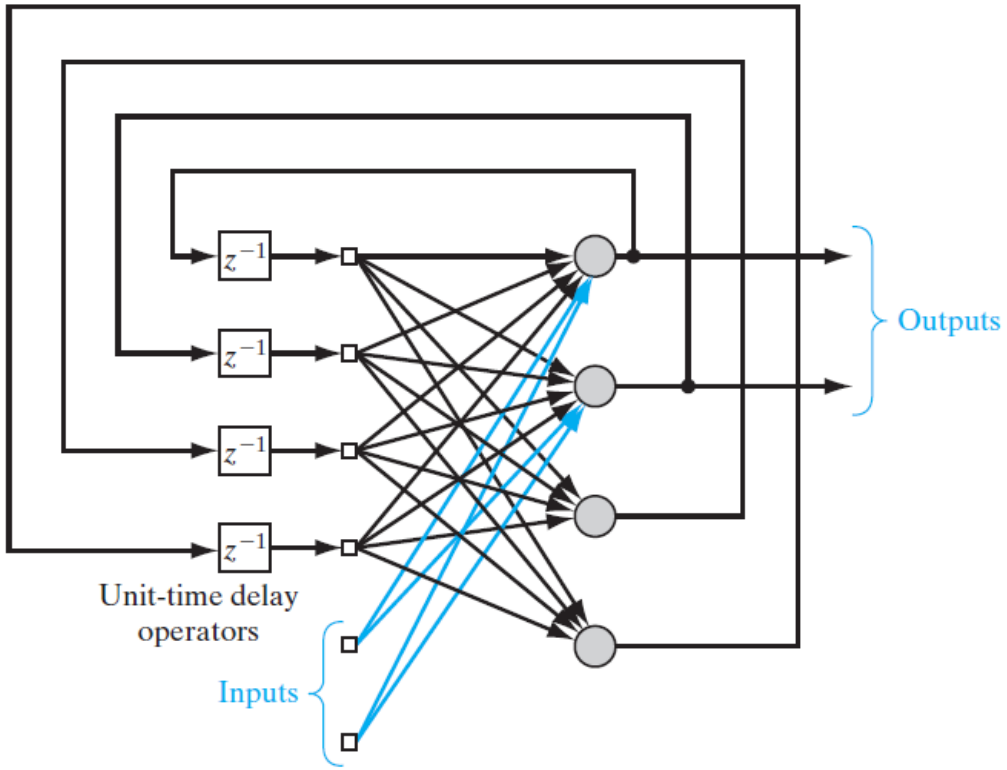




Two-Layer XOR Network

RECURRENT NETWORK

- Recurrent network with hidden neurons



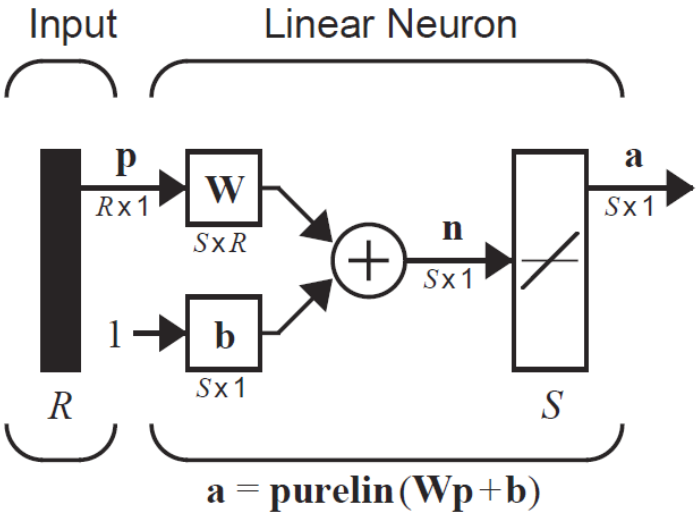
ADALINE

- **ADALINE Network**
  - The ADALINE network has the same basic structure as the perceptron network. The only difference is that it has a linear transfer function.

$$\mathbf{a} = \mathbf{purelin}(\mathbf{Wp} + \mathbf{b}) = \mathbf{Wp} + \mathbf{b}$$

$$a_i = purelin(n_i) = purelin({}_i\mathbf{w}^T\mathbf{p} + b_i) = {}_i\mathbf{w}^T\mathbf{p} + b_i,$$

$${}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}.$$



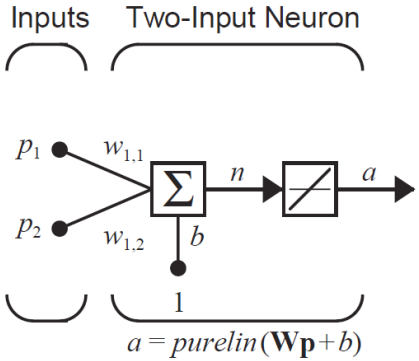
ADALINE Network

ADALINE

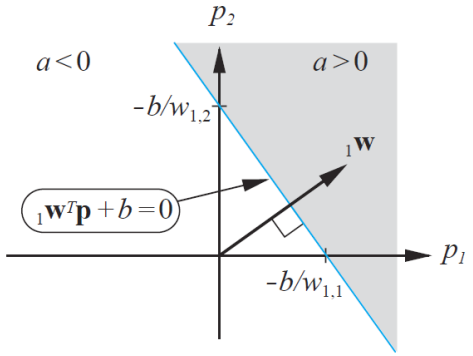
- **Single ADALINE**
  - The neuron output is greater than 0 in the gray area. In the white area the output is less than zero.
  - It says that the ADALINE can be used to classify objects into two categories. However, it can do so only if the objects are linearly separable. Thus, in this respect, the ADALINE has the same limitation as the perceptron.

$$a = purelin(n) = purelin({}_1\mathbf{w}^T\mathbf{p} + b) = {}_1\mathbf{w}^T\mathbf{p} + b$$

$$= {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b\ .$$



Two-Input Linear Neuron



Decision Boundary for Two-Input ADALINE

LMS ALGORITHM

Mean Square Error

- As with the perceptron rule, the LMS algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

  Where p_q is an input to the network, and t_q is the corresponding target output. As each input is applied to the network, the network output is compared to the target.

- The LMS algorithm will adjust the weights and biases of the ADALINE in order to minimize the mean square error, where the error is the difference between the target output and the network output.

  mean square error:

$$\hat{F}(\mathbf{x}) = (t(k) - a(k))^2 = e^2(k)$$

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE = mean squared error
n = number of data points
Y_i = observed values
$\hat{Y}_i$ = predicted values

- **Stochastic gradient**: When this is used in a gradient descent algorithm, it is referred to as "on-line" or incremental learning, since the weights are updated as each input is presented to the network.

LMS ALGORITHM

Stochastic gradient

- The LMS algorithm can be written conveniently in matrix notation:

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k) \qquad \mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix} \qquad \mathbf{z} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}$$

$$a = {}_1\mathbf{w}^T\mathbf{p} + b$$

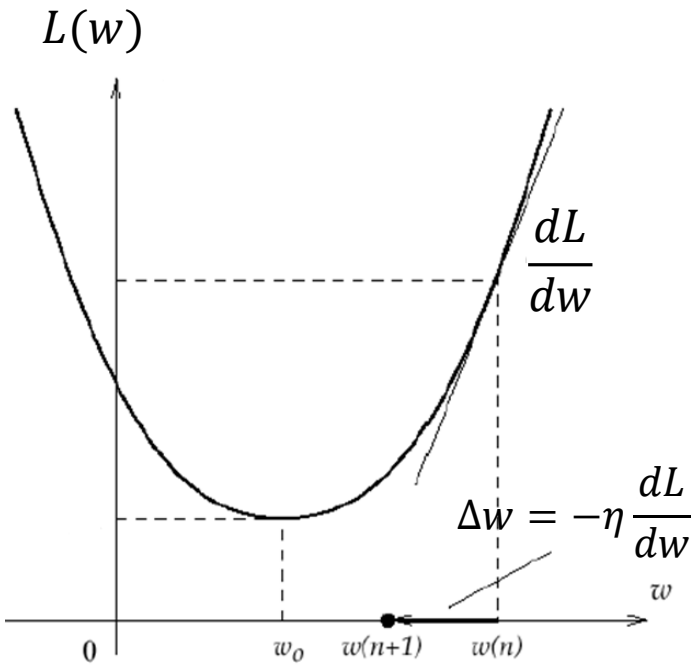
$$a = \mathbf{x}^T\mathbf{z}$$

- **Gradient Descent**

  $\boldsymbol{g} = \nabla L(\boldsymbol{w}), \qquad \boldsymbol{w}(n+1) = \boldsymbol{w}(n) - \eta\boldsymbol{g}(n)$

  where η is a positive constant called the stepsize, or learning-rate, parameter, and g(n) is the gradient vector evaluated at the point w(n). In going from iteration n to n+1, the algorithm applies the correction:

  $\Delta\boldsymbol{w}(n) = \boldsymbol{w}(n+1) - \boldsymbol{w}(n) = -\eta\boldsymbol{g}(n)$



LMS ALGORITHM

Stochastic gradient

- When this is used in a gradient descent algorithm, it is referred to as “on-line” or incremental learning, since the weights are updated as each input is presented to the network.

$$[\nabla e^2(k)]_j = \frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\frac{\partial e(k)}{\partial w_{1,j}} \quad \text{for } j = 1, 2, \ldots, R$$

$$[\nabla e^2(k)]_{R+1} = \frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}$$

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial [t(k) - a(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}[t(k) - ({}_1\mathbf{w}^T\mathbf{p}(k) + b)]$$

$$= \frac{\partial}{\partial w_{1,j}}\left[t(k) - \left(\sum_{i=1}^{R} w_{1,i}p_i(k) + b\right)\right]$$

where $p_i(k)$is the ith element of the input vector at the kth iteration. This simplifies to

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k) \qquad \frac{\partial e(k)}{\partial b} = -1$$

LMS ALGORITHM

Stochastic gradient

- Note that $p_j(k)$ and 1 are the elements of the input vector **z**, so the gradient of the squared error at iteration k can be written

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(k) = -2e(k)\mathbf{z}(k) \qquad \mathbf{z} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_k} \qquad \mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + 2\alpha e(k)\mathbf{z}(k)$$

$${}_1\mathbf{w}(k+1) = {}_1\mathbf{w}(k) + 2\alpha e(k)\mathbf{p}(k) \qquad {}_i\mathbf{w}(k+1) = {}_i\mathbf{w}(k) + 2\alpha e_i(k)\mathbf{p}(k)$$

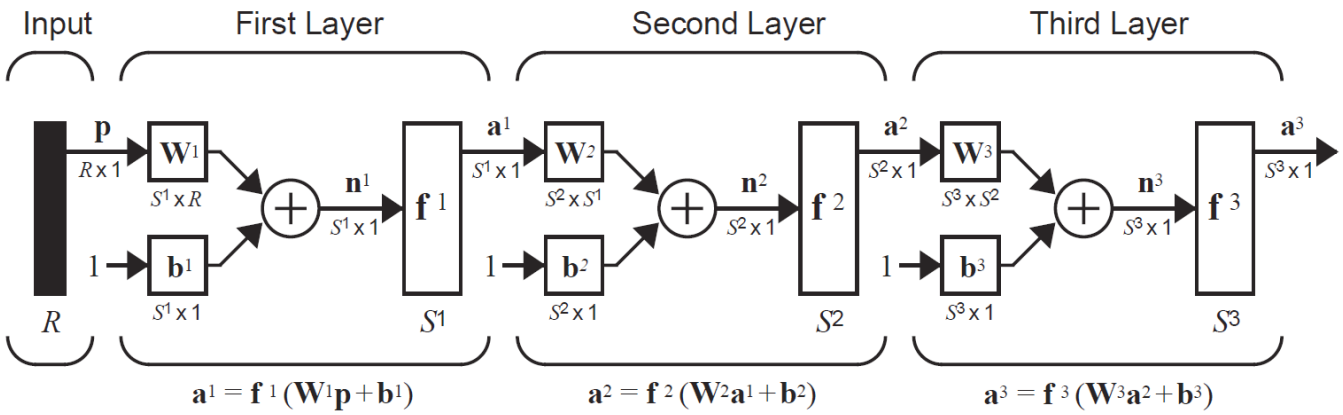
$$b(k+1) = b(k) + 2\alpha e(k) \qquad b_i(k+1) = b_i(k) + 2\alpha e_i(k)$$

BACKPROPAGATION ALGORITHM

- Generalization of the LMS algorithm called **backpropagation**, can be used to train **multilayer networks**.
  - As with the LMS learning law, backpropagation is an approximate steepest descent algorithm, in which the performance index is mean square error.
  - The difference between the LMS algorithm and backpropagation is only in the way in which the derivatives are calculated.
    - For a **single-layer** linear network the error is an explicit linear function of the network weights, and its derivatives with respect to the weights can be easily computed.
    - In **multilayer** networks with nonlinear transfer functions, the relationship between the network weights and the error is more complex. In order to calculate the derivatives, we need to use the **chain rule** of calculus.

BACKPROPAGATION ALGORITHM

- **Multilayer network**



BACKPROPAGATION ALGORITHM

- The backpropagation algorithm for multilayer networks is a generalization of the LMS algorithm, and both algorithms use the same performance index: mean square error.
  - The algorithm is provided with a set of examples of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

  - The algorithm should adjust the network parameters in order to minimize the mean square error:

$$F(\mathbf{x}) = E[e^2] = E[(t-a)^2]$$

where **x** is the vector of network weights and biases

$$F(\mathbf{x}) = E[\mathbf{e}^T\mathbf{e}] = E[(\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})]$$

BACKPROPAGATION ALGORITHM

$$\hat{F}(\mathbf{x}) = (\mathbf{t}(k) - \mathbf{a}(k))^T(\mathbf{t}(k) - \mathbf{a}(k)) = \mathbf{e}^T(k)\mathbf{e}(k)$$

where the expectation of the squared error has been replaced by the squared error at iteration *k*.

The steepest descent algorithm for the approximate mean square error (stochastic gradient descent) is

$$w^m_{i,j}(k+1) = w^m_{i,j}(k) - \alpha\frac{\partial\hat{F}}{\partial w^m_{i,j}},$$

$$b^m_i(k+1) = b^m_i(k) - \alpha\frac{\partial\hat{F}}{\partial b^m_i},$$

Where α is the learning rate.

So far, this development is identical to that for the LMS algorithm.

BACKPROPAGATION ALGORITHM

Chain Rule

- For the multilayer network the error is not an explicit function of the weights in the hidden layers, therefore these derivatives are not computed so easily.
- Because the error is an indirect function of the weights in the hidden layers, we will use the chain rule of calculus to calculate the derivatives.
- To review the chain rule, suppose that we have a function that is an explicit function only of the variable. We want to take the derivative of with respect to a third variable *w*. The chain rule is then:

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw}$$

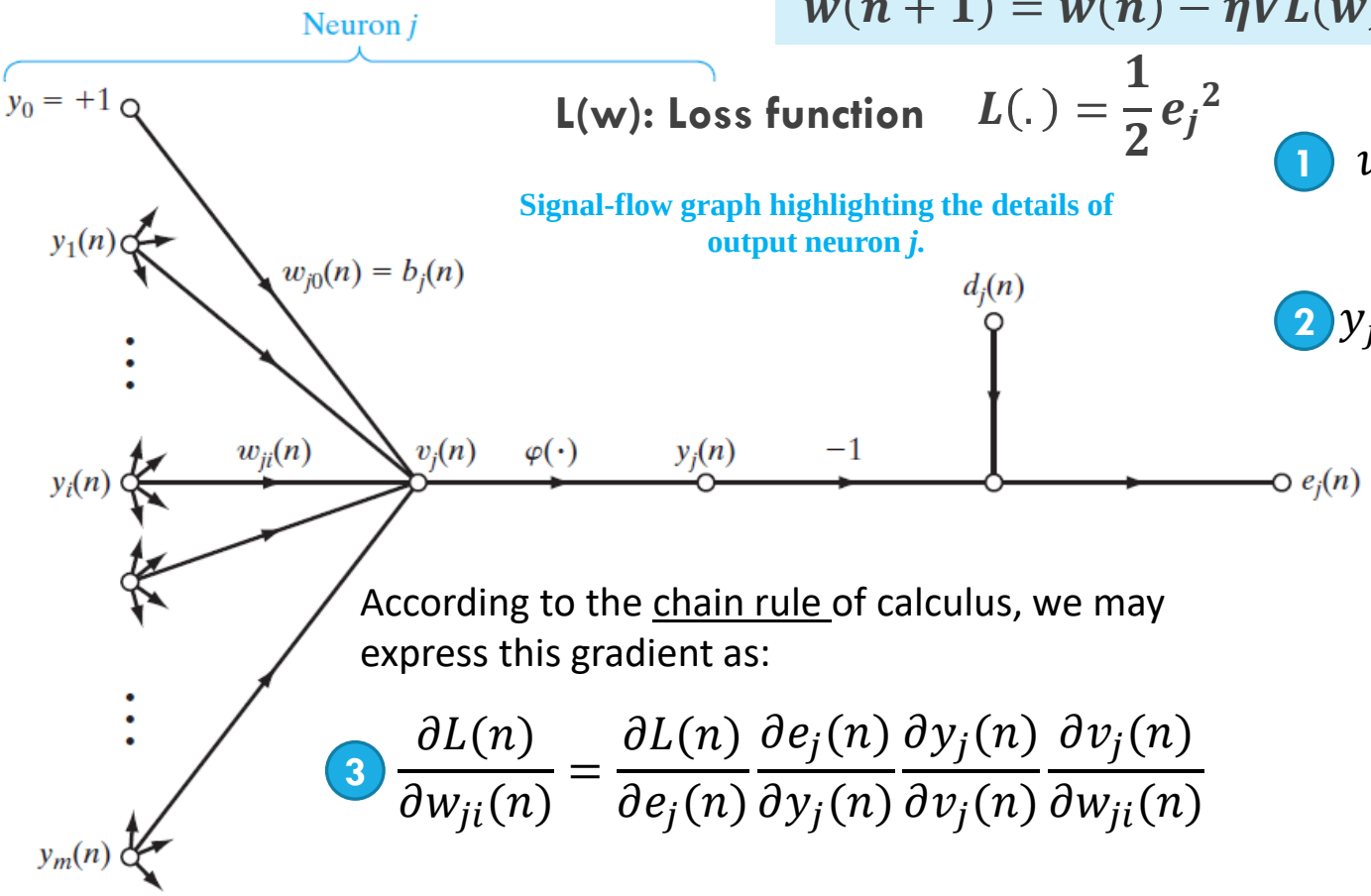
- For example, if

$$f(n) = e^n \text{ and } n = 2w, \text{ so that } f(n(w)) = e^{2w},$$

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw} = (e^n)(2).$$

BACKPROPAGATION ALGORITHM

$$w(n+1) = w(n) - \eta \nabla L(w)$$



L(w): Loss function $L(.) = \frac{1}{2} e_j^2$

Signal-flow graph highlighting the details of output neuron *j*.

1. $v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n)$
2. $y_j(n) = \varphi_j(v_j(n))$
4. $\frac{\partial L(n)}{\partial e_j(n)} = e_j(n)$

According to the chain rule of calculus, we may express this gradient as:

3. $\frac{\partial L(n)}{\partial w_{ji}(n)} = \frac{\partial L(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$

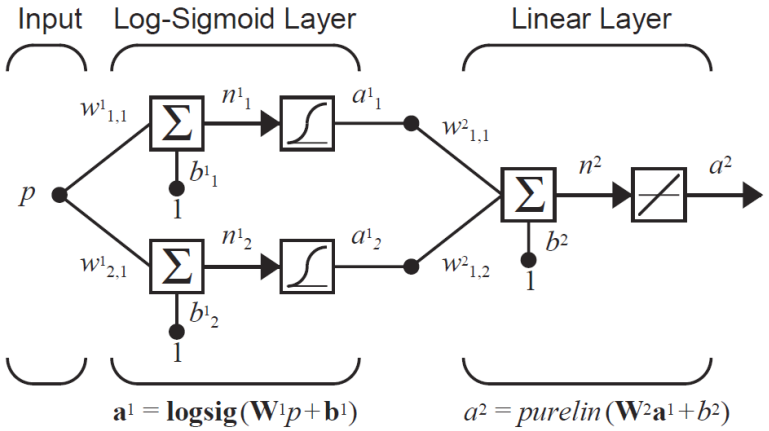
5. $\frac{\partial e_j(n)}{\partial y_j(n)} = -1$

6. $\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n))$

FUNCTION APPROXIMATION

- Up to this point in the text we have viewed neural networks mainly in the context of **pattern classification**.
- It is also instructive to view networks as **function approximators**.
- The following example will illustrate the flexibility of the **multilayer perceptron** for implementing functions.
- Consider the two-layer, 1-2-1 network shown in figure. For this example the transfer function for the first layer is log- sigmoid and the transfer function for the second layer is linear.

$$f^1(n) = \frac{1}{1+e^{-n}} \quad \text{and} \quad f^2(n) = n$$



Example Function Approximation Network

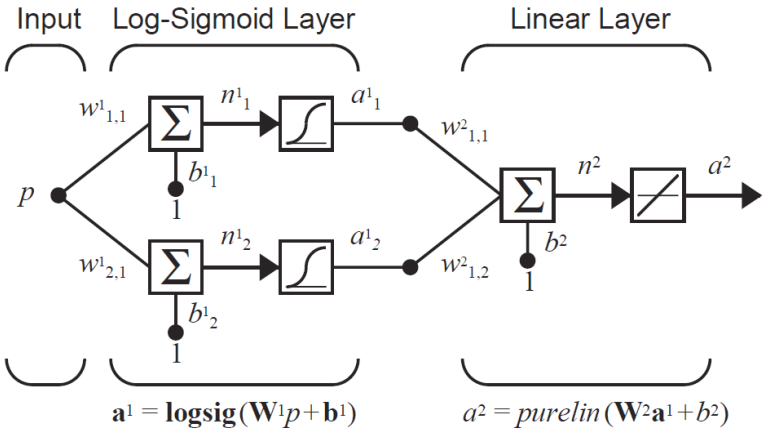
FUNCTION APPROXIMATION

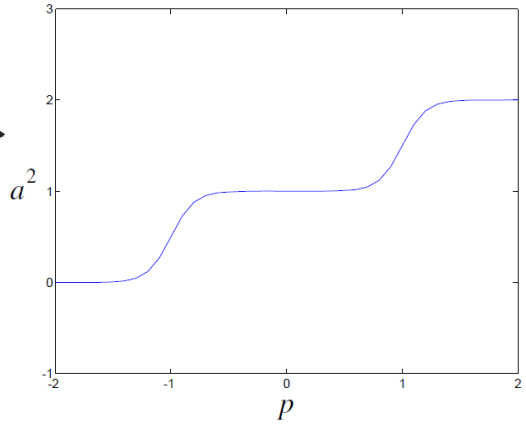
- Suppose that the nominal values of the weights and biases for this network are

$$w^1_{1,1} = 10,\ w^1_{2,1} = 10,\ b^1_1 = -10,\ b^1_2 = 10,$$

$$w^2_{1,1} = 1,\ w^2_{1,2} = 1,\ b^2 = 0.$$

- The network response for these parameters is shown in figure, which plots the network output as the input is varied over the range.





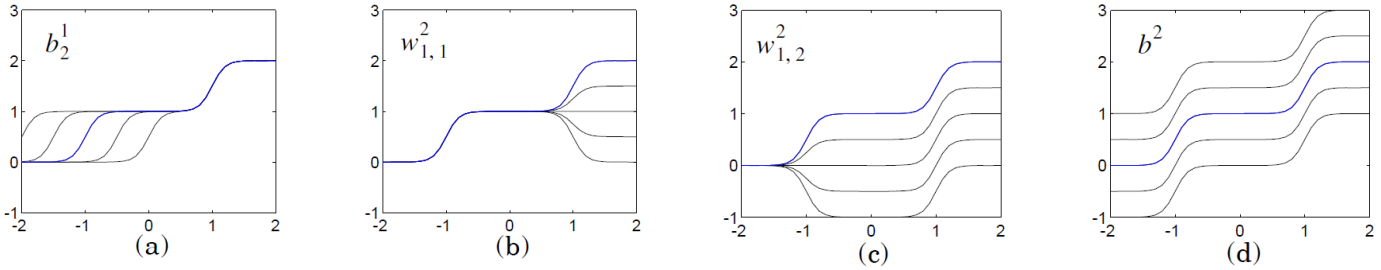
$$n^1_1 = w^1_{1,1}p + b^1_1 = 0 \Rightarrow p = -\frac{b^1_1}{w^1_{1,1}} = -\frac{-10}{10} = 1$$

$$n^1_2 = w^1_{2,1}p + b^1_2 = 0 \Rightarrow p = -\frac{b^1_2}{w^1_{2,1}} = -\frac{10}{10} = -1$$

$$f^1(n) = \frac{1}{1+e^{-n}} \text{ and } f^2(n) = n$$

FUNCTION APPROXIMATION

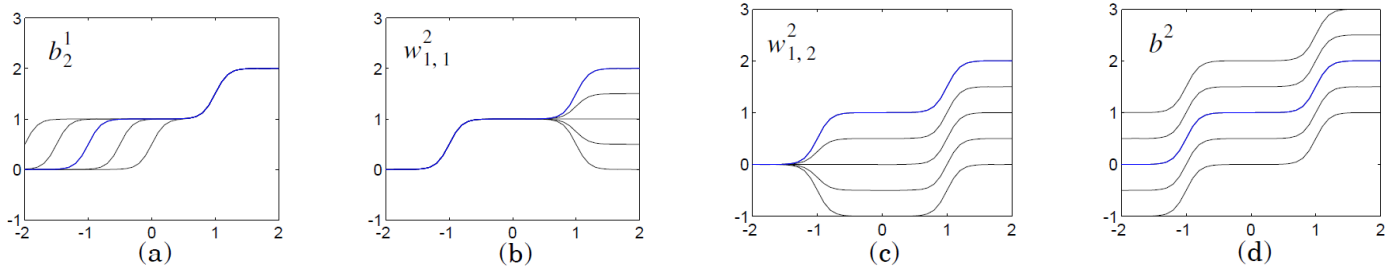
- Notice that the response consists of two steps, one for each of the log-sigmoid neurons in the first layer.
- By adjusting the network parameters we can change the shape and location of each step.
- Figure illustrates the effects of parameter changes on the network response. The blue curve is the nominal response. The other curves correspond to the network response when one parameter at a time is varied over the following ranges:



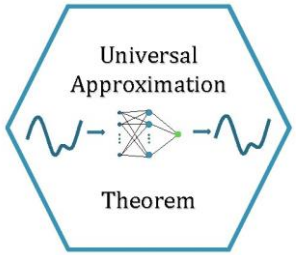
$$-1 \le w_{1,1}^2 \le 1\,,\ -1 \le w_{1,2}^2 \le 1\,,\ 0 \le b_2^1 \le 20\,,\ -1 \le b^2 \le 1$$

FUNCTION APPROXIMATION

- Figure (a) shows how the network biases in the first (hidden) layer can be used to locate the position of the steps.
- Figure (b) illustrates how the weights determine the slope of the steps.
- The bias in the second (output) layer shifts the entire network response up or down, as can be seen in Figure (d).
- If we had a sufficient number of neurons in the hidden layer. In fact, it has been shown that two-layer networks, with sigmoid transfer functions in the hidden layer and linear transfer functions in the output layer, can approximate virtually any function of interest to any degree of accuracy, provided sufficiently many hidden units are available.



$$-1 \le w_{1,1}^2 \le 1\,,\; -1 \le w_{1,2}^2 \le 1\,,\; 0 \le b_2^1 \le 20\,,\; -1 \le b^2 \le 1$$



UNIVERSAL APPROXIMATION THEOREM

Question: What is the minimum number of hidden layers in a multilayer perceptron with an input–output mapping that provides an approximate realization of any continuous mapping?

Answer: The answer to this question is embodied in the universal approximation theorem for a nonlinear input–output mapping, which may be stated as follows:

Universal Approximation Theorem

Let $\varphi(.)$ *be a nonconstant, bounded, and monotone increasing continuous function and* $\varepsilon > 0$, *there exist an integer* m_1 *and real constants* $\alpha_i, b_i,$ *and* $w_{i,j}$, *where* $i = 1, \dots, m_0$ *such that we may define*

$$F(x_1, \dots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i\right)$$

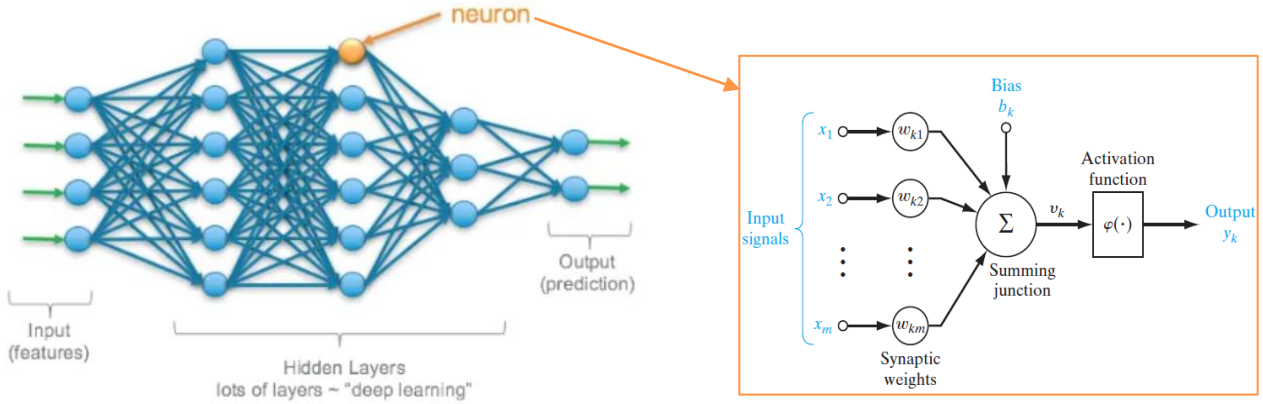
as an approximation realization of function $f(.)$; *that is,*

$$\left|F(x_1, \dots, x_{m_0}) - f(x_1, \dots, x_{m_0})\right| < \varepsilon$$

for all $x_1, x_2, \dots, x_{m_0}$ *that lie in the input space.*

ACTIVATION FUNCTION

- **Definition:** activation functions are simple transformations that are applied to the outputs of individual neurons in the network, introducing non-linearity to it and enabling it to learn more complex patterns.
  - Without activation functions, no matter how many layers of neurons we add to the network, it would still be limited in what it can learn because the output would always be a simple linear combination of the inputs.
  - By adding non-linearity, the network can model more complex relationships between the inputs and outputs, allowing it to discover more interesting and valuable patterns



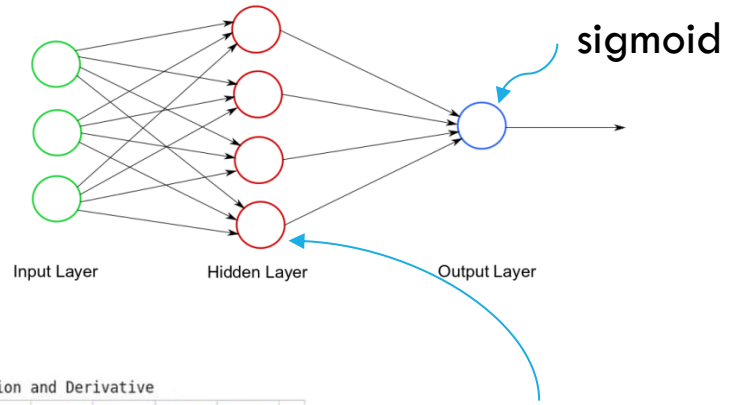
ACTIVATION FUNCTION

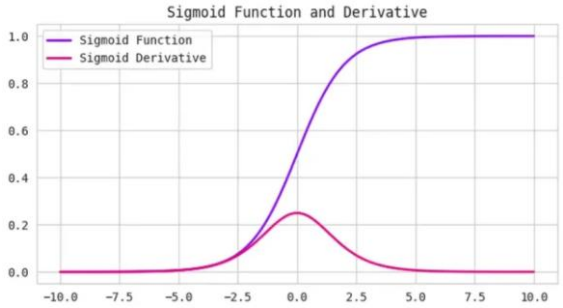
- **Different types of activation functions**
  - Binary function
  - Linear function
  - Non-linear functions
    - Sigmoid
    - Tanh
    - ELUs
    - ReLU
    - Softmax

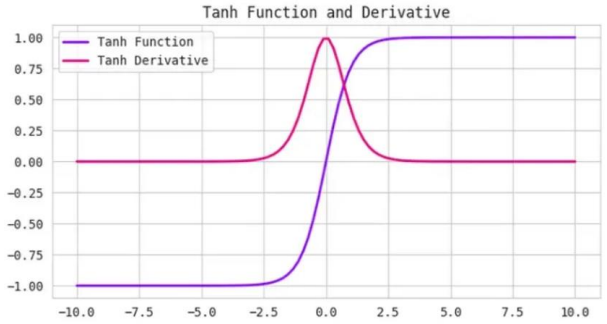
ACTIVATION FUNCTION

- Different types of activation functions
  - Sigmoid function

    As activation function of output layer in binary classification
  - Tanh







Sigmoid or Tanh?

Limitations of Sigmoid and Tanh

Sigmoid and tanh functions, however, face challenges during model training, particularly the vanishing and exploding gradient problems.

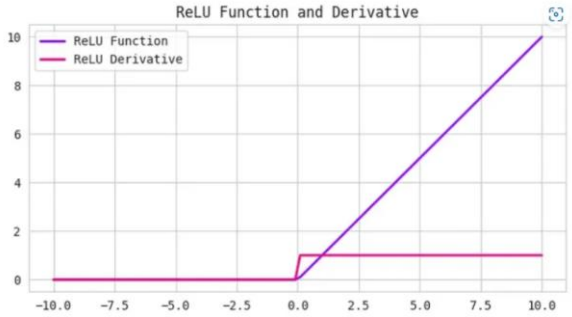
ACTIVATION FUNCTION

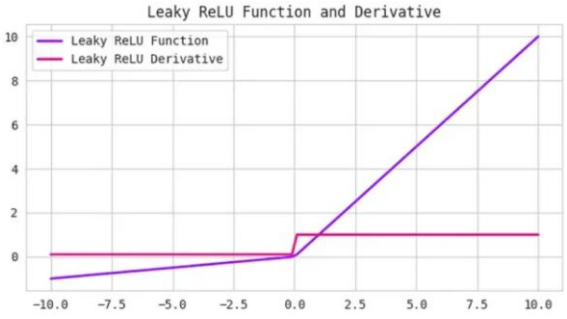
- Different types of activation functions
  - **Rectified Linear Unit, or ReLU**

    ReLU, is a common activation function that is both simple and powerful. It takes any input value and returns it if it is positive or 0 if it is negative. In other words, ReLU sets all negative values to 0 and keeps all positive.

  - **Leaky ReLU Function**

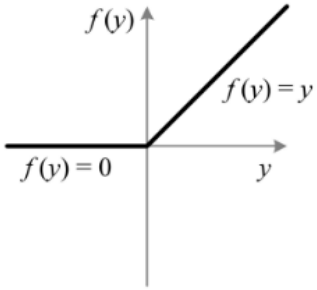
    The Leaky ReLU function is an extension of the ReLU function that attempts to solve the “dying ReLU” problem. Instead of setting all negative values to 0, Leaky ReLU sets them to a small positive value, such as 0.1 times the input value. his guarantees that even if a neuron receives negative information, it may still learn from it.

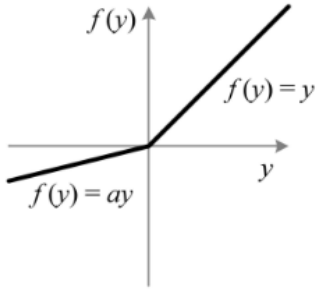




ACTIVATION FUNCTION

- Different types of activation functions
  - A **Parametric Rectified Linear Unit**, or **PReLU**, is an activation function that generalizes the traditional rectified unit with a slope for negative values. Formally:





$$f(y_i) = y_i \text{ if } y_i \geq 0$$

$$f(y_i) = a_i y_i \text{ if } y_i \leq 0$$

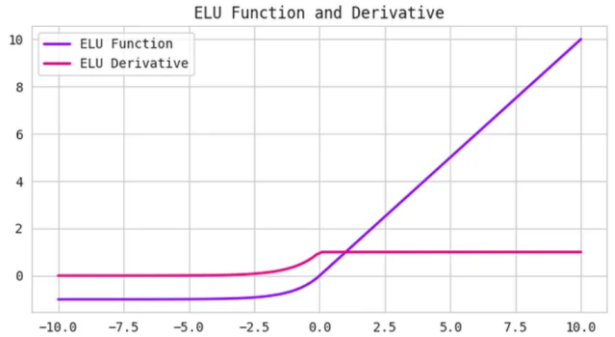
  - **ReLU vs. Leaky ReLU vs. Parametric ReLU**

Property	ReLU	LReLU	PReLU
Advantage	Solves gradient problems	Solves gradient problems	Solves gradient problems
Disadvantage	Dying relu problem	Inconsistent output for negative input	Fine-tune α
Hyperparameter	None	None	1
Speed	Fastest	Faster	Fast
Accuracy	High	Higher	Highest
Convergence	Slow	Fast	Fastest

ACTIVATION FUNCTION

- Different types of activation functions
  - **Exponential Linear Units (ELUs) Function**
    - Another form of activation function that has gained prominence in recent years is Exponential Linear Units or ELUs.
    - ELUs introduce a non-zero slope for negative inputs, which aids in the prevention of the “dying ReLU” problem

$$f(x) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$



    - Where alpha is a hyperparameter that controls the degree of negative saturation.
    - ELUs have been shown to improve both training and test accuracy compared to other activation functions like ReLU and tanh. They are particularly useful in deep neural networks that require a high level of accuracy

ACTIVATION FUNCTION

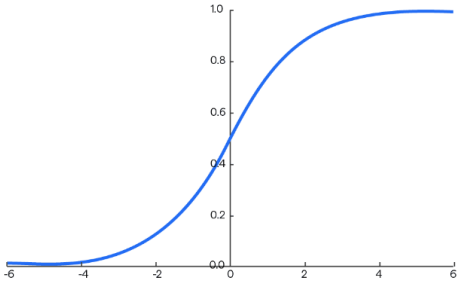
Softmax Function

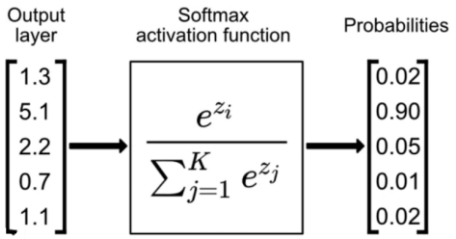
- The **softmax** function is often used as the activation function in the output layer of a neural network that needs to classify inputs into multiple categories.
- It takes as input a vector of real numbers and returns a probability distribution of each category

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

where x is the input vector and i and j are indices that range from 1 to the number of categories.

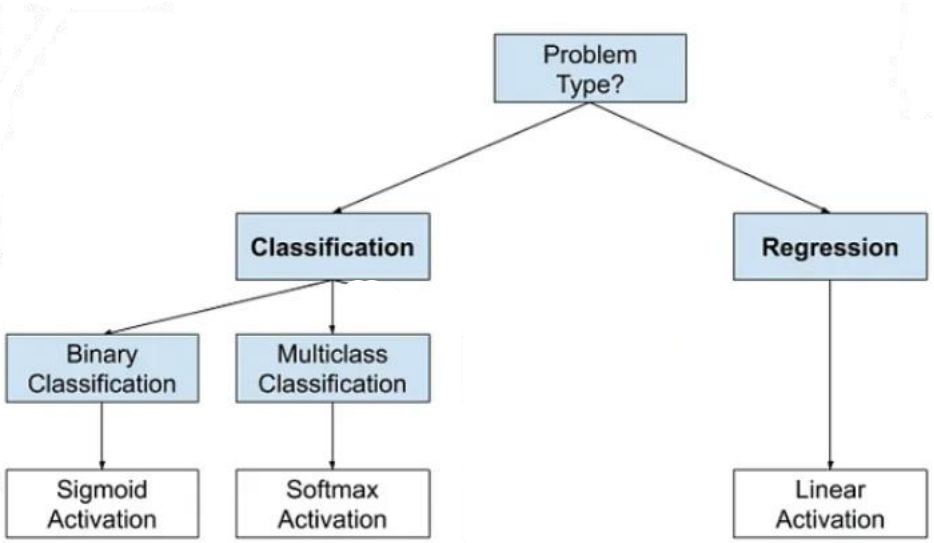
- Softmax is useful for **multi-class classification** problems because it ensures that the **output** probabilities sum to 1, making it easy to interpret the results. It is also differentiable, which allows it to be used in backpropagation during training.





ACTIVATION FUNCTION

- **How to choose output layer activation function?**



ACTIVATION FUNCTION

- **How to choose hidden layer activation function?**
  - **Sigmoid Activation Function**: Almost never used, except for the output layer in binary classification.
  - **Tanh Activation Function**: Strictly superior to the sigmoid function, especially for hidden units.
  - **ReLU Activation Function**: The default choice for most hidden layers, as it often leads to faster learning.
  - **Leaky ReLU Activation Function**: Similar to ReLU, but with a slight slope for negative values of z. Can be an alternative to ReLU.

ACTIVATION FUNCTION

How to choose hidden layer activation function?

- **Sigmoid** function it suffers from the vanishing gradient problem. This indicates that when the input becomes increasingly large or tiny, the gradient of the function becomes very small, slowing down the learning process in deep neural networks.
- Like the **Sigmoid** function, the **Tanh** function can also suffer from the vanishing gradient problem as the input becomes very large or very small. Yet, the **Tanh** function is still commonly used in neural networks, especially in the hidden layers of the network.
- One of the benefits of using **ReLU** is that it is computationally efficient and simple to implement. It is also known for helping to mitigate the vanishing gradient problem that can occur in deep neural networks.
- **ReLU** can suffer from a problem known as the "dying ReLU" problem. This happens when a neuron's input is negative, leading the neuron to output 0. If this happens too frequently, the neuron "dies" and stops learning.

ACTIVATION FUNCTION

- **Some tips for choosing an appropriate activation function**
  - **PReLU** has been shown to work well in some types of problems, particularly in image recognition tasks.
  - **ReLU** activation function should only be used in the hidden layers.
  - **Regression** — **Linear** Activation Function in output layer
  - **Binary Classification** — **Sigmoid**/Logistic Activation Function in output layer
  - **Multiclass** Classification — **Softmax** in output layer